Golfing Robot
Remote Communications Subsystem
Team A

Prepared for: Dr. William Rison
Dr. Kevin Wedeward

Prepared by: Mazhar Memon
Quinn Harris
Phuoc Le

May 6, 2002

New Mexico Tech
EE Department

Abstract

The following report will show the design and implementation of a remote communications system used to provide two-way wireless communications between a base station and an autonomous golfing robot. The purpose of the golfing robot is to locate and place a golf ball into a hole given the GPS coordinates of the ball and the hole. The golfing robot consists of four subsystems: chassis, ball / hole location, remote communication, and navigation. This groups objective is to develop the remote communications portion of the robot to display ball, hole and robot location as well as send GPS coordinates of the ball and hole to the robot. A base station transceiver is connected to the base station and sends data wirelessly to the remote transceiver. The remote transceiver is integrated with the navigation subsystem on the robot.

This report will cover a basic overview of the golfing robot project and each component of the remote communications subsystem. Enough specific design details of the hardware and software behind the communications subsystem will be covered to replicate this design.

List of Illustrations

Figures

Tables

Contents

1 Introduction of Robot project

The task for our team is to develop a 'golfing' robot capable of locating a golf ball given GPS coordinates, pickup the golf ball, locate the ball hole given GPS coordinates, transport the ball to the ball hole, and placing the ball into the ball hole. The robot is constructed by a team of four groups each responsible for one robot subsystem. The robot subsystems include: ball and hole location, smart chassis, robot navigation and location, and remote station communications. The navigation subsystem is responsible for navigating the robot to the ball and hole by calculating the desired course with GPS coordinates. The navigation subsystem is also responsible for sending status information to the communications subsystem such as robot location and status for display on the base station. Once the robot is within a few feet of the ball or hole, the ball-hole location subsystem is assigned the task of navigating the robot to the desired location. The smart chassis subsystem is assigned the task of moving the robot according to the demands of the navigation or ball-hole subsystem. The functional diagram is shown below:

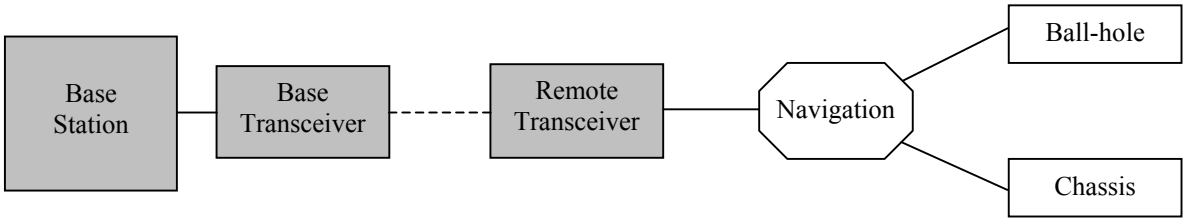


Figure 1: Functional diagram of robot project

Each subsystem must have a programmable device, the ability to communicate with other subsystems and I/O devices as needed. A certain amount of abstraction is required of each subsystem. For example, the navigation subsystem may command the chassis subsystem to travel two meters with a bearing of twenty degrees, but the chassis subsystem must take care of the lower level details such as speed control, slip compensation, etc. Each group will be provided an HC12 evaluation board microcontroller, multimeter, logic probe, wire strippers, proto-board, and other basic tools (e.g. screwdrivers, pliers, etc.). Our team is given one full semester to design and develop the robot to functionality.

2 Specifications and Requirements

Our group is assigned the task of developing the remote communications subsystem. Our subsystem must provide a means of two-way wireless communication between the robot and the base station capable of the desired data rate. The base station must be able to send ball and hole location as well as fundamental commands such as start and stop to the robot. The base station will have a graphical user interface (GUI) as a fronted showing ball, hole, and robot locations and other status information such as the current task of the robot (e.g. looking for ball) and link statistics.

3 Basic Overview of Design

The basic overview of the design of our remote communications system is depicted in the following figure. The remaining portion of this report will commence as depicted in the following graphic. There will be a overview of the requirements, interfaces, design, and implementation of the base station, base station transceiver, remote transceiver, and robot integration (in that order). Each overview will be followed by an in-depth discussion on each part with specific design details referenced in the appendices followed by a discussion of the communications protocol. The next two sections (Communications protocol usage and Communications protocol implementations) are not necessary information for the general user, but useful information for people desiring very specific implementation details of the communications protocol.

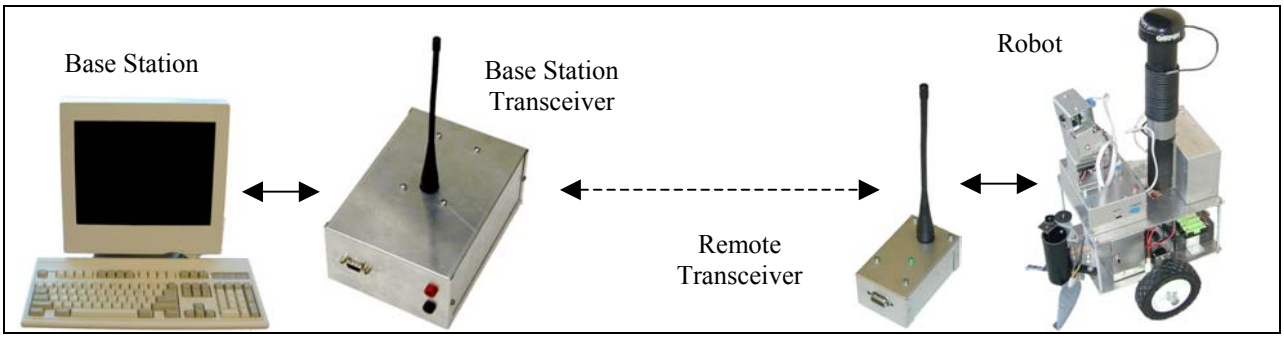


Figure 2: Basic Overview

4 Base Station

The base station contains a graphical user interface (GUI) that allows the user to interact with the robot. As described earlier, the GUI displays status information and is the interface by which the user can send GPS coordinates of the ball and hole. Below is a screen shot of the GUI:

Figure 3: Screenshot of GUI

The usage of the GUI is simple. To enter ball or hole coordinates, simply enter the longitude and latitude and specify which coordinate it is for (ball or hole) by clicking on the respective button. The display shows the positions of the robot, ball and hole. The status information of the robot is displayed on the left side. The slider bar on the

bottom allows the user to view different instances of time during the robots operation. The slider bar on the bottom right allows the user to view the history of the robot's operation at different speeds.

5 Transceiver Implementation

The transceiver consists of an ATA-100 transceiver module by ABACom technologies integrated on a printed circuit board (PCB) with six schmitt triggers, and an RC circuit. The schmitt triggers are used to clean the signal and power lines from the HC12 to the transceiver from any unwanted noise. The RC circuit is used to prevent any voltage spikes emanating from the HC12. Through many hours of debugging, we've concluded that it is absolutely necessary to regulate the +5V power drawn from the HC12 for the transceiver module.

It is important to note that the transceiver module that we're using transmits and receives data based on signal levels (high or low) and does not transmit at the byte level. Therefore, it is our job to devise a suitable digital encoding to transmit and receive data. We use pulse width modulation (PWM) and will be discussed further in section Error! Reference source not found. Error! Reference source not found.. Since the module operates at half-duplex, our communications protocol switches the transceiver between transmit and receive as needed. We must also provide our own error detection and correction. With this known, this transceiver module allows us a lot of freedom to implement whatever digital encoding and error recovery we desire.

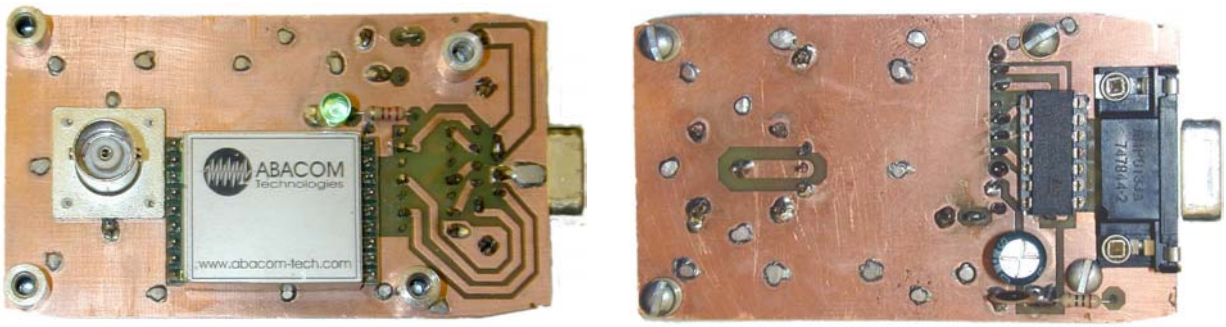
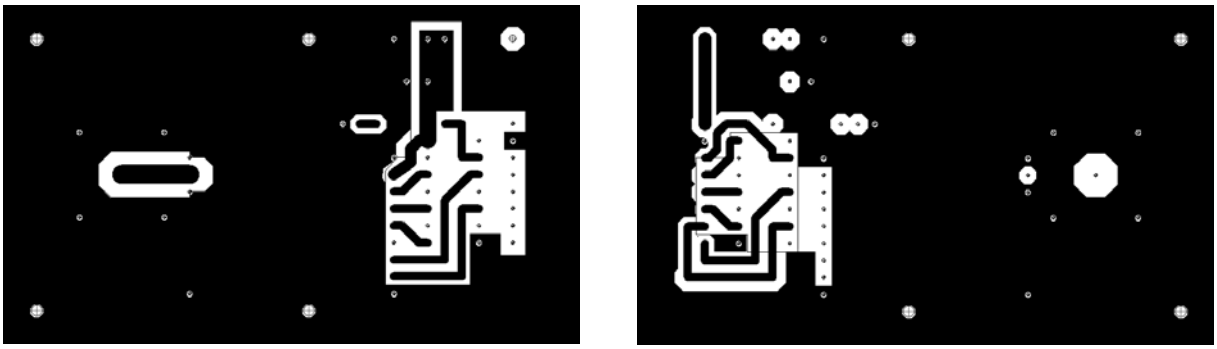


Figure 4: Photos of PCB (front and back)

Figure 5: PCB layout

5.1 Base Station Transceiver

The base station transceiver is connected to the base station through its serial port using an RS-232 cable. The RS-232 cable is connected directly to an HC12 which is connected to the actual transceiver through a nine pin inline connector. The base station transceiver does not contain a status LED as previously mentioned because all status information would appear on the base station GUI. The pin assignments between the HC12 and the base station transceiver is located in **16.1.1 Transceiver Module Pin-outs**.



Refer to **16.1 Transceiver Module Specs** for detailed transceiver module specifications.

5.2 Remote Transceiver

There is a very small difference between the remote and the base transceiver. Although the pin assignments between the HC12s and respective transceivers are identical, the PCB layout is slightly different. The base station transceiver does not contain a status LED. The LED status indicator shows whether or not a carrier is detected. The carrier detect is determined at a much higher level and will not be discussed further (see Base station protocol).

The pin assignments between the HC12 and the transceiver are identical to the base station transceiver and is located in **16.1.1 Transceiver Module Pin-outs**.

Refer to **16.1 Transceiver Module Specs** for detailed transceiver module specifications.

6 Robot Integration

Once the communications system has been fully assembled and tested, integration with the rest of the robot is necessary to establish the wireless link to the base station. As mentioned earlier (see **1 Introduction of Robot project**) all communication between robot subsystems is done directly through the navigation subsystem. Therefore, all communications are done either directly to the navigation subsystem or indirectly to other subsystems through navigation. The following discussion will first explain how internal communications between various subsystems occurs followed by why and how we communicate differently with the navigation subsystem. Finally, we will discuss what types of data we send and receive to and from the robot.

6.1 Intended protocol

The navigation subsystem provides all other subsystems with a very simple protocol over the microcontroller's SPI (serial .....). The navigation microcontroller is setup as a master which controls communications between itself and the other slave microcontrollers for each subsystem. Two input capture pins are allocated to each of the other three subsystems on the navigation microcontroller (one for read and one for write). A subsystem (other than navigation) wanting to write data on the bus would set its write line and send its data. Likewise, the read line would be set high when data is requested from navigation. The setting of the respective read and write lines are needed by navigation for arbitration and control over all internal communications. All multibyte messages are prefixed with a message ID which prevents confusion and missynchronization by other microcontrollers.

Consider, for example, a four byte message sent from the ball-hole subsystem to the chassis subsystem requesting a change in course towards a located ball. The first of the four bytes is a message ID specifying a course change while the next three bytes are heading and distance changes. The ball-hole microcontroller will put a byte on the SPI data register and set their write pin high triggering an interrupt service routine on the navigation microcontroller. The write pin will be set low after a short delay. The navigation module, after capturing the rising edge of the ball-hole write line, will supply the clock necessary to shift the data from the ball-hole SPI data register to navigation's SPI data register. The process repeats for any remaining bytes. To receive data from the SPI bus, the ball-hole subsystem will pulse its read line and wait for the SPI data register to fill and repeat for all subsequent bytes.

6.2 Actual protocol

Our communications protocol and digital encoding use up much of the processing power of the HC12, we modified our integration code to be run in a loop rather than in an interrupt service routine (ISR). This would mean that all communications with navigation would have to be run periodically in a low priority loop to avoid the performance penalty of an ISR. Such a low-priority function waiting to be executed is called a bottom half handler. Other bottom half functions were discussed previously.

6.3 Integration with Navigation

The navigation subsystem requires the GPS coordinates of the ball and the hole and start / stop commands from communications subsystem. All of these commands and information must come only after the calibration of the robot' s digital compass and initialization of all of the subsystems has occurred. Therefore, after the navigation subsystem has verified that all of the initialization and calibration is complete, a request for ball and hole GPS coordinates is sent to the communications subsystem. Next, the ball and hole GPS coordinates and a 'start' command are sent to navigation. The navigation subsystem now has all of the information it needs to complete its task. The communications module on the robot has the ability to read (sniff) all messages between all other subsystems because the navigation subsystem selects the slave line of communications and the subsystem receiving a message when messages are sent. All messages sent to the navigation subsystem is copied to the communications module once received. The communications module now has the ability to transmit these messages to the base station once the data is classified into atomics (see Software design above). The data sniffed on the robot is the primary source of status information of the robot. Such information includes: robot heading, ball hole targeting status, and whether the ball or the hole is being located.

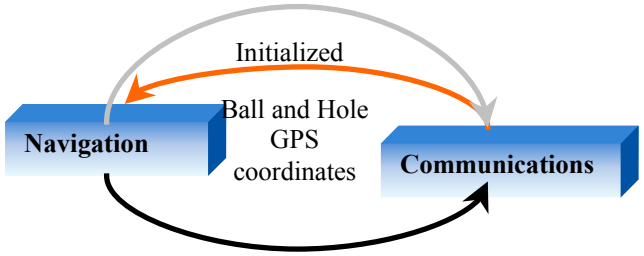


Figure 6: Robot Integration

7 Communication Protocol Overview

7.1 Design Objectives

- All of the data that needs to be transmitted is well structured and can be easily classified. This includes the GPS information consisting of latitude, longitude, height, and other GPS receiver status. It would therefore be beneficial to develop a protocol that understands the structure of the data. This will also aid the graphical user interface in displaying the data in a user friendly manner.
- The design should be reasonably flexible. Adding a new data to send should be as simple as possible and in no way break any existing code.
- A user is expected to be monitoring the data. It would therefore be beneficial if the data was received in a timely manner. For this reason, the design should attempt to keep all latencies under a 1/10th of a second.

7.2 Terminology

Atomic	A collection of data that must be transmitted at the same time. All data that is transmitted must be classified as an atomic
Message	The data of an atomic.
Packet	Contains a set of messages and error checking data.

The communication protocol attempts to keep the data classified as atomics on the transmitting end and synchronized with the data on the receiving end. To construct a packet, the protocol chooses the most important atomics with new data that have not been sent. This packet is then transmitted once it is the transceiver's turn to transmit.

8 Budget

Our expected budget determined at the midterm of the semester was $300. This figure was estimated to be our maximum budget use assuming the purchase of commercial transceivers. As shown below we are about $35 under budget. This budget also includes the parts used during a failed attempt at building a home made transceiver (listed as discrete components).

Item	Quantity	Unit Price	Subtotal
Tranceiver	2	$57.00	$114.00
Antenna	2	$10.80	$21.60
BNC connectors	2	$4.95	$9.90
Dbl sided circuit board	2	$12.35	$24.70
Sngl sided circuit board	1	$5.00	$5.00
DB9 connectors	2	$0.00	$0.00
Reset button	1	$1.34	$1.34
Enclosure (Small)	1	$4.50	$4.50
Enclosure (Large)	1	$12.00	$12.00
Banana plugs	2	$2.75	$5.50
Spacers, nuts, bolts			$2.25
Discrete components			$35.00
Shipping / Handling			$30.00
Total			**$265.79**

Table 1: Monetary Budget

8.1 Power Budget

Our final power budget (1.10 W) came to be .75W less than originally expected at midterm. We are therefore, under our power budget which is depicted below:

Component	Voltage	Current	Power
Transceiver	5.0 V	40 mA	0.20 W
HC12 + On board components	5.0 V	180 mA	0.90 W
Total		**210 mA**	**1.10 W**

Table 2: Power Budget

9 General software issues

A majority of the operations performed on the microcontroller is started in response to an external event. Most of these events are handled by interrupts, but the operation that must be performed as a result of the interrupt can be time consuming. For this reason most interrupts will signal a bottom half handler outside the interrupt to finish the action. This is necessary because other interrupts are blocked when an interrupt is being serviced. If an interrupt runs too long, other interrupts might not have an opportunity to run in a timely fashion. If the flag for a handler is raised, the handler will be executed in the main loop that is run any time the microcontroller is not in an interrupt.

The use of any operation that causes extensive delays (1ms) in the code execution will cause many interrupt and bottom halves not to be serviced in time. Therefore, it is inappropriate to use *printf*'s to debug.

All of the code for the microcontroller is included in one file using the #include directive. This allows the compiler to inline nearly every function as most of them are only called in one place. Despite this, the compiler sometimes chooses not to inline a function, probably because of their size.

The DBug12 and the boot loader on the microcontroller intercept interrupts thereby adding an additional 30 cycles to run. Placing the code for this project in flash memory overwriting DBug12 would allow this overhead to be reduced to the 6 clock cycles caused by the boot loader. The boot loader must be overwritten to completely avoid overhead clock cycles.

All of the code for this project was coded in C++ and compiled with GCC 3.0.4. Information about the HC12 port of GCC can be found at http://myweb.worldnet.net/~stcarrez/m68hc11_port.html

10 Communication Protocol Usage

This section describes how to extend the code base to send new atomics. This information is useful for individuals seeking to understand the details of the protocol code.

10.1 Atomic Implementation Overview

All the atomics that can be sent are represented by a structure named *SendStruct* that is instantiated once as *send*. Each atomic is a subclass of the *atomic* template class which provides a *commit* method to mark the data of the atomic as ready to be sent. The atomic template class also defines an atomic ID, a unique integer value used to identify each atomic. In addition, an *Atom* template class is provided to allow sending atomics consisting of only one elementary data type such as an integer, without the need to explicitly invoke the commit method. This provides a very simple and familiar interface for sending data as shown in the following example:

```
struct GPSCordinate :
       public Atomic<1> {     // Inherits Atomic class
        s32     latitude;
        s32     longitude;
};

struct SendStruct {
        Atom<0, u32>           count;  // u32 is the data type, an unsigned 32bit integer
        struct GPSCordinate    GPS;
        . . .
} send;

// The following line sends 5 as the value for count
send.count = 5;        // The commit method is implicitly executed

// The following lines send a GPS coordinate
send.GPS.latitude = 1;
send.GPS.longitude = 2;
send.GPS.commit();     // Commit the atomic after all the data has been sent
```

The commit method will queue the atomic to be sent. When an atomic is ready to be sent, a message will be constructed. Every class of atomics can have specialized functions for constructing a message. The appropriate function to use for a given atomic ID is defined by a function pointer in an array. This array also contains the position and size of the atomic data in the SendStruct. The following is a definition of this array for the preceding example:

```
SEND(send_u32, count),
SEND(send_GPS, GPS),
```

SEND is a macro that generates the appropriate data for the function pointer, position and size fields of the element in the array. The first parameter is the name of the function that generates the message for that type. The second parameter is the name of the element within the SendStruct structure. Notice that the order that the elements appear correspond with the atomic ID. This is necessary for proper functionality.

It would be possible to develop a preprocessor to automatically generate the SendStruct and the array, eliminating the need for the user to be concerned with proper assignment of atomic ID's and the construction of the array. It is

doubtful that a method exists that would retain the same functionality and performance of this method without a preprocessor. One alternative method considered would involve using a linker script and section attributes within the atomic class to construct the array.

The functions to generate a message are passed pointers to the data that needs to be sent and the data that has been known to have been sent. It is the function's responsibility to fill a buffer with the actual message to be sent and return whether the message contains all the data of the atomic or the difference. The function to send the unsigned 32bit integer is as follows:

```
bool send_u32(u32 *newData, u32 *oldData, u8)
{
        // Get the difference of the new and old data
        u32 diff = *newData - *oldData;

        // Determine if this difference can be expressed in one byte
        if (uOF(diff)) {
                // The data can not be expressed in one byte so send all the data
                add(*newData);
                return false;
        } else {
                // The data can be expressed in one byte so send only that difference.
                add((u8)diff);
                return true;
        }
}
```

The *uOF* macro returns *true* of the data cannot be expressed in one byte. The *add* macro copies the parameter to the buffer of the data to be sent. This macro will use the data type to determine how much data to copy.

Note: The last parameter is the size of the atomic that is to be constructed. This allows the use of only one function to generate a message for multiple atomics of similar structure but different size. For example, one function could be used for all integers no matter what size. This has not been done because the compiler had difficulty generating efficient code.

10.1.1 Adding new atomics

All atomics that are sent from the robot to the base station are contained within the SendStruct defined in "common/RobotSendStruct.hh" (p.41). The file "common/CommonStruct.hh" (p.40) contains some additional structures used in the SendStruct. In order to add a new atomic, an element must be added to the SendStruct that is inherited from the atomic template class. Note the Atom class inherits the atomic class. For example, to add a new atomic that is an unsigned integer simply add the following line inside the SendStruct, where UNIQUE ID is an unused atomic identifier less than 127.

```
Atom<UNIQUE ID, u32>  AtomicName;
```

Now a corresponding entry must be added to the "common/RobotSendArray.hh" (p.41) file. The atomic ID will define where in the file the entry should be added. Use the position of existing entries and there corresponding identifiers to determine the appropriate position for the new atomic entry, noting that an atomic with an ID number of one greater than another atomic will appear immediately after the other atomic. The entry will take the form:

```
SEND(function name, AtomicName),
```

The *AtomicName* is the same name as that used in the SendStruct. The function name is the function used to generate a message for the atomic.

If the type of atomic has not already been defined it will be necessary to create a new send function and a new class (or structure) that inherits the atomic class. The GPSStruct is an example of a specialized atomic. If a new specialized atomic is needed, it is suggested that the GPSStruct be used as an example to assist in its construction.

The new atomic will now need to be added to the graphical user interface code. Refer to section .. for an explanation of this process.

If an atomic needs to be added that will be sent from both the robot and base microcontroller, the *CommonStruct* in "common/CommonStruct.hh" (p.40) should be modified along with "common/CommonSendArray.hh" (p.40)

If an atomic needs to be added that will be sent from the link microcontroller, the SendStruct in "link/LinkSendStruct.hh" (p.56) should be modified along with "link/LinkSendArray.hh" (p.56).

Adding atomics to be sent from the computer to the robot requires the definition of functions that will be called upon receiving the message. The atomic must then be added to the SendStruct in "link/LinkRecvStruct.hh" (p.57) that will act as a buffer until that data can be sent. The elements within the structure do not need to inherit the atomic class unless the link itself needs to send data to the robot. The atomics to send must also be added to "link/LinkRecvArray.hh" (p.57). On the robot end, an entry must be added to "robot/RecvArray.hh" that references the function that handles the atomic in the "robot/RecvFunction.hh" (p.46) file.

11 Communications Protocol Implementation

This section describes the inner working of the communication protocol. It is useful to individuals wishing to understand and extend the core protocol.

The communication protocol handles sending the atomic data. It consists of the following four distinct layers starting from the highest level of abstraction:

Layer Name	Layer Description	Code Page
Atomic Pool	Determines what atomics to send from those that are ready	21
Messaging	Constructs a message representing the current data of an atomic	25
Packet	Combines multiple messages into one packet and verifies if the packet is successfully sent	28
Transceiver	Sends or receives a stream of bytes over the transceiver hardware	33

Table 3: Protocol Layers

Each layer of the protocol exposes a set of functions that are called by the layer immediately below it. The Transceiver layer controls the flow of instruction execution.

11.1 Atomic Pool Layer

11.1.1 Objective

The Atomic Pool must allow each atomic a fair chance to be sent. Yet, some atomics are more important than others. For this reason, some mechanism to give these important atomics a higher priority is desired. Generally the important atomics change less frequently. For example, the robots GPS coordinates will probably change once per second while the distance the wheels have traveled will probably change hundreds of times per second. Because the GPS coordinates are update less frequently, a single update will be more valuable than a single update of the distance the wheels have traveled. For this reason a GPS coordinate should be favored over wheel position when both are available to be sent. The Atomic Pool should therefore place a higher priority on atomics that were last sent a longer time ago.

11.1.2 Implementation Overview

Each atomic has a relative priority to the other atomics, represented by a doubly linked list. When an atomic is requested to be sent, the atomic pool algorithm finds the highest priority atomic that is waiting to be sent. When an atomic is successfully sent it is set to the lowest priority. If an atomic changes when it was already waiting to be sent, the data to send is updated but not the priority.

11.1.3 Implementation Details

```
namespace AtomicPool {
	// Class prototype
	class AtomicLink;

	// Initialize the AtomicPool
	static inline void init();

	// Add an Atomic to the pool by marking it as queued
```

```
        void add(AtomID);

        // Return if an Atomic has been queued
        static inline bool queued(AtomID);

        // Queue the Atomic as if get() didn't return it
        static inline void queue(AtomID);

        // Dequeue the Atomic
        static inline void dequeue(AtomID);

        // Return the Atomic with the highest priority
        static inline AtomID get();
        // Returned by get() to represent that no Atomic are queued
        const AtomID nullID = 0xFF;

        // Set the Atomic to the lowest priority
        static inline void put(AtomID);
};
```

- The AtomicLink class provides an abstraction of a doubly linked list with the sole purpose of defining the relative priority of atomics.
- The *add* function is called by whenever an atomic needs to be sent. The Atomic class provides an abstraction of the AtomicPool and the data buffers as described earlier in section 9.
- The *get* function is used by the Messaging layer to determine the next atomic to send. The Messaging layer will call *put* if the message data for the atomic is sent successfully and call *queue* if not.

11.2 Messaging Layer

11.2.1 Objective

The Messaging layer must construct an actual message to be sent. It would be advantageous for the message data to be as small as possible without requiring large amounts of computational power. This expectation arose from the assumption that the transceiver hardware would not support very high data rates, as would be the case if a commercial transceiver was not used.

11.2.2 Implementation Overview

Because of expected complexity and excessive computational demands, the use of a lossless compression was not perused. Instead, because of the atomic nature of the data, knowledge of the data already transmitted is used to construct a message containing the changes in the data. This provides a substantial reduction in the required data rate for any atomic larger than one byte. A buffer of the data known to have been successfully sent must be maintained. In order to keep this buffer updated, the Packet layer must determine if message data was properly sent.

The Messaging layer uses the message generation functions to construct a message that is added to the send_buffer which is shared between this layer and the packet layer. Every atomic sent will have its data and identifier copied to a message buffer. This buffer stores all the data that has been sent but is not yet known to have been received by the other end. If the data is confirmed to have been received, this buffer will be used to update the sent buffer. If the data is confirmed to have not been received, the message buffer will be cleared.

11.2.3 Implementation Details

```
namespace MessageBuffer {
        // Initialize the MessageBuffer
        static inline void init();

        // Compose a new message and place it in the send buffer
        static inline bool get();

        // Commit all the messages sense the last commit to the sent buffer.
        static inline void commit();
```

```
        // Requeue all the atomics and clear the message buffer.
        static inline void dump();
};
```

- The *get* function will place a message in the send buffer shared between the Messaging layer and the Packet layer. If the *get* function returns false, no new message were available to send.
- The *commit* function will be called if the last series of messages are known to have been sent. The *dump* function is called if the last series of messages is known to have not been sent.

11.3 Packet Layer

11.3.1 Objective

The Packet layer must send a set of messages in a packet. One packet will be sent with each transmission. The Packet layer must determine if the last packet sent was successfully received at the other end for use by the Messaging layer. In addition, the Packet layer must determine if the data received has been corrupted. This layer is responsible for informing the underlying Transceiver layer when all the data for a packet has been transmitted or received.

11.3.2 Implementation Overview

A packet consists of an identifier of the last packet received by the transmitting transceiver followed by a series of messages and a checksum. The checksum is preceded by a constant atomic identifier signifying the end of a packet. A packet is deemed invalid if the checksum is incorrect or the packet is malformed. A malformed packet either contains an invalid message identifier or exceeds a predefined maximum length. The Messaging layer will have no knowledge of receiving of an invalid packet. The Transceiver layer will be signaled to switch modes if the entire packet is received or the packet is determined to be invalid.

Every packet sent by a transceiver contains an identifier of the last packet successfully received by that transceiver. The identifier of a packet is simply the packet checksum. The identifier is used by the receiving transceiver to determine if the last or second from last packet was received. If a transceiver transmits without verifying what the last successfully transmitted packet was, a dummy packet will be sent. This packet contains no data, just the last packet received identifier and a checksum. The dummy packets identifier is not recorded as the last sent packet.

11.3.3 Implementation Details

```
namespace Packet {
        // Initialize the Packet layer
        static inline void init();

        // Return true if more data is availible for the current packet
        static inline bool send_availible();

        // Return a byte to send for the current packet
        static inline u8 send_byte();

        // Passed byte received
        static inline bool recv_byte(u8);

        // Called if receive sequence is prematurely terminated
        static inline void recv_term();
};
```

The *send_byte* is called to get the next byte to transmit. If send_byte returns the last byte of a message, a bottom half handler will be invoked which calls the get() function of the Messaging layer. This will construct the next message to send. This approach will construct a new message at the very last moment. The helps ensure all the data sent is recent.

11.4 Transceiver Layer

11.4.1 Objective

The Transceiver layer must transmit and receive a stream of bytes generated by the Packet layer. This layer could be implemented entirely in hardware.

This protocol is based on the assumption that data cannot be sent and received at the same time. It is therefore the responsibility of this layer to switch between the transmit and receive modes on the transceiver hardware as instructed by the Packet layer.

11.4.2 Implementation Overview

The transceiver layer uses pulse width modulation. The data is encoded by changing the duration of a half cycle. The input capture and output compare subsystems of the HC12 are used to encode and decode this signal.

A sync byte is used to detect the start of a packet. Once a sync byte is received, the Transceiver layer will relay each successive byte to the Packet layer.

11.4.3 Implementation Details

The following parameters define the shape of the pulse width modulated signal sent

- Bit Exponent - The number of bits encoded in each half cycle is 2^*b*, where *b* is the Bit Exponent. Therefore the number of possible half cycle duration is 2^(2^b). The Bit Exponent value must be between 0 and 3.
- Minimum Period - The duration of the shortest period. This value must be great enough to give the HC12 adequate time to process the last received half period.
- Period Exponent - The difference in duration between different values encoded in the half cycle period. The actual difference is 2^*p* where *p* is the Period Exponent. An exponent is used to allow quicker computation, there is no need to divide or multiply, bit shifting will work. The transceiver used requires this value to be between 6 and 12.

Maximum Transfer Rate	Minimum Transfer Rate	Average Transfer Rate
$\frac{c}{m}2^b$	$\frac{c}{m+2^{p(2\hat{}2\hat{}b-1)}}2^b$	$\frac{c}{m+2^{p(2\hat{}(2\hat{}b-1)-1)}}2^b$

Table 4: Transfer rate equations

11.5 Execution flow

11.5.1 Receiving data

The input capture interrupt handler in the Transceiver layer will wait for a sync byte to be received. The interrupt will be called multiple times to receive a single byte (as it takes multiple half cycles to encode a byte). After the sync byte is received, the Tranciever layer will call *recv_byte*(*u8*) of the Packet layer after receiving each full byte. The recv_byte(u8) function will determine if the packet is valid. If it is invalid, or the entire packet has been received, this function will return false which will cause the interrupt handler to switch the transceiver to send mode. Once a complete packet has been received, the recv_byte(u8) function will flag the *receive* bottom half handler of the Packet layer to run. This handler will use the last received packet identifier to invoke the commit() or dump() function of the Messaging layer. These functions will call put() or queue() from the AtomicPool respectively to update the atomic pool. The bottom half handler will then process the messages in the received packet by invoking the appropriate atomic receive function or marking the received data as ready to be relayed to the computer. This latter operation is abstracted by the RecvBuffer namespace.

On the base end, if the transceiver is in receive mode for a certain period of time, it will switch to send mode. The robot end will remain in receive mode until a sync byte possibly followed by a valid packet is received. This helps ensure that both ends do not transmit at the same time.

11.5.2 Sending data

The output compare interrupt handler in the Transceiver layer will be invoked after each half period is sent. For each byte to be sent, the *send_available*() function in the Packet layer will be called to determine if more data needs to be sent. If this returns true, the send_byte() function will be called to get a byte to send, otherwise the transceiver will be switched to receive mode. The send_byte() function will generate the last received packet identifier and the checksum. To generate the message data, the get() function of the Messaging layer will be called until it returns false signifying no more messages to send or the packet reaches its maximum length. This function will call get() of the AtomicPool layer to get the identifier of the atomic to send. The appropriate message generation function will be invoked to create the message.

12 GPS integration

The robot microcontroller code can parse the NMEA GPGGA sentence send over the serial port from the GPS receiver. The data of the sentence is stored in a buffer by the interrupt handler for the serial port. Once the line goes idle, a bottom half handler is invoked to process the sentence in the buffer. In order to expedite execution, a hash table is used to look up the weight of each ascii character in a decimal number that is part of the sentence. This allows for quick parsing with the ability to send a 32bit integer representing latitude, longitude and height. The same mechanism described above for sending the data is used.

13 Serial Interface

The base microcontroller uses the serial port to communicate with the computer. Interrupts are used to both send and receive data over the serial port. The mechanism for sending data from the base transceiver to the computer is virtually identical to the above communications protocol with a single simpler layer to replace the Packet and Transceiver layer. The new layer features a pass through mechanism that will send data received through the transceiver that has been confirmed to be valid directly to the computer without processing or changing it. The computer will receive a stream of messages just like those sent in a packet consisting of a message identifier followed by the message data.

Data sent from the computer takes the same form as that received without the difference feature. The computer must wait for an acknowledge message for each byte of data that it sends. This provides flow control that ensures that the computer does not send data to the microcontroller faster than it can receive it. This was found to be necessary as an implementation without it dropped data. The data sent to the link from the computer is buffered and the atomic is committed to be sent.

14 Graphical User Interface

14.1 Overview

The graphical user interface was programmed in C++ using Qt 3.0. Information about Qt can be found at http://www.trolltech.com/

The code for the graphical user interface contains some code to access the serial port that is specific to Linux. For this reason it might not work on other Unix variants and is guaranteed not to work on Windows 9x/NT or Macintosh OS X. It would be feasible to modify this code to support these platforms code to access the serial port is produced for each platform.

The code was compiled using Qt 3.0.3. Qt 3.0 or later is required with Qt thread support enabled.

In order to receive data over the serial port, the GUI code creates a separate thread. This thread reads the data from the serial port and stores the data as defined by the *AtomicDataSet class*. The data is then displayed by the GUI.

14.2 Adding new atomics

Every atomic that the computer receives must be created and added to the recv_class_array in the “command/atomicCollection.cc” (p.72) file. The atomic identifier defines the position in this array just as it did in

the array for the microcontroller code explained in **Adding new atomics** on page 12. The object created must be connected to the GUI with Qt signals and slots.

15 Conclusion

The final status of the remote communications subsystem is that the base station displays the location and heading of the robot as well as the locations of the ball and hole. Our communications system sends the ball and hole GPS coordinates at the request of the navigation subsystem (see **6. Robot Integration**). The GUI displays the actual GPS coordinates of the robot and has the ability to do basic logging of the robot's history. Also included as a tab on the GUI is a list of link statistics.

16 Appendices

16.1 Transceiver Module Specs

16.1.1 Transceiver Module Pin-outs

ABACom ATRT100 – Transceiver Module Pin Assignments

Pin 1-3	RF Ground
Pin 2	50Ω impedance antenna connection
Pin 5,9,10,18	Ground
Pin 11	Carrier Detect (active low)
Pin 12	Receiver data out
Pin 13	Analog output of FM detector
Pin 14	Input to transmitter (0-5V)
Pin 15	Transmit Enable (active low)
Pin 16	Receive Enable (active low)
Pin 17	Vcc (+5V ± 10%)

Table 5: Pin Assignments

16.2 Base Station and Remote Transceiver

16.2.1 Transceiver Pin-outs

As discussed in the text, a cable having the 9-pin line connector on both sides is connected to the transceiver PCB and to the HC12 microcontroller using the pins depicted in the table below. The nine connections from the 9-pin connector on the HC12 are connected to the pins on the HC12 shown below in the third column.

Transceiver Module pins	9-pin inline connector	HC12 Connection	Function
1	8,9	Vcc	Vcc
2	7	Ground	Ground
3	5	Port T pin 4	Receive Enable
4	4	Port T pin 6	LED Status Indicator
5	3	Port T pin 2	Receiver Data Out
6	2	Port T pin 7	Carrier Detect
7	6	Port T pin 0	Input to Transmitter
8	1	Port T pin 5	Transmit Enable

Table 6: Pin connections

16.2.2 Transceiver Circuit Diagram

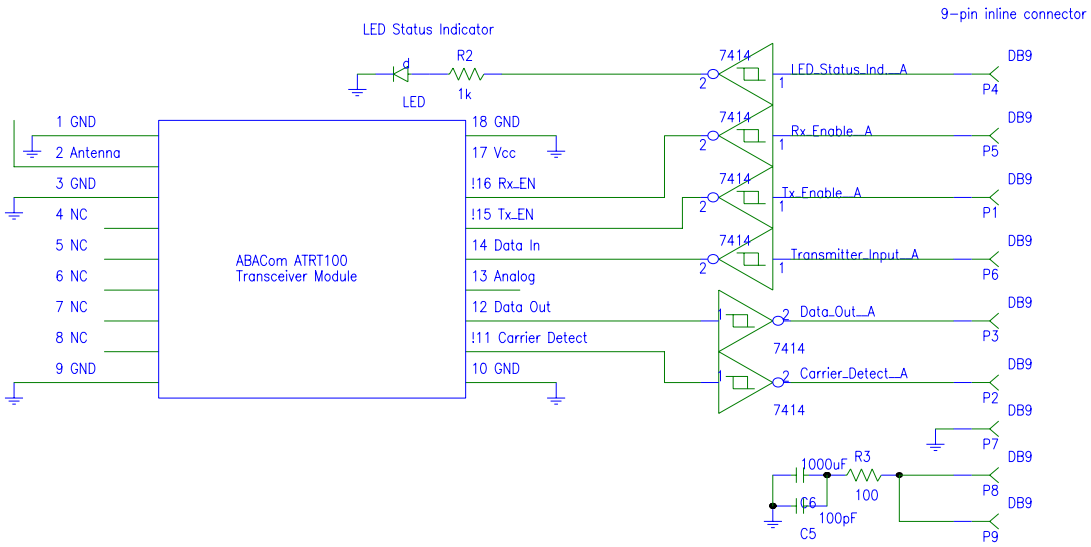


Table 7: Transceiver Circuit Diagram

16.3 Code Listing

16.3.1 Common

16.3.1.1 Definitions

16.3.1.1.1 types.h

```
typedef unsigned char   u8;
typedef signed char     s8;
typedef unsigned short  u16;
typedef signed short    s16;
typedef unsigned long   u32;
typedef signed long     s32;

typedef u8  AtomID;
```

16.3.1.1.2 params.h

```
const u8 send_buf_len = 16;
const u8 message_buf_len = 192;
const u8 packet_max_size = 64;
const u8 recv_buf_len = 96;
```

16.3.1.1.3 SendDef.hh

```
namespace AtomicPool {
      void add(AtomID);
      static inline bool queued(AtomID);
};

template <int ID> class Atomic {
public:
      inline void commit() const { AtomicPool::add(ID); }
      inline bool queued() const { return AtomicPool::queued(ID); }
      inline u8 index() const { return ID; }
};

template <int ID, class T> class Atom :
        public Atomic<ID> {
        T data;

public:
        inline void operator=(T val)
        {
            if (data == val)
                  return;

                data = val;
                commit();
        }

      inline void operator++(int)
      {
            data++;
            commit();
      }

        inline operator T() const
```

```
        {
                return data;
        }

      inline int special_inc()
      {
            data++;
            return ID;
      }
};
```

16.3.1.1.4 SendArrayHeader.hh

```
typedef bool (*diff_send)(void *, void *, u8);

struct SendData {
      diff_send   func;
      u8          pos;
      u8          size;
};

struct SendStruct send;

u8 send_buffer[send_buf_len];
u8 send_buf_pos;

#include "SendFunctions.hh"


#define SEND(func, entry)     { (diff_send)func, ((char *)&send.entry - (char
*)&send), sizeof(send.entry) }
#define ZERO      { 0, 0, 0}

const struct SendData send_func_array[] = {
```

16.3.1.1.5 SendArrayFooter.hh

```
};

#undef SEND
#undef ZERO

const u8 atomics = sizeof(send_func_array) / sizeof(SendData);
```

16.3.1.2 Protocol

16.3.1.2.1 atomicPool.hh

```
namespace AtomicPool {
      // Class prototype
      class AtomicLink;

      // Initialize the AtomicPool
      static inline void init();

      // Add an Atomic to the pool by marking it as queued
      void add(AtomID);

      // Return if an Atomic has been queued
```

```
	static inline bool queued(AtomID);

	// Queue the Atomic as if get() didn't return it
	static inline void queue(AtomID);

	// Dequeue the Atomic
	static inline void dequeue(AtomID);

	// Return the Atomic with the highest priority
	static inline AtomID get();
	// Returned by get() to represent that no Atomic are queued
	const AtomID nullID = 0xFF;

	// Set the Atomic to the lowest priority
	static inline void put(AtomID);
};

/* class AtomicLink
 *
 * This class provides an abstraction of an efficent doubly linked list with
a
 * fixed number of elements.  The order of the elements is defined by the
link
 * list.
 */
class AtomicPool::AtomicLink {
	// Pointers (or indexes here) for doubly linked list.
	u8	_next;
	u8	_prev;

	/* A boolean is used to make the operations on it atomic.
	 * the upper bit of the _next and _prev pointers could be used but this
	 * would require adding some mechanism (probably disabling interrupts)
	 * to ensure the operations on it are atomic.  But, this would cut the
	 * memory used by the AtomicLinks in half. */
	bool  _queued;

	/* This class should be a size that is a power of 2 to eliminate the
need
	 * for a multiply instruction when indexing an element in the array[]
*/
	u8    filler;

	// Static array that all AtomicLinks are part of.
	static class AtomicLink array[];
public:
	// Move this element before the parameter
	inline void move(AtomicLink *elem)
	{
		// Remove for current position
		array[_prev]._next = _next;
		array[_next]._prev = _prev;

		// Insert this above elem
		_prev = elem->_prev;
		elem->_prev = array[_prev]._next = this - array;
```

```
      _next = elem - array;
}

// Return pointer to next AtomicLink in list
inline AtomicLink * next()
{
      return &array[_next];
}

// Return the index of this AtomicLink
inline operator AtomID()
{
      return this - array;
}

// Manipulate the queued flag
inline bool queued() { return _queued; }
inline void set_queued() { _queued = true; }
inline void clr_queued() { _queued = false; }

// Return last AtomicLink in list.
// This AtomicLink doesn't represent an actual atomic.
// It is a place holder to establish a first and last AtomicLink.
static inline AtomicLink * entry()
{
      return &array[atomics];
}

// Return the AtomicLink corresponding the parameter index ID
static inline AtomicLink & index(AtomID ID)
{
      return array[ID];
}

// Initialize the link list and clear the flags
static inline void init()
{
      /* The link list must be initialized, giving each valid
       * AtomicLink some arbitrary position in the link list. */
      u8 last_valid = atomics;
      for (u8 i = 0; i < atomics; i++) {
            if (!send_func_array[i].func)
                  continue;
            array[i]._prev = last_valid;
            last_valid = i;
      }
      array[atomics]._prev = last_valid;

      u8 next_valid = atomics;
      for (u8 i = atomics - 1; i != 0xFF; i--) {
            if (!send_func_array[i].func)
                  continue;
            array[i]._next = next_valid;
            next_valid = i;
      }
      array[atomics]._next = next_valid;
```

```
	}
};

/* Initialize the static array of all the AtomicLinks.
 * Create one extra AtomicLink to be a place holder to establish a first and
 * last element in the list. */
class AtomicPool::AtomicLink AtomicPool::AtomicLink::array[atomics + 1];


static inline void AtomicPool::init()
{
	AtomicLink::init();
}

void AtomicPool::add(AtomID ID)
{
	AtomicLink &atom = AtomicLink::index(ID);

	if (!atom.queued()) {
		atom.set_queued();

		/* Signal that an atomic is ready to be sent.
		 * This is used by the link micro-controller to start sending
		 * data over the serial port if it is currently idle. */
		atom_data_ready();
	} else {
		/* If this atomic is already queued, increment the dropped data
		 * count.  We can not call add(AtomID) (this function) like we
		 * can everywhere else as it will cause an  infinite number of
		 * recursive calls if the dropped atomic is already queued. */
		AtomicLink::index(AtomicDropped.special_inc()).set_queued();
	}
}

static inline bool AtomicPool::queued(AtomID ID)
{
	return AtomicLink::index(ID).queued();
}

static inline void AtomicPool::queue(AtomID ID)
{
	AtomicLink::index(ID).set_queued();
}

static inline void AtomicPool::dequeue(AtomID ID)
{
	AtomicLink::index(ID).clr_queued();
}

/* Search through the AtomicLinks starting with the first in the list until
one
 * that is marked as queued is found.  If no AtomicLinks are marked as queued
 * then return nullID.
 *
 * dueue(AtomID) must be called right after the message to be sent has been
```

```
 * formed for the atomic returned by get().  This is not taken care of in
get()
 * to prevent sending the same data twice in the event that add(AtomID) is
 * called in an interrupt between the time get() returns and the data to be
sent
 * is copied from the send structure.
 *
 * This algorithum has a complexity of ohm(1) and O(n) where
 * n = (the number of atomics).
 * The best and worse case time are well defined but are substantially
different.
 * This algorithum therefore can take a substantially different amount of
time
 * to run depending on the circumstances.  This can be a source of problems
on
 * a micro-controller in which the processor time is fully utilized. */
static inline AtomID AtomicPool::get()
{
      AtomicLink *atom = AtomicLink::entry()->next();

      while (!atom->queued()) {
            atom = atom->next();
            if (atom == AtomicLink::entry())
                  return nullID;
      }

      return *atom;
}

static inline void AtomicPool::put(AtomID ID)
{
      AtomicLink::index(ID).move(AtomicLink::entry());
}
```

16.3.1.2.2 messageBuffer.hh

```
// Interface
namespace MessageBuffer {
      // Initialize the MessageBuffer
      static inline void init();

      // Compose a new message and place it in the send buffer
      static inline bool get();

      // Commit all the messages sense the last commit to the sent buffer.
      static inline void commit();

      // Requeue all the atomics and clear the message buffer.
      static inline void dump();
};

// Internal data
namespace MessageBuffer {
      // Message buffer to store sent data before we know if it was received.
      u8 message_buffer[message_buf_len];
      u8 message_buf_pos;
```

```
	// Sent buffer has the same structure as the send struct, containing
	// data known to have been received.
	u8 sent[sizeof(SendStruct)];
};

static inline void MessageBuffer::init()
{
	message_buf_pos = 0;

	/* Clear the sent buffer (all zeros is the intial known state) */
	for (u8 i = 0; i < sizeof(SendStruct); i++)
		sent[i] = 0;
}

/* Fill the send_buffer with a message and setup the message buffer to allow
 * committing the sent data to the sent buffer if it is confirmed to have
been
 * sent. */
static inline bool MessageBuffer::get()
{
	// Get the next atomic to send
	AtomID ID = AtomicPool::get();
	if (ID == AtomicPool::nullID)
		return false;

	// Clear the buffer (this buffer grows down)
	send_buf_pos = send_buf_len;

	// Fill message buffer
	message_buffer[message_buf_pos++] = ID;

	u8 size = send_func_array[ID].size;

	bool diff;

	if (size) {
		// Find the position of the atomic in the send buffer;
		u8 pos = send_func_array[ID].pos;

		void *dst = (void *)&message_buffer[message_buf_pos];
		void *src = &(((char *)&send)[pos]);

		/* The data that needs to be sent is copied into the message
		 * buffer with interrupts disabled.  This prevents an interrupt
		 * from writing the the atomic in the middle of the copy.
		 * The atomic is then dequeued.  Refer to atomicPool.hh for an
		 * explanation of why dequeue is called here */
		lock();
		memcpy(dst, src, size);
		AtomicPool::dequeue(ID);
		unlock();

		message_buf_pos += size;
```

```
            // Call the appropriate function to generate the data for this
atomic
            diff = send_func_array[ID].func(
                  dst,
                  (void *)&sent[pos],
                  size);
      } else {
            diff = false;
            lock();
            AtomicPool::dequeue(ID);
            unlock();
      }

      // Write header
      send_buffer[--send_buf_pos] = (ID << 1) | diff;

      return true;
}

static inline void MessageBuffer::commit()
{
      // Enumerate each message in the message buffer and commit it to the
      // sent buffer.
      for (u8 i = 0; i < message_buf_pos;) {
            AtomID ID = message_buffer[i++];

            AtomicPool::put(ID);

            u8 pos = send_func_array[ID].pos;
            u8 size = send_func_array[ID].size;

            memcpy(&sent[pos], &message_buffer[i], size);
            i += size;
      }

      // Clear message buffer
      message_buf_pos = 0;

      // i == message_buf_pos, if not BUG
}

static inline void MessageBuffer::dump()
{
      // Enumerate each message in the message buffer and requeue it to the
      // atomic pool to be sent again.
      for (u8 i = 0; i < message_buf_pos;) {
            AtomID ID = message_buffer[i++];

            AtomicPool::queue(ID);

            i += send_func_array[ID].size;
      }

      // Clear message buffer
      message_buf_pos = 0;
```

```
	// Increment the lost count
	::send.common.link.lost++;
}
```

16.3.1.2.3 packet.hh

```
// Interface
namespace Packet {
	// Initialize the Packet layer
	static inline void init();

	// Return true if more data is availible for the current packet
	static inline bool send_availible();

	// Return a byte to send for the current packet
	static inline u8 send_byte();

	// Passed byte received
	static inline bool recv_byte(u8);

	// Called if receive sequence is prematurely terminated
	static inline void recv_term();
};

// Internal Data
namespace Packet {
	// Defines if last packet sent has been confirmed to have been sent or
not
	bool last_confirmed;

	// Checksum identifiers packets sent and recieved
	u8 recv_last_packet;
	u8 send_this_packet;
	u8 send_last_packet;

	// Bytes sent in packet
	u8 send_length;

	// Running checksum of sent packet
	u8 send_checksum;

	// State of data availibility from send bottom half for interrupt
handler
	bool availible;
	bool pending;

	// Running checksum or received packet
	u8 recv_checksum;

	// Position within a message in a packet
	u8 recv_msg_pos;

	// Reset receive state to receive a new packet
	static inline void recv_end();

	// Bottom half handelers
```

```
	bool run_bh_send;
	static inline void bh_send();

	bool run_bh_recv;
	static inline void bh_recv();

	static inline void bottom_half_handler();
};

static inline void Packet::init()
{
	run_bh_send = false;
	run_bh_recv = false;
	last_confirmed = false;

	recv_last_packet = 0;
	send_this_packet = 0;
	send_last_packet = 0;
	send_length = 0;

	recv_end();
}

#define MSG_CHECKSUM    0xFF

/* The function constructs a packet to be sent by the send_byte function */
static inline void Packet::bh_send()
{
	if (last_confirmed) {
		if (send_length < packet_max_size &&
		    MessageBuffer::get()) {
			// Allow send_byte to send the data in send_buffer
			// created by MessageBuffer::get()
			pending = false;
			availible = true;

			send_length += send_buf_len - send_buf_pos;

			for (u8 i = send_buf_pos; i < send_buf_len; i++)
				send_checksum += send_buffer[i];

			return;
		}

		// Put a the checksum at the end of the packet
		send_buf_pos = send_buf_len - 2;
		send_buffer[send_buf_pos] = MSG_CHECKSUM;
		send_buffer[send_buf_pos+1] = send_checksum;

		pending = false;
		availible = false;

		if (send_length) {
			// Remeber this packet as the last sent
			send_last_packet = send_this_packet;
```

```
                if (send_last_packet == send_checksum)
                        send_this_packet = ~send_checksum;
                else
                        send_this_packet = send_checksum;
        }

        // Clear confirmed flag
        // Set if packet is confirmed to have been sent
        last_confirmed = false;

        // Reset to send a new packet
        send_length = 0;
        send_checksum = 0;

        // Update Status
        _io_ports[M6812_PORTT] &= ~(1 << LED_PORT);
        ::send.common.link.sent++;

        return;
    } else {
        // Send a dummy packet if the last packet wasn't confirmed to
        // have been sent or not.  We can't continue until we know the
        // state of the data at the other end.
        send_buf_pos = send_buf_len;
        send_buffer[--send_buf_pos] = recv_last_packet;
        send_buffer[--send_buf_pos] = MSG_CHECKSUM;

        pending = false;
        availible = false;

        send_length = 0;
        send_checksum = 0;

        // Update Status
        _io_ports[M6812_PORTT] &= ~(1 << LED_PORT);
        ::send.common.link.sent++;
        return;
    }
}

// Return if the send bottom half has more data to send
static inline bool Packet::send_availible()
{
    if (send_buf_pos == send_buf_len)
        return availible;

    return true;
}

static inline u8 Packet::send_byte()
{
    // If the bottom half didn't finish in time send a filler byte
    if (pending) {
        send_length++;
        return filler_message();
    }
```

```
	// If this is the last byte in the buffer, enable the bottom half
	// to generate more data to send
	if (send_buf_pos == send_buf_len - 1 && availible) {
		run_bh_send = true;
		pending = true;
	}

	return send_buffer[send_buf_pos++];
}


/* This recv_byte function determines if the recieved packet is valid without
 * the assistance of a bottom half.  This is likely the primary performace
bottle neck.
 * Changing this to be done outside the interrupt could substantially improve
the
 * maximum data rate.
 * The robot and link use distinctly different methods for handling recieved
data
 * for this reason the recieve buffer has been abstracted to allow the same
recieve
 * code for both. Like everything else hopefully the compiler can inline
this. */
static inline bool Packet::recv_byte(u8 byte)
{
	// Ignore first byte, the last recieved packet identifier
	// used by bottom half
	if (RecvBuffer::put_size()) {
		if (!recv_msg_pos) {    // If this is the start of a message
			static bool in_checksum;

			if (in_checksum) {
				// Check checksum
				if (recv_checksum == byte) {
					if (RecvBuffer::put_size() != 1) {
						// If this is not a dumby packet set it
as
						// the last recieved packet
						if (recv_last_packet == recv_checksum)
							recv_last_packet = ~recv_checksum;
						else
							recv_last_packet = recv_checksum;
					}

					// Run the bottom half handler
					run_bh_recv = true;

					// Commit the received data.  Only used on the
					// link end to start the data pass through
					RecvBuffer::put_commit();

					recv_end();
				} else {
					::send.common.link.BadChecksum++;
					recv_term();
```

```
                        }

                        in_checksum = false;
                        return false;
                  }
                  if (byte == MSG_CHECKSUM) {
                        in_checksum = true;
                        return true;
                  }

                  // Check if this is a valid message identifier
                  if (byte >= MessageBuffer::messages()) {
                        ::send.common.link.Malformed++;
                        recv_term();
                        return false;
                  }

                  recv_msg_pos = MessageBuffer::msg_size(byte);

                  // Check if this will remain withing the packet size limit
                  if (RecvBuffer::put_size() + recv_msg_pos >= recv_buf_len -
2) {
                        ::send.common.link.Malformed++;
                        recv_term();
                        return false;
                  }
            } else
                  recv_msg_pos--;
      }

      // Handle a buffer overflow (This has never been known to happen)
      if (RecvBuffer::put_can()) {
            recv_checksum += byte;
            RecvBuffer::put(byte);
      } else {
            ::send.common.link.BufferOverflow++;
            recv_term();
            return false;
      }

      return true;
}

static inline void Packet::recv_end()
{
      // Set the first byte of the next packet to be sent,
      // the last packet received identifier.
      send_buf_pos = send_buf_len - 1;
      send_buffer[send_buf_pos] = recv_last_packet;
      send_checksum = recv_last_packet;

      pending = false;
      availible = true;

      // Clear vars to receive next packet
      recv_checksum = 0;
```

```
	recv_msg_pos = 0;
}

static inline void Packet::recv_term()
{
	// Mark data just received as bad
	RecvBuffer::put_dump();
	recv_end();
}

static inline void Packet::bh_recv()
{
	u8 header = RecvBuffer::get_first();

	// Inform Message Buffer layer of success or failure of last
transmition
	if (header == send_this_packet) {
		last_confirmed = true;
		MessageBuffer::commit();
	} else if (header == send_last_packet) {
		send_this_packet = send_last_packet;
		last_confirmed = true;
		MessageBuffer::dump();
	}

	MessageBuffer::put_good();
}


static inline void Packet::bottom_half_handler()
{
	if (run_bh_recv) {
		run_bh_recv = false;
		bh_recv();
	}
	if (run_bh_send) {
		run_bh_send = false;
		bh_send();
	}
}
```

16.3.1.2.4 transciever.hh

```
#define SYNC_BYTE 0xFF

/* Data I/O port assignments on port T */
#define IN_PORT         0
#define OUT_PORT  2
#define RX_PORT         4
#define TX_PORT         5
#define CD_PORT         7

/* Timer Prescaler 2^val <= 32 */
#define TIMER_PRESCALE  0;
```

```
/* Number of bits to send per period 2^val <= 8 */
#define BIT_EXPONENT    1

/* Minimum time of period */
#define MIN_PERIOD      640   // Works between (1024 and 320(bit flaky))

/* Time added val*2^val */
#define PERIOD_EXPONENT 6

/* Time to wait before transmiting */
#define SEND_DELAY      50    // 2 works

#define TIMEOUT_PERIOD  0xFFFF



#define BIT_MASK  (0xFF >> (8 - (1 << BIT_EXPONENT)))

#define _tc_regs  ((volatile unsigned short *)&_io_ports[M6812_TC0])


namespace Tranciever {
      using Packet::send_availible;
      using Packet::send_byte;
      using Packet::recv_byte;
      using Packet::recv_term;


      static inline void init();

      static inline void set_output();
      static inline void clr_output();

      static inline void send_mode();
      static inline void recv_mode();

      void __attribute__((interrupt)) send_handler(void);
      void __attribute__((interrupt)) pre_handler(void);
      void __attribute__((interrupt)) recv_handler(void);
      void __attribute__((interrupt)) timeout_handler(void);

      unsigned char pre_state;
      unsigned char send_state;
      unsigned char send_buffer;
      unsigned char recv_state;
      unsigned char recv_buffer;

      unsigned char usin;
};

static inline void Tranciever::init()
{
      // Enable Timer System
      _io_ports[M6812_TSCR] = 0x80;

      // Set Timer prescaler
```

```
      // and (optional) Enable timer overflow interupt
      _io_ports[M6812_TMSK2] = TIMER_PRESCALE;  //0x80 | TIMER_PRESCALE;

      // For utilization part
      set_timer_counter(0);
      _io_ports[M6812_TFLG2] = 0x80;

      // Set ports to input capture or output compare
      // [ Each bit corresponds to a T port ]
      // [ 0 = input capture, 1 = output compare ]
      _io_ports[M6812_TIOS] = 1 << OUT_PORT;

      // Set input action
#if IN_PORT > 3
      _io_ports[M6812_TCTL3] = 0x03 << ((IN_PORT - 4) << 1);
#else
      _io_ports[M6812_TCTL4] = 0x03 << (IN_PORT << 1);
#endif

      // Setup port T
      _io_ports[M6812_DDRT] =
            (1 << LED_PORT) |
            (1 << RX_PORT) |
            (1 << TX_PORT);

      recv_state = 0;
      recv_buffer = ~SYNC_BYTE;

        //setup interrupt vector
        *((int *)(0x0B2E - OUT_PORT*2)) = (int)pre_handler;
        *((int *)(0x0B2E - IN_PORT*2)) = (int)recv_handler;

      recv_mode();
}

static inline void Tranciever::set_output()
{
      // Set output action on output compare
#if OUT_PORT > 3
            _io_ports[M6812_TCTL1] = 0x01 << ((OUT_PORT - 4) << 1);
#else
            _io_ports[M6812_TCTL2] = 0x01 << (OUT_PORT << 1);
#endif
}

static inline void Tranciever::clr_output()
{
      // Set output action on output compare
#if OUT_PORT > 3
            _io_ports[M6812_TCTL1] = 0x02 << ((OUT_PORT - 4) << 1);
#else
            _io_ports[M6812_TCTL2] = 0x02 << (OUT_PORT << 1);
#endif
}

static inline void Tranciever::send_mode()
```

```
{
	set_output();

	// Set Tranciever to transmit mode
	//_io_ports[M6812_PORTP] = 0x80;
	_io_ports[M6812_PORTT] = (1 << TX_PORT) |
		(_io_ports[M6812_PORTT] & (1 << LED_PORT));

	// Set interrupt to run after SEND_DELAY time
	//_tc_regs[OUT_PORT] =_io_ports[M6812_TCNT] + SEND_DELAY;
	_tc_regs[OUT_PORT] =_io_ports[M6812_TCNT] + MIN_PERIOD;
	pre_state = SEND_DELAY;

	send_buffer = SYNC_BYTE;
	send_state = 8 / (1 << BIT_EXPONENT);

	  *((int *)(0x0B2E - OUT_PORT*2)) = (int)pre_handler;

	// Clear Interrupt flag
	_io_ports[M6812_TFLG1] = 1 << OUT_PORT;

	// Set only the send interrupt
	_io_ports[M6812_TMSK1] = 1 << OUT_PORT;
}

static inline void Tranciever::recv_mode()
{
	clr_output();

	*((int *)(0x0B2E - OUT_PORT*2)) = (int)timeout_handler;

	_tc_regs[OUT_PORT] =_io_ports[M6812_TCNT] + TIMEOUT_PERIOD;

	usin = 5;   // 1 Works at best

	// Set Tranciever to recieve mode
	//_io_ports[M6812_PORTP] = 0x40;
	_io_ports[M6812_PORTT] = (1 << RX_PORT) |
		(_io_ports[M6812_PORTT] & (1 << LED_PORT));

	// Clear Interrupt flag
	_io_ports[M6812_TFLG1] = 1 << IN_PORT;
	_io_ports[M6812_TFLG1] = 1 << OUT_PORT;

	// Set only the recv interrupt
	_io_ports[M6812_TMSK1] = (1 << IN_PORT) |
		(1 << OUT_PORT);
}

void Tranciever::timeout_handler(void)
{
	if (!usin) {
#ifdef MASTER_TRANCIEVER
		::send.timeout++;
		recv_term();
		send_mode();
```

```
#else
            _io_ports[M6812_PORTT] |= (1 << LED_PORT);
#endif
      }

      usin--;

      _io_ports[M6812_TFLG1] = 1 << OUT_PORT;
}

void Tranciever::pre_handler(void)
{
      if (!pre_state)
            *((int *)(0x0B2E - OUT_PORT*2)) = (int)send_handler;
      else
            pre_state--;

      _tc_regs[OUT_PORT] += MIN_PERIOD;

      _io_ports[M6812_TFLG1] = 1 << OUT_PORT;
}

void Tranciever::send_handler(void)
{
      if (!send_state) {
            if (send_availible()) {
                  send_buffer = send_byte();
                  send_state = 8 / (1 << BIT_EXPONENT) - 1;
            } else {
                  recv_mode();
            }
      } else
            send_state--;

      /* Set next edge time */
      _tc_regs[OUT_PORT] +=
            MIN_PERIOD + ((send_buffer & BIT_MASK) << PERIOD_EXPONENT);

      /* Shift to next set of bits */
      send_buffer >>= (1 << BIT_EXPONENT);

      _io_ports[M6812_TFLG1] = 1 << OUT_PORT;
}


#define BIT_EXP_COMP    (8 - (1 << BIT_EXPONENT))

void Tranciever::recv_handler(void)
{
      static unsigned short   last;

      unsigned short diff = _tc_regs[IN_PORT] - last;
      last = _tc_regs[IN_PORT];

      // Transform diff from a half cycle period to an integer representing
      // the actual data sent in that period at the upper bits of a byte.
```

```
	// This bizzar method of using the upper bits improves the performance
	// of this function for the particular settings that work with the
	// tranciever.
	diff -= MIN_PERIOD - (1 << (PERIOD_EXPONENT - 1));
	diff >>= PERIOD_EXPONENT - BIT_EXP_COMP;

	if (diff >= (1 << (BIT_EXPONENT + 1)) << BIT_EXP_COMP) {
		// diff is outside data range, terminate receive
		if (recv_state)   {
			recv_term();
			send_mode();
			recv_state = 0;
		}
		recv_buffer = 0;
	} else {
		// Store byte segment into byte receive buffer
		recv_buffer |= diff & (0xFF << BIT_EXP_COMP);

		if (recv_state == 8 / (1 << BIT_EXPONENT)) {
			if (recv_byte(recv_buffer))
				recv_state = 1;
			else {
				// All data received, switch to send mode
				send_mode();
				recv_state = 0;
			}
			recv_buffer = 0;
		} else {
			if (recv_state) {
				recv_state++;
			} else if (recv_buffer == SYNC_BYTE)
				// Sync received, now accept data
				recv_state = 1;

			// Shift byte buffer to accept next byte segment
			recv_buffer >>= 1 << BIT_EXPONENT;
		}
	}

	_io_ports[M6812_TFLG1] = 1 << IN_PORT;
}
```

16.3.1.2.5 sendFunctions.hh

```
#define add(val)	{ send_buf_pos -= sizeof(val); \
			*(__typeof__(val) *)(&send_buffer[send_buf_pos]) = val;  }

#define sOF(val)  (val < -128 || val > 127)
#define uOF(val)  (val > 255)

bool send_null(void *, void *)
{
	return false;
}

bool send_c8(u8 *newData, u8 *oldData, u8)
```

```
{
	add(*newData);
	return false;
}

bool send_s16(s16 *newData, s16 *oldData, u8)
{
	s16 diff = *newData - *oldData;

	if (sOF(diff)) {
		add(*newData);
		return false;
	} else {
		add((s8)diff);
		return true;
	}
}

bool send_s32(s32 *newData, s32 *oldData, u8)
{
	s32 diff = *newData - *oldData;

	if (sOF(diff)) {
		add(*newData);
		return false;
	} else {
		add((s8)diff);
		return true;
	}
}


bool send_u16(u16 *newData, u16 *oldData, u8)
{
	u16 diff = *newData - *oldData;

	if (uOF(diff)) {
		add(*newData);
		return false;
	} else {
		add((u8)diff);
		return true;
	}
}

bool send_u32(u32 *newData, u32 *oldData, u8)
{
	u32 diff = *newData - *oldData;

	if (uOF(diff)) {
		add(*newData);
		return false;
	} else {
		add((u8)diff);
		return true;
	}
```

```
}

bool send_GPS(GPSPosition *newData, GPSPosition *oldData)
{
      // Difference feature unimplemented
      add(*newData);
      return false;
}

bool send_GPSMessage(GPSMessage *newData, GPSMessage *oldData)
{
      add(*newData);
      return false;
}

#undef add
#undef sOF
#undef uOF
```

16.3.1.3 Atomic Structures

16.3.1.3.1 CommonStruct.hh

```
/* Every Atomic here must be reflected in CommonSendArray */

struct GPSPosition {
      s32   latitude;
      s32   longitude;
};

struct GPSMessage {
      u8                sat_qual;
      struct GPSPosition      cordinate;
      s32               height;
      u16               dilution;
};

struct LinkStatusStruct {
      Atom<0, u32>      AtomicDropped;
      Atom<1, u32>      lost;
      Atom<2, u32>      sent;
      Atom<3, u32>      BufferOverflow;

      Atom<4, u32>      BadChecksum;
      Atom<5, u32>      Malformed;
};

struct CommonStruct {
      struct LinkStatusStruct link;
};
```

16.3.1.3.2 CommonSendArray.hh

```
      SEND(send_u32, common.link.AtomicDropped),
      SEND(send_u32, common.link.lost),
      SEND(send_u32, common.link.sent),
      SEND(send_u32, common.link.BufferOverflow),
      SEND(send_u32, common.link.BadChecksum),
```

```
      SEND(send_u32, common.link.Malformed),
            ZERO,
```

16.3.1.3.3 RobotSendStruct.hh

```
/* Every Atomic here must be reflected in RobotSendArray.hh */

struct GPSStruct :
      public Atomic<8>,
      public GPSMessage {};

struct SendStruct {
      struct CommonStruct     common;

      Atomic<7>   reset;

      struct GPSStruct  GPS;

      Atom<9, u8> pot1;
      Atom<10, u32>     badMessages;

      Atomic<11>  filler;

      // Integration Data
      Atom<16, u16>     heading;
      Atom<37, u8>      ball_hole;
};

#define LAST_LOCAL_ID   11
```

16.3.1.3.4 RobotSendArray.hh

```
      SEND(send_null, reset),
      SEND(send_GPSMessage, GPS),
      SEND(send_c8, pot1),
      SEND(send_u32, badMessages),
      SEND(send_null, filler),
            ZERO,
            ZERO,
            ZERO,
            ZERO,
      SEND(send_s16, heading),
```

16.3.1.3.5 RecvStruct.hh

```
struct RecvStruct {
      struct GPSPosition      ball;
      struct GPSPosition      hole;
};
```

16.3.1.4 Makefile

```
objects := $(patsubst %.cc,%.o,$(wildcard *.cc))

hc12.s19 : hc12.elf
      m6812-elf-objcopy --output-target=srec --only-section=.text --only-
section=.rodata --only-section=.vectors $< $@

hc12.elf : $(objects)
```

```
	m6812-elf-gcc -m68hc12 -mshort -o $@ $(objects) ./../lib/libutil.a
./../config/m68hc12-eval/memory.x

%.o : %.cc %.d
	m6812-elf-g++ -m68hc12 -mshort -ffixed-z -fomit-frame-pointer -msoft-
reg-count=0 -Wall -Os -I./../include -I../common -I. -c $< -o $@

%.d : %.cc
	set -e; $(CPP) -I./../include -I../common -I. -MM $< \
		  | sed -e 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
		  [ -s $@ ] || rm -f $@

include $(objects:.o=.d)

clean::
	rm -f *.d
	rm -f *.elf
	rm -f *.s19
	rm -f *.o
```

16.3.2 Robot

16.3.2.1 main.cc

```
#include <string.h>

#include <sys/locks.h>
#include <sys/ports_def.h>
#include <sys/ports.h>

#include "types.h"
#include "params.hh"

#include "SendDef.hh"
#include "CommonStruct.hh"
#include "RobotSendStruct.hh"

//
#include "SendArrayHeader.hh"
#include "CommonSendArray.hh"
#include "RobotSendArray.hh"
#include "SendArrayFooter.hh"

struct SmallCordinate {
	short ball_longitude;
	short	ball_latitude;
	short hole_longitude;
	short	hole_latitude;
} smallCord;
bool ballAvail, holeAvail;

static inline void atom_data_ready(void) {}

typeof(::send.common.link.AtomicDropped) &AtomicDropped =
::send.common.link.AtomicDropped;
static inline u8 filler_message()
{
```

```
	return send.filler.index();
}

#define LED_PORT  6

#include "atomicPool.hh"
#include "messageBuffer.hh"

#include "spiInterface.hh"

#include "RecvStruct.hh"
#include "Recv.hh"
#include "messageRecv.hh"

#include "packet.hh"

#include "tranciever.hh"

#include "gpsInterface.hh"

int main()
{
	AtomicPool::init();

	RecvBuffer::init();

	MessageBuffer::init();
	Packet::init();

	for (u8 i = 0; i < sizeof(send); i++)
		((char *)&send)[i] = 0;

	_io_ports[M6812_ATD0CTL2] = 0x80;
	_io_ports[M6812_ATD0CTL4] = 0x81;
	_io_ports[M6812_ATD0CTL5] = 0x63;

	ballAvail = false;
	holeAvail = false;

	Tranciever::init();
	Interface::init();
	SerialPort::init();

	while (1) {
		for (unsigned int i = 0; i < 0x00FF; i++) {
			Packet::bottom_half_handler();
			Interface::bottom();
			SerialPort::bottom_half();
		}

		send.pot1 = (
			_io_ports[M6812_ADR00H] +
			_io_ports[M6812_ADR01H] +
			_io_ports[M6812_ADR02H] +
			_io_ports[M6812_ADR03H] +
			_io_ports[M6812_ADR04H] +
```

```
                  _io_ports[M6812_ADR05H] +
                  _io_ports[M6812_ADR06H] +
                  _io_ports[M6812_ADR07H]) >> 3;

      }
}
```

16.3.2.2 Definitions

16.3.2.2.1 Recv.hh

```
typedef void (*diff_recv)(void *, void *, u8);

struct RecvData {
      diff_recv   func;
      u8          pos;
      u8          size;
};

struct RecvStruct recv;

#include "RecvFunctions.hh"


#define RECV(func, entry)     { (diff_recv)func, ((char *)&recv.entry - (char
*)&recv), sizeof(recv.entry) }

const struct RecvData recv_func_array[] = {
#include "RecvArray.hh"
};

#undef RECV

const u8 recv_funcs = sizeof(recv_func_array) / sizeof(RecvData);
```

16.3.2.2.2 RecvArray.hh

```
      { (diff_recv)recv_filler, 0, 0 },
      { (diff_recv)recv_start, 0, 0 },
      { (diff_recv)recv_stop, 0, 0 },
      RECV(recv_ball, ball),
      RECV(recv_hole, hole),
```

16.3.2.3 Imperative

16.3.2.3.1 messageRecv.hh

```
namespace RecvBuffer {
      u8 buffer[recv_buf_len];
      u8 pos;

      static inline void init();

      static inline bool put_can();
      static inline void put(u8);
      static inline u8 put_size();
      static inline void put_commit();
      static inline void put_dump();

      static inline bool get_avail(u8);
```

```
	static inline u8 get_first();
	static inline u8 *get(u8);
};

static inline void RecvBuffer::init()
{
	pos = 0;
}

static inline bool RecvBuffer::put_can()
{
	return pos < recv_buf_len;
}

static inline void RecvBuffer::put(u8 val)
{
	buffer[pos++] = val;
}

static inline u8 RecvBuffer::put_size()
{
	return pos;
}

static inline void RecvBuffer::put_commit()
{
}

static inline void RecvBuffer::put_dump()
{
	pos = 0;
}

static inline bool RecvBuffer::get_avail(u8 val)
{
	return val < pos;
}

static inline u8 RecvBuffer::get_first()
{
	return buffer[0];
}

static inline u8 * RecvBuffer::get(u8 val)
{
	return &buffer[val];
}



namespace MessageBuffer {
	static inline void put_good();

	static inline u8 messages();
	static inline u8 msg_size(AtomID);
}
```

```
static inline void MessageBuffer::put_good()
{
	// For each message received execute the appropriate function
	for (u8 i = 1; RecvBuffer::get_avail(i);) {
		AtomID ID = *RecvBuffer::get(i++);

		bool diff = ID & 0x01;
		ID >>= 1;

		u8 pos = recv_func_array[ID].pos;
		u8 size = diff ? 1 : recv_func_array[ID].size;

		recv_func_array[ID].func(
			(void *)RecvBuffer::get(i),
			(void *)((char *)&recv + pos),
			size);

		i += size;
	}

	// Clear buffer
	RecvBuffer::put_dump();
}

static inline u8 MessageBuffer::messages()
{
	return recv_funcs * 2;
}

static inline u8 MessageBuffer::msg_size(AtomID ID)
{
	if (ID & 1)
		return 1;
	else
		return recv_func_array[ID >> 1].size;
}
```

16.3.2.3.2 RecvFunctions.hh

```
void recv_filler(void *, void *, u8)
{

}

void recv_start(void *, void *, u8)
{
	// Start Command
}

void recv_stop(void *, void *, u8)
{
	// Stop Command

}
```

```
void recv_generic(u8 *srcData, u8 *dstData, u8 size)
{
      for (u8 pos = 0; pos < size; pos++)
            dstData[pos] = srcData[pos];
}

void recv_ball(GPSPosition *srcData, u8 *dstData, u8 size)
{
      smallCord.ball_longitude = srcData->longitude;
      smallCord.ball_latitude = srcData->latitude;
      ballAvail = true;
}

void recv_hole(GPSPosition *srcData, u8 *dstData, u8 size)
{
      smallCord.hole_longitude = srcData->longitude;
      smallCord.hole_latitude = srcData->latitude;
      holeAvail = true;
}
```

16.3.2.4 Interface

16.3.2.4.1 gpsInterface.hh

```
#define BUFFER_LEN 82

#include <sys/sio.h>

namespace SerialPort {
      void __attribute__((interrupt)) sci_handler(void);

      static inline void init();

      static inline void bottom_half();

      static inline bool parse();

      char buffer[BUFFER_LEN];
      char pos;

      bool run_bottom;
};

void __attribute__((interrupt)) SerialPort::sci_handler(void)
{
      char state = _io_ports[M6812_SC0SR1];

      if (state & M6812_IDLE) {
            pos = 0;
            run_bottom = true;
      }

      if (state & M6812_RDRF) {
            if (pos == BUFFER_LEN - 1) {
//                send.common.link.BufferOverflow++;
                  pos = 0;
            }
```

```
            buffer[pos++] = serial_recv();
      } else
            serial_recv();
}

static inline void SerialPort::init()
{
      serial_baud(4800);
      serial_init();

      run_bottom = false;
      pos = 0;

      *((int *)0x0B16) = (int)sci_handler;

//    _io_ports[M6812_HPRIO] = 0x16;

      serial_set_rx_int();
      serial_set_idle_int();
}

static inline void SerialPort::bottom_half()
{
      if (run_bottom) {
            //parse();
            if (!parse())
                  send.common.link.BufferOverflow++;
            run_bottom = false;
      }
}

const char header[] = "$GPGGA,";

const long decimal_array[][16] = {
      { 0, 60000000, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
      { 0, 6000000, 12000000, 18000000, 24000000, 30000000, 36000000,
42000000, 48000000, 54000000, 0, 0, 0, 0, 0, 0 },
      { 0, 600000, 1200000, 1800000, 2400000, 3000000, 3600000, 4200000,
4800000, 5400000, 0, 0, 0, 0, 0, 0  },
      { 0, 100000, 200000, 300000, 400000, 500000, 600000, 700000, 800000,
900000, 0, 0, 0, 0, 0, 0  },
      { 0, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 0,
0, 0, 0, 0, 0, 0  },
      { 0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 0, 0, 0, 0,
0, 0  },
      { 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 0, 0, 0, 0, 0, 0  },
      { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 0, 0, 0, 0, 0, 0  },
      { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 0, 0, 0, 0, 0  }
};

const char short_decimal_array[][16] = {
      { 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 0, 0, 0 ,0 ,0, 0 },
      { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 0, 0, 0 ,0 ,0, 0 }
};
```

```
const char small_decimal_array[16] =
      { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 }
;

#define COMMA_CHECK     if (buffer[pos++] != ',')     return false

static inline bool SerialPort::parse()
{
//$GPGGA,005108,3403.9857,N,10654.4168,W,1,05,3.5,1489.1,M,-23.7,M,,*4B
//    static const char buffer[] =
"$GPGGA,012417,3403.9785,N,10654.4511,W,1,05,2.5,1419.9,M,-23.7,M,,*4C";
      unsigned char pos = 0;

//    while (buffer[pos] != '$' && pos < (BUFFER_LEN - 70))
//          pos++;

//    pos++;

      // Check header
      for(unsigned char i = 0; header[i]; i++) {
            if (buffer[pos++] != header[i])
                  return false;
      }

      // Skip time
      pos += 6;
      COMMA_CHECK;
//    return true;

      // LATITUDE
      long &latitude = send.GPS.cordinate.latitude;
      latitude = 0;

      // Get first part of Latitude
      for(unsigned char i = 1; i < 5; i++, pos++) {
            char byte = buffer[pos];
            if (byte < '0' || byte > '9')
                  return false;
            latitude += decimal_array[i][byte - '0'];
      }

      // Check for period
      if (buffer[pos++] != '.')
            return false;

      // Get last part of Latitude
      for(unsigned char i = 5; i < 9; i++, pos++) {
            char byte = buffer[pos];
            if (byte < '0' || byte > '9')
                  return false;
            latitude += decimal_array[i][byte - '0'];
      }

/*    send.GPS.commit();
      return true;*/
```

```
COMMA_CHECK;

if (buffer[pos] == 'S')
      latitude = -latitude;
else if (buffer[pos] != 'N')
      return false;
pos++;

COMMA_CHECK;

//LONGITUDE
long &longitude = send.GPS.cordinate.longitude;
longitude = 0;

// Get first part of Longitude
for(unsigned char i = 0; i < 5; i++, pos++) {
      char byte = buffer[pos];
      if (byte < '0' || byte > '9')
            return false;
      longitude += decimal_array[i][byte - '0'];
}

// Check for period
if (buffer[pos++] != '.')
      return false;

// Get last part of Longitude
for(unsigned char i = 5; i < 9; i++, pos++) {
      char byte = buffer[pos];
      if (byte < '0' || byte > '9')
            return false;
      longitude += decimal_array[i][byte - '0'];
}

COMMA_CHECK;

if (buffer[pos] == 'W')
      longitude = -longitude;
else if (buffer[pos] != 'E')
      return false;
pos++;

COMMA_CHECK;

unsigned char &sat_qual = send.GPS.sat_qual;

sat_qual = (buffer[pos++] - '0') << 4;
if (sat_qual > (9 << 4))
      return false;

COMMA_CHECK;
{
      char byte = buffer[pos++];
      if (byte < '0' || byte > '1')
            return false;
      sat_qual += small_decimal_array[byte - '0'];
```

```
	}
	if (buffer[pos] < '0' || buffer[pos] > '9')
		return false;
	sat_qual += buffer[pos++] - '0';

	COMMA_CHECK;

	unsigned short &dilution = send.GPS.dilution;

	if (buffer[pos] != ',') {
		char first = buffer[pos++] - '0';
		if (first > 9)
			return false;
		if (buffer[pos] == '.')
			dilution = short_decimal_array[1][first];
		else {
			dilution = short_decimal_array[0][first];
			if (buffer[pos] < '0' || buffer[pos] > '9')
				return false;
			dilution += short_decimal_array[1][buffer[pos] - '0'];
		}
		pos++;
		if (buffer[pos] < '0' || buffer[pos] > '9')
			return false;
		dilution += buffer[pos++] - '0';
	} else
		dilution = 0;

	COMMA_CHECK;

	long &height = send.GPS.height;
	height = 0;

	if (buffer[pos] != ',') {
		// Get first part of Height
		for(unsigned char i = 4; i < 8; i++, pos++) {
			char byte = buffer[pos];
			if (byte < '0' || byte > '9')
				return false;
			height += decimal_array[i][byte - '0'];
		}

		// Check for period
		if (buffer[pos++] != '.')
			return false;

		height += buffer[pos++] - '0';
	}

	send.GPS.commit();
	return true;
}
```

16.3.2.4.2 spiInterface.hh

```
/* interface */
```

```
#define DATA_AVAIL (_io_ports[M6812_SP0SR] & 0x80)
#define GARBAGE 0
#define DELAY  1000

#define RECV_ID   0x55

namespace Interface {
      static inline void init();
      static inline void bottom();
      static inline void send_to_master_poll();

      bool tx;
      char data;
      unsigned char wstate;
      unsigned int clock_count;

      unsigned char msg_pos;
      unsigned char msg_size;
      unsigned char msg_ID;
};

static inline void Interface::init()
{
      wstate = 0;
      tx = false;

      _io_ports[M6812_DDRP] = 0xFF;
      _io_ports[M6812_DDRS] |= 0x10;
      _io_ports[M6812_SP0CR1] = 0x4C;
      _io_ports[M6812_SP0CR2] = 0x00;

      _io_ports[M6812_PORTP] = 0x10;

      msg_size = 0;
}

static inline void Interface::bottom()
{
      static char count;
      if (!tx) {
            if (DATA_AVAIL) {
                  char byte;
                  if ((byte = _io_ports[M6812_SP0DR]) == RECV_ID) {
                        msg_pos = 0;
                        wstate = 0;
                        tx = true;
                  } else {
                        _io_ports[M6812_PORTP] = (count++ % 10) << 4;

                        if (msg_size) {
                              msg_size--;
                              ((char *)&send)[msg_pos++] = byte;
                        } else {
                              msg_ID = byte;
```

```
                              if (msg_ID <= LAST_LOCAL_ID ||
                                  msg_ID >= atomics ||
                                  !send_func_array[msg_ID].func) {
                                    send.badMessages++;
                                    return;
                              }

                              msg_size = send_func_array[msg_ID].size;
                              msg_pos = send_func_array[msg_ID].pos;
                        }

                        if (!msg_size)
                              AtomicPool::add(msg_ID);
                  }
            }
      } else {
            if (!ballAvail || !holeAvail)
                  return;

            _io_ports[M6812_PORTP] = 0x00;
            send_to_master_poll();
            if (msg_pos == sizeof(SmallCordinate))
                  tx = false;
            }

}

static inline void Interface::send_to_master_poll()
{
      if (wstate == 0) {
            _io_ports[M6812_SP0DR] = ((char *)&smallCord)[msg_pos++];
            _io_ports[M6812_PORTP] = 0x01;
            wstate = 1;
            clock_count = DELAY;

      } else if (wstate == 1) {

            if (!--clock_count)
                  wstate = 2;
      } else if (wstate == 2) {

            if (DATA_AVAIL)
                  wstate = 3;
      } else if (wstate == 3) {
            wstate = 0;
      }
}
```

16.3.3 Link

16.3.3.1 main.cc

```
#include <string.h>

#define MASTER_TRANCIEVER

#include <sys/locks.h>
```

```
#include <sys/ports_def.h>
#include <sys/ports.h>

#include "types.h"
#include "params.hh"

#include "SendDef.hh"
#include "CommonStruct.hh"
#include "LinkSendStruct.hh"

// Array of data to send to computer, inlined to get size of array
#include "SendArrayHeader.hh"
#include "CommonSendArray.hh"
#include "LinkSendArray.hh"
#include "SendArrayFooter.hh"

static inline void atom_data_ready(void);
static inline void message_data_ready(void);

typeof(send.relay.AtomicDropped) &AtomicDropped = send.relay.AtomicDropped;

#include "atomicPool.hh"

// Copy of data robot sends, used to verify data received
namespace RobotRecv {
      #include "RobotSendStruct.hh"

      struct SendData {
            u8    size;
      };
      struct SendStruct send;

      #define SEND(func, entry)     { sizeof(send.entry) }
      #define ZERO                  { 0xFF }

      const struct SendData send_func_array[] = {
      #include "CommonSendArray.hh"
      #include "RobotSendArray.hh"
      #include "SendArrayFooter.hh"
};

#define LED_PORT  6

// Some code is duplicated almost verbatim.  We have the memory to waste, and
making
// this object oriented would incur a substantial performance overhead.
namespace Transport {
/* This "SendDef.hh" will refer to a different AtomicPool than the previous
*/
#include "SendDef.hh"
#include "CommonStruct.hh"
#include "LinkRecvStruct.hh"

//
#include "SendArrayHeader.hh"
```

```
#include "LinkRecvArray.hh"
#include "SendArrayFooter.hh"

typeof(::send.common.link.AtomicDropped) &AtomicDropped =
::send.common.link.AtomicDropped;
static inline u8 filler_message()
{
      ::send.filler++;
      return 0;
}

#include "atomicPool.hh"
#include "messageBuffer.hh"
#include "messageRecv.hh"
#include "packet.hh"

#include "receive.hh"

#include "tranciever.hh"
};

#include "send.hh"
#include "serial.hh"

static inline void bottom_half()
{
        Transport::Packet::bottom_half_handler();
        Send::bottom_half_handler();
}

int main()
{
        AtomicPool::init();

      Transport::AtomicPool::init();
      Transport::MessageBuffer::init();
      Transport::Packet::init();
      Transport::RecvBuffer::init();
      Transport::Receive::init();

      for (u8 i = 0; i < sizeof(send); i++)
            ((char *)&send)[i] = 0;

      Send::init();
      SerialPort::init();

      Transport::Tranciever::init();

      while (1)
            bottom_half();
}
```

16.3.3.2 Definitions

16.3.3.2.1 Recv.hh

```
struct RecvData {
```

```
      u8          pos;
      u8          size;
};

struct RecvStruct recv;

#define RECV(entry)     { ((char *)&recv.entry - (char *)&recv),
sizeof(recv.entry) }

#include "RecvFunctions.hh"

const u8 recv_funcs = sizeof(recv_func_array) / sizeof(RecvData);

#undef RECV
```

16.3.3.2.2 SendThrough.hh

```
struct SendData {
        u8              size;
};

struct SendStruct sendThrough;

#define SEND(func, entry)     { sizeof(sendThrough.entry) }

const struct SendData sendThrough_func_array[] = {
#include "SendArray.hh"
};

#undef SEND
```

16.3.3.2.3 RecvFunctions.hh

```
const struct RecvData recv_func_array[] = {
      RECV(link.missed),
      RECV(link.cpuTime),
      RECV(GPS),
};
```

16.3.3.2.4 LinkSendStruct.hh

```
struct LinkRelayStruct {
      Atom<7, u32>      AtomicDropped;
};

struct SendStruct {
      struct CommonStruct     common;

      struct LinkRelayStruct  relay;

      Atom<8, u32>      timeout;

      Atom<11, u32>     filler;

      Atomic<12>  flow;
};
```

16.3.3.2.5 LinkSendArray.hh

```
      SEND(send_u32, relay.AtomicDropped),
      SEND(send_u32, timeout),
            ZERO,
            ZERO,
      SEND(send_null, filler),
      SEND(send_null, flow),
```

16.3.3.2.6 LinkRecvStruct.hh

```
// Data Received by robot
#include "RecvStruct.hh"

struct SendStruct {
      struct RecvStruct recv;
      u8    output;
};
```

16.3.3.2.7 LinkRecvArray.hh

```
      { (diff_send)send_null, 0, 0 },
      { (diff_send)send_null, 0, 0 },
      { (diff_send)send_null, 0, 0 },
      SEND(send_GPS, recv.ball),
        SEND(send_GPS, recv.hole),
      SEND(send_c8, output),
```

16.3.3.3 Imperative

16.3.3.3.1 MessageRecv.hh

```
namespace RecvBuffer {
      // Buffer must be 256 long
      u8 buffer[256];
      u8 in, out, commit;

      static inline void init();

      static inline bool put_can();
      static inline void put(u8);
      static inline u8 put_size();
      static inline void put_commit();
      static inline void put_dump();

      static inline bool get_avail();
      static inline u8 get_first();
      static inline u8 get();
};

static inline void RecvBuffer::init()
{
      in = 0;
      out = 0;
      commit = 0;
}

static inline bool RecvBuffer::put_can()
{
      return in + 1 != out;
}
```

```
static inline void RecvBuffer::put(u8 val)
{
      buffer[in++] = val;
}

static inline u8 RecvBuffer::put_size()
{
      return in - out;
}

static inline void RecvBuffer::put_commit()
{
      commit = in;
}

static inline void RecvBuffer::put_dump()
{
      in = commit;
}

static inline bool RecvBuffer::get_avail()
{
      return out != commit;
}

static inline u8 RecvBuffer::get_first()
{
      return get();
}

static inline u8 RecvBuffer::get()
{
      return buffer[out++];
}


namespace MessageBuffer {
      static inline void put_good();

      static inline u8 messages();
      static inline u8 msg_size(AtomID);
}

static inline void MessageBuffer::put_good()
{
      // If data was added, start sending to the computer
      if (RecvBuffer::put_size() > 0)
            message_data_ready();
}

static inline u8 MessageBuffer::messages()
{
      return RobotRecv::atomics * 2;
}
```

```
static inline u8 MessageBuffer::msg_size(AtomID ID)
{
	if (ID & 1)
		return 1;
	else
		return RobotRecv::send_func_array[ID >> 1].size;
}
```

16.3.3.3.2 receive.hh

```
namespace Receive {
	u8 recv_msg_size;
	u8 recv_msg_pos;
	u8 recv_msg_ID;

	static inline void init();

	void recv_byte(u8);
};

static inline void Receive::init()
{
	recv_msg_size = 0;
}

void Receive::recv_byte(u8 byte)
{
	if (recv_msg_size) {
		recv_msg_size--;

		((char *)&send)[recv_msg_pos++] = byte;
	} else {
		if (byte >= atomics) {
			::send.flow.commit();
			return;
		}

		recv_msg_size = send_func_array[byte].size;
		recv_msg_pos = send_func_array[byte].pos;
		recv_msg_ID = byte;
	}

	if (!recv_msg_size)
		AtomicPool::add(recv_msg_ID); // Commit data to send

	// Send message to confirm recieving a byte (flow control)
	::send.flow.commit();
	return;
}
```

16.3.3.3.3 serial.hh

```
#include <sys/sio.h>
#include <sys/interrupts.h>

namespace SerialPort {
	static inline void init();
```

```
};

void __attribute__((interrupt)) sci_handler(void)
{
//    utilization.comp_inter.start();

        if (serial_tx_empty()) {
            if (Send::send_avail()) {
                  serial_send(Send::send_byte());
            } else
                        serial_clr_tx_int();
        }

        if (serial_rx_full()) {
            Transport::Receive::recv_byte(serial_recv());
      }

//    utilization.comp_inter.stop();
}

static inline void SerialPort::init()
{
      serial_baud(38400);
      serial_init();

      while (!serial_rx_full());

      serial_recv();

      *((int *)0x0B16) = (int)sci_handler;

//    _io_ports[M6812_HPRIO] = 0x16;

      serial_set_rx_int();
}

static inline void atom_data_ready(void)
{
      if (!Send::send_avail())
            Send::run_bottom = true;
}

static inline void relay_data_ready(void)
{
      if (serial_tx_empty()) {
            Send::passThrough = false;
            serial_set_tx_int();
      }
}

static inline void message_data_ready(void)
{
      if (serial_tx_empty()) {
            Send::passThrough = true;
            serial_set_tx_int();
      }
```

```
}
```

16.3.4 Graphical User Interface

16.3.4.1 Data Decloration

16.3.4.1.1 types.hh

```
/*template <class T> class BigEndianDataType16 {
      unsigned short data;
public:
      inline operator T() const
      {
            unsigned short x = data;
            __asm__("xchgb %b0,%h0" \
                  : "=q" (x) \
                  :  "" (x));
            return x;
      }
};

template <class T> class BigEndianDataType32 {
      unsigned long data;
public:
      inline operator T() const
      {
            unsigned long x = data;
            __asm__("bswap %0" : "=r" (x) : "" (x));
            return x;
      }
      };*/

/*typedef unsigned char                  u8;
typedef signed char                    s8;
typedef BigEndianDataType16<unsigned short>     u16;
typedef BigEndianDataType16<signed short> s16;
typedef BigEndianDataType32<unsigned long>      u32;
typedef BigEndianDataType32<signed long>  s32;*/

template <class T> class BigEndianDataType {
      T data;
public:
      inline operator T() const
      {
            if (sizeof(T) == 1)
                  return data;
            else if (sizeof(T) == 2) {
                  unsigned short x = data;
                  __asm__("xchgb %b0,%h0" \
                        : "=q" (x) \
                        :  "" (x));
                  return x;
            } else if (sizeof(T) == 4) {
                  unsigned long x = data;
                  __asm__("bswap %0" : "=r" (x) : "" (x));
                  return x;
            }
      }
```

```
};

typedef unsigned char                     u8;
typedef signed char                       s8;
typedef BigEndianDataType<unsigned short> u16;
typedef BigEndianDataType<signed short>         s16;
typedef BigEndianDataType<unsigned long>  u32;
typedef BigEndianDataType<signed long>          s32;
```

16.3.4.1.2 recvAtomic.hh

```
#include <qmutex.h>
#include <qdatetime.h>


class TimeIndex {
private:
        unsigned int          time;

public:
      static QTime      base;

      TimeIndex() { time = base.msecsTo(QTime::currentTime()); }
      TimeIndex(int val) : time(val) {}

        bool operator < (const TimeIndex &right) const { return time <
right.time; }
      QString getTime() const { return
(base.addMSecs(time)).toString("hh:mm:ss.z"); }

      static TimeIndex null() { return TimeIndex(0xFFFFFFFF); }
      bool isnull() const { return time == 0xFFFFFFFF; }

      operator unsigned int() const { return time; }
};


typedef unsigned char AtomID;

class ParseMessage {
public:
      static unsigned char *ptr;
};


class AtomicSize {
private:
      unsigned char           full_size;
      unsigned char           diff_size;

      AtomicSize();
public:
      AtomicSize(unsigned char full, unsigned char diff)
            : full_size(full), diff_size(diff) {}
      unsigned char getsize(bool type) { return type ? diff_size : full_size;
}
```

```
      AtomicSize operator+(AtomicSize right)
      {
            return AtomicSize(full_size + right.full_size,
                              diff_size + right.diff_size);
      }
};

class RecvAtomic :
      public AtomicSize {
private:
      static QMutex           mutex;
      static RecvAtomic *_head, *_tail;
      RecvAtomic        *_next;

protected:
      void queue();

      bool              queueable;

public:
      static RecvAtomic *dequeue();

      RecvAtomic(AtomicSize val) : AtomicSize(val), _next(0), queueable(true)
{}

      virtual     void        recv(bool) = 0;
      virtual void            update() = 0;
      virtual void            update(TimeIndex) = 0;
};

// From recvSignals.cpp
extern RecvAtomic *recv_class_array[];
extern unsigned int NumRecvAtomics;

static inline unsigned char size_recv(AtomID ID)
{
      return recv_class_array[ID >> 1]->getsize(ID & 1);
}

static inline void call_recv(AtomID ID, void *src)
{
      ParseMessage::ptr = (unsigned char*)src;

      recv_class_array[ID >> 1]->recv(ID & 1);
}
```

16.3.4.1.3 messageData.hh

```
#include <qstring.h>
#include <math.h>

#include "recvAtomic.hh"
#include "gpsCord.hh"

class MessageData :
```

```
      public TimeIndex {
public:
      MessageData() : TimeIndex() {}
      MessageData(const TimeIndex &val) : TimeIndex(val) {}
};

// Basic Data Types
template <class T> class template_RawData {
public:
      T     data;

      template_RawData() {}
      template_RawData(T val) : data(val) {}

      int getIntValue() const { return data; }
      double getDoubleValue() const { return data; }
      QString getStringValue() const
      {
            QString str;
            str.setNum(getIntValue());
            return str;
      }
};

typedef template_RawData<unsigned char>         u8_RawData;
typedef template_RawData<signed char>           s8_RawData;
typedef template_RawData<unsigned short>  u16_RawData;
typedef template_RawData<signed short>          s16_RawData;
typedef template_RawData<unsigned long>         u32_RawData;
typedef template_RawData<signed long>           s32_RawData;



template <class T> class template_MessageData :
      public MessageData,
      public template_RawData<T>
{
public:
      template_MessageData(T val) : MessageData(), template_RawData<T>(val)
{}
      template_MessageData(template_RawData<T> val) : MessageData(),
template_RawData<T>(val) {}
      template_MessageData(const TimeIndex &val) : MessageData(val) {}
};

typedef template_MessageData<unsigned char>     u8_MessageData;
typedef template_MessageData<signed char> s8_MessageData;
typedef template_MessageData<unsigned short>    u16_MessageData;
typedef template_MessageData<signed short>      s16_MessageData;
typedef template_MessageData<unsigned long>     u32_MessageData;
typedef template_MessageData<signed long> s32_MessageData;


class Time_RawData {
protected:
      unsigned char     hour;
```

```
	unsigned char	minute;
	unsigned char	second;
public:
	Time_RawData() {}
	Time_RawData(unsigned char _hour, unsigned char _minute, unsigned char
_second)
		: hour(_hour), minute(_minute), second(_second) {}

	QString getStringValue() const
	{
		QString str;
		QString tmp;

		str.setNum(hour);
		tmp.setNum(minute);
		str.append(':');
		str.append(tmp);
		tmp.setNum(second);
		str.append(":");
		str.append(tmp);

		return str;
	}
};

class GPSCordinate_RawData :
	public GPSCordinate {
public:
	GPSCordinate_RawData() {}
	GPSCordinate_RawData(SubCord _lat, SubCord _lon) :
		GPSCordinate(_lat, _lon) {}
};

class GPSStatus_RawData {
protected:
	enum States { no_fix = 0, non_diff = 1, diff = 2, estimate = 6 };
	States state;
	unsigned int satelites;

public:
	GPSStatus_RawData() {}
	GPSStatus_RawData(unsigned char val)
	{
		state = (States)(val >> 4);
		satelites = val & 0x0F;
	}

	int getIntValue() const { return satelites; }

	QString getStringValue() const
	{
		switch (state) {
		case no_fix:
			return QString("Not Available");
		case non_diff:
			return QString("Non Differential");
```

```
            case diff:
                  return QString("Differential");
            case estimate:
                  return QString("Estimated");
            default:
                  return QString("BAD");
            }
      }
};

class GPS_MessageData :
      public MessageData {
public:
      GPS_MessageData(GPSStatus_RawData _status, GPSCordinate_RawData _cord,
s32_RawData _height, u16_RawData _dilution)
            : MessageData(), status(_status), cordinate(_cord),
height(_height), dilution(_dilution) {}
      GPS_MessageData(const TimeIndex &val) : MessageData(val) {}

      GPSStatus_RawData status;
      GPSCordinate_RawData    cordinate;
      s32_RawData       height;
      u16_RawData       dilution;
};
```

16.3.4.1.4 parseMessage.hh

```
#include <qobject.h>

#include "types.hh"

#include "messageData.hh"


class AtomicSignals : public QObject {
      Q_OBJECT
signals:
      void valueChanged(int);
      void valueChanged(double);
      void valueChanged(SubCord);
      void valueChanged(GPSCordinate);
      void valueChanged(const QString &);
};

template <class Data, class Diff> class template_RawParseMessage :
      public ParseMessage,
      public AtomicSignals {
private:
      Data  data;
public:
      template_RawData<Data> getval(bool type)
      {
            if (type) {
                  data += *(Diff *)((BigEndianDataType<Diff> *)ptr);
                  ptr += sizeof(Diff);
            } else {
```

```
                data = *(BigEndianDataType<Data> *)ptr;
                ptr += sizeof(Data);
        }

        return data;
    }

    AtomicSize getTypeSize() { return AtomicSize(sizeof(Data),
sizeof(Diff)); }

    void emitSignals(const template_RawData<Data> *data) {
valueChanged(data->getIntValue()); }
};

template <class Data, class Diff> class template_ParseMessage :
    public template_RawParseMessage<Data, Diff> {
public:
    template_MessageData<Data> getval(bool type)
    {
        return template_RawParseMessage<Data, Diff>::getval(type);
    }
};

class SubCord_ParseMessage :
    public template_ParseMessage<signed long, signed char> {
public:
    void emitSignals(const SubCord data) { emit valueChanged(data); }
};

class Time_ParseMessage :
    public ParseMessage,
    public AtomicSignals {
private:
    unsigned char     hour;
    unsigned char     minute;
    unsigned char     second;

public:
    Time_RawData getval(bool type)
    {
        if (type) {
            if (second == 59) {
                second = 0;
                if (minute == 59) {
                    minute = 0;
                    if (hour == 23)
                        hour = 0;
                    else
                        hour++;
                } else
                    minute++;
            } else
                second++;
        } else {
            hour = *((u8 *)(ptr++));
            minute = *((u8 *)(ptr++));
```

```
                second = *((u8 *)(ptr++));
        }

        return Time_RawData(hour, minute, second);
    }

    AtomicSize getTypeSize() { return AtomicSize(3, 0); }

    void emitSignals(const Time_RawData *data) { valueChanged(data-
>getStringValue()); }
};

class GPSCordinate_ParseMessage :
    public ParseMessage,
    public AtomicSignals {
public:
    SubCord_ParseMessage latitude;
    SubCord_ParseMessage longitude;

    GPSCordinate_RawData getval(bool type)
    {
        SubCord lat = latitude.getval(type).getIntValue();
        SubCord lon = longitude.getval(type).getIntValue();
        return GPSCordinate_RawData(lat,
                                    lon);
    }

    AtomicSize getTypeSize()
    {
        return latitude.getTypeSize() +
              longitude.getTypeSize();
    }

    void emitSignals(const GPSCordinate_RawData *data)
    {
        latitude.emitSignals(data->getLatitude());
        longitude.emitSignals(data->getLongitude());

        valueChanged(*data);
    }
};

class GPSStatus_ParseMessage :
    public ParseMessage,
    public AtomicSignals {
public:
    GPSStatus_RawData getval(bool type)
    {
        return GPSStatus_RawData(*((u8 *)(ptr++)));
    }

    AtomicSize getTypeSize() { return AtomicSize(1, 1); }

    void emitSignals(const GPSStatus_RawData *data)
    {
        valueChanged(data->getIntValue());
```

```
            valueChanged(data->getStringValue());
      }
};

class GPSHeight_ParseMessage :
      public template_RawParseMessage<signed long, unsigned char> {
public:

      void emitSignals(const s32_RawData *data)
      {
            QString str;
            str.setNum(data->getDoubleValue() / 10);
            str.append(" m");

            valueChanged(str);
      }
};

class GPS_ParseMessage {
public:
      GPSStatus_ParseMessage                          status;
      GPSCordinate_ParseMessage                       cordinate;
      GPSHeight_ParseMessage                          height;
      template_RawParseMessage<unsigned short, unsigned char>     dilution;

      GPS_MessageData getval(bool type)
      {
            return GPS_MessageData(status.getval(type),
cordinate.getval(type), height.getval(type), dilution.getval(type));
      }

      AtomicSize getTypeSize() { return status.getTypeSize() +
                                  cordinate.getTypeSize() +
height.getTypeSize() +
      dilution.getTypeSize(); }

      void emitSignals(const GPS_MessageData *data)
      {
            status.emitSignals(&data->status);
            cordinate.emitSignals(&data->cordinate);
            height.emitSignals(&data->height);
            dilution.emitSignals(&data->dilution);
      }
};


class null_ParseMessage :
      public ParseMessage,
      public AtomicSignals {
private:
      unsigned long count;
public:
      u32_MessageData getval(bool type)
      {
            return ++count;
      }
```

```
      AtomicSize getTypeSize() { return AtomicSize(0,0); }

      void emitSignals(const u32_MessageData *data)
      {
            valueChanged(data->getIntValue());
      }
};
```

16.3.4.1.5 atomicCollection.hh

```
#include "parseMessage.hh"
#include "atomicSet.hh"


void registerUpdate(TimeIndex, RecvAtomic *);

template <class Data, class Info, class Parse> class
template_AtomicDataCollection :
      public RecvAtomic,
      public Parse,
      private AtomicDataSet<Data, Info> {
private:
      const char  *name;
      QMutex            mutex;

      template_AtomicDataCollection();
public:
      template_AtomicDataCollection(const char *val)
            : RecvAtomic(getTypeSize()),
            AtomicDataSet<Data, Info>(),
            name(val) {}

      virtual     void        recv(bool);
      virtual void            update();
      virtual void            update(TimeIndex);

      virtual ~template_AtomicDataCollection() {}
};

template <class Data, class Info, class Parse>
void template_AtomicDataCollection<Data, Info, Parse>::recv(bool type)
{
      Data val = getval(type);

      mutex.lock();
      add(val);
      mutex.unlock();

      queue();
}

template <class Data, class Info, class Parse>
void template_AtomicDataCollection<Data, Info, Parse>::update()
{
      mutex.lock();
```

```
	value_type value = last();
	mutex.unlock();

	queueable = true;

	emitSignals(value.first);
}

template <class Data, class Info, class Parse>
void template_AtomicDataCollection<Data, Info, Parse>::update(TimeIndex
index)
{
//	qDebug("PRE FIND");
	mutex.lock();
	complete_type complete = find(index);
//	qDebug("POST FIND");
	mutex.unlock();

	if (complete.second.isnull())
		queueable = true;
	else {
//		qDebug("OLD");
		queueable = false;
		registerUpdate(complete.second, this);
	}

	emitSignals(complete.first.first);

//	qDebug("POST UPDATE");
}

class null_RecvAtomic :
	public template_AtomicDataCollection<u32_MessageData,
	AtomicInfo,
	null_ParseMessage>
{
public:
	null_RecvAtomic(const char *val) :
		template_AtomicDataCollection<u32_MessageData,
					AtomicInfo,
					null_ParseMessage>(val) {}
};

template <class Data, class Diff> class template_RecvAtomic :
	public template_AtomicDataCollection<template_MessageData<Data>,
				AtomicInfo,
				template_ParseMessage<Data, Diff> >
{
public:
	template_RecvAtomic(const char *val) :
		template_AtomicDataCollection<template_MessageData<Data>,
					AtomicInfo,
					template_ParseMessage<Data, Diff> >(val)
{}
};
```

```
class GPS_RecvAtomic :
      public template_AtomicDataCollection<GPS_MessageData,
      AtomicInfo,
      GPS_ParseMessage>
{
public:
      GPS_RecvAtomic(const char *val) :
            template_AtomicDataCollection<GPS_MessageData,
                                          AtomicInfo,
                                          GPS_ParseMessage>(val) {}
};
```

16.3.4.1.6 atomicCollection.cc

```
//#include <qapplication.h>
#include <qlabel.h>
#include <qlcdnumber.h>
#include <qprogressbar.h>

#include "common.hh"
#include "atomicCollection.hh"
#include "arbitrator.hh"

typedef template_RecvAtomic<unsigned char, unsigned char>   uu8_RecvAtomic;
typedef template_RecvAtomic<unsigned short, signed char>    us16_RecvAtomic;
typedef template_RecvAtomic<unsigned long, unsigned char>   uu32_RecvAtomic;


uu32_RecvAtomic   robot_dropped("Robot Dropped");
uu32_RecvAtomic   robot_lost("Robot Lost");
uu32_RecvAtomic robot_sent("Robot Sent");
uu32_RecvAtomic robot_overflow("Robot Overflow");
uu32_RecvAtomic robot_checksum("Robot Checksum");
uu32_RecvAtomic robot_malformed("Robot Malformed");

null_RecvAtomic robot_reset("Robot Reset");

GPS_RecvAtomic    robot_GPS("GPS");

uu8_RecvAtomic    pot1("Pot 1");
uu32_RecvAtomic   robot_badmessages("Pot 2");

null_RecvAtomic   robot_filler("Robot Filler Message");

us16_RecvAtomic robot_heading("Robot Heading");


uu32_RecvAtomic   link_dropped("Link Dropped");
uu32_RecvAtomic   link_lost("Link Lost");
uu32_RecvAtomic link_sent("Link Sent");
uu32_RecvAtomic link_overflow("Link Overflow");
uu32_RecvAtomic link_checksum("Link Checksum");
uu32_RecvAtomic link_malformed("Link Malformed");

uu32_RecvAtomic   comp_dropped("Computer Dropped");
```

```
uu32_RecvAtomic link_timeout("Trans Timeout");

uu32_RecvAtomic link_filler("Link Filler Message");


class flow_RecvAtomic : public RecvAtomic
{
public:
      flow_RecvAtomic();

      virtual void recv(bool);
      virtual void update();
      virtual void update(TimeIndex);
};

flow_RecvAtomic::flow_RecvAtomic() : RecvAtomic(AtomicSize(0,0)) {}
void flow_RecvAtomic::recv(bool)
{
      arbitrate.flowRecv();
}

void flow_RecvAtomic::update() {}
void flow_RecvAtomic::update(TimeIndex) {}


flow_RecvAtomic link_flow;


null_RecvAtomic empty("EMPTY");


RecvAtomic *recv_class_array[] = {
      &robot_dropped,
      &robot_lost,
      &robot_sent,
      &robot_overflow,
      &robot_checksum,
      &robot_malformed,
      &empty,

      &robot_reset,

      &robot_GPS,

      &pot1,
      &robot_badmessages,

      &robot_filler,

      &empty,     // 12
      &empty,     // 13
      &empty,     // 14
      &empty,     // 15
      &robot_heading,   // 16
```

```
      &link_dropped,
      &link_lost,
      &link_sent,
      &link_overflow,
      &link_checksum,
      &link_malformed,
      &empty,

      &comp_dropped,

      &link_timeout,

      &empty,
      &empty,

      &link_filler,
      &link_flow,
};

unsigned int NumRecvAtomics =
sizeof(recv_class_array) / sizeof(RecvAtomic *);

void initRecv(QObject *canvas)
{
/*    QObject::connect(&robot_GPS.time, SIGNAL(valueChanged(const QString
&)),
      mainWin->Value_GPS_Time, SLOT(setText(const QString &)));*/
      QObject::connect(&robot_GPS.status, SIGNAL(valueChanged(const QString
&)),
                  mainWin->Value_GPS_Status, SLOT(setText(const QString
&)));
      QObject::connect(&robot_GPS.status, SIGNAL( valueChanged(int) ),
                  mainWin->Value_GPS_Satelites, SLOT( setNum(int) ));

      QObject::connect(&robot_GPS.cordinate.latitude, SIGNAL(
valueChanged(SubCord) ),
                  mainWin->Cord_Robot_Lat, SLOT( setCord(SubCord) ));
      QObject::connect(&robot_GPS.cordinate.longitude, SIGNAL(
valueChanged(SubCord) ),
                  mainWin->Cord_Robot_Lon, SLOT( setCord(SubCord) ));
      QObject::connect(&robot_GPS.height, SIGNAL( valueChanged(const QString
&) ),
                  mainWin->Value_GPS_Height, SLOT( setText(const QString &)
));
      QObject::connect(&robot_GPS.dilution, SIGNAL( valueChanged(int) ),
                  mainWin->Value_GPS_Dilution, SLOT( setNum(int) ));

      QObject::connect(&robot_GPS.cordinate, SIGNAL(
valueChanged(GPSCordinate) ),
                  canvas, SLOT(robotSet(GPSCordinate)));
      QObject::connect(&robot_heading, SIGNAL( valueChanged(int) ),
                  canvas, SLOT(robotRotate(int)));


      QObject::connect(&robot_heading, SIGNAL( valueChanged(int) ),
                  mainWin->Value_NAV_Heading, SLOT( setNum(int) ));
```

```
QObject::connect(&robot_badmessages, SIGNAL( valueChanged(int) ),
             mainWin->Value_NAV_BadMsg, SLOT( setNum(int) ));


QObject::connect(&robot_dropped, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Dropped, SLOT(setNum(int)));

QObject::connect(&robot_lost, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Lost, SLOT(setNum(int)));

QObject::connect(&robot_sent, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Sent, SLOT(setNum(int)));

QObject::connect(&robot_overflow, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Overflow, SLOT(setNum(int)));

QObject::connect(&robot_checksum, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Checksum, SLOT(setNum(int)));

QObject::connect(&robot_malformed, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Malformed, SLOT(setNum(int)));

QObject::connect(&robot_filler, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Filler, SLOT(setNum(int)));

QObject::connect(&robot_reset, SIGNAL( valueChanged(int) ),
             mainWin->Value_Robot_Reset, SLOT(setNum(int)));


QObject::connect(&link_dropped, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Dropped, SLOT(setNum(int)));

QObject::connect(&link_lost, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Lost, SLOT(setNum(int)));

QObject::connect(&link_sent, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Sent, SLOT(setNum(int)));

QObject::connect(&link_overflow, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Overflow, SLOT(setNum(int)));

QObject::connect(&link_checksum, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Checksum, SLOT(setNum(int)));

QObject::connect(&link_malformed, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Malformed, SLOT(setNum(int)));

QObject::connect(&link_filler, SIGNAL( valueChanged(int) ),
             mainWin->Value_Link_Filler, SLOT(setNum(int)));



QObject::connect(&comp_dropped, SIGNAL( valueChanged(int) ),
             mainWin->Value_Computer_Dropped, SLOT(setNum(int)));
```

```
      QObject::connect(&link_timeout, SIGNAL( valueChanged(int) ),
                  mainWin->Value_Computer_Timeout, SLOT(setNum(int)));


      QObject::connect(&pot1, SIGNAL( valueChanged(int) ),
                  mainWin->Value_Pot1, SLOT(setProgress(int)));


      QObject::connect(&link_sent, SIGNAL( valueChanged(int) ),
                  mainWin->BIG1, SLOT(display(int)));

      QObject::connect(&link_timeout, SIGNAL( valueChanged(int) ),
                  mainWin->BIG2, SLOT(display(int)));

      QObject::connect(&pot1, SIGNAL( valueChanged(int) ),
                  mainWin->BIG3, SLOT(setProgress(int)));
}
```

16.3.4.1.7 atomicSet.hh

```
#include <map.h>

struct AtomicInfo {
//    MessageListViewItem     *currentMSG;
};


template <class _Data, class _Info> class AtomicDataSet {
private:
      typedef map<_Data, _Info>     MapType;
      MapType                       dataSet;
      MapType::iterator       dataIterator;

protected:
      typedef std::pair<const _Data *, _Info *> value_type;
      typedef std::pair<value_type, TimeIndex>  complete_type;

      AtomicDataSet() : dataIterator(dataSet.end()) {}

      void add(const _Data &__k)
      {
//          qDebug("Set Add Pre");
            MapType::iterator __i = dataSet.lower_bound(__k);
            // __i->first is greater than or equivalent to __k.
            if (__i == dataSet.end() || dataSet.key_comp()(__k,
(*__i).first))
                  __i = dataSet.insert(__i, MapType::value_type(__k,
_Info()));

            //    return value_type(&__i->first, &__i->second);
//          qDebug("Set Add Post");
      }

      complete_type find(const TimeIndex &__k)
      {
//          qDebug("Set Find Pre");
```

```
            MapType::iterator __n = dataSet.upper_bound(static_cast<const
_Data>(__k));
            MapType::iterator __i = __n;
            __i--;

            TimeIndex next = TimeIndex::null();
            if (__n != dataSet.end()) {
                  next = __n->first;
                  //qDebug("OLD %u - %u", __k, next);
            }

//          qDebug("Set Find Post");

            return complete_type(value_type(&__i->first, &__i->second),
                                 next);
      }

      value_type last()
      {
            MapType::iterator __v = dataSet.end();
            __v--;

            return value_type(&__v->first, &__v->second);
      }

      bool have_next()
      {
            return dataIterator != dataSet.end();
      }

      value_type next()
      {
            dataIterator--;
            return value_type(&dataIterator->first, &dataIterator->second);
      }
};

#include <map.h>

struct AtomicInfo {
//    MessageListViewItem     *currentMSG;
};


template <class _Data, class _Info> class AtomicDataSet {
private:
      typedef map<_Data, _Info>     MapType;
      MapType                       dataSet;
      MapType::iterator       dataIterator;

protected:
      typedef std::pair<const _Data *, _Info *> value_type;
      typedef std::pair<value_type, TimeIndex>  complete_type;

      AtomicDataSet() : dataIterator(dataSet.end()) {}
```

```
	void add(const _Data &__k)
	{
//		qDebug("Set Add Pre");
		MapType::iterator __i = dataSet.lower_bound(__k);
		// __i->first is greater than or equivalent to __k.
		if (__i == dataSet.end() || dataSet.key_comp()(__k,
(*__i).first))
			__i = dataSet.insert(__i, MapType::value_type(__k,
_Info()));

		//    return value_type(&__i->first, &__i->second);
//		qDebug("Set Add Post");
	}

	complete_type find(const TimeIndex &__k)
	{
//		qDebug("Set Find Pre");

		MapType::iterator __n = dataSet.upper_bound(static_cast<const
_Data>(__k));
		MapType::iterator __i = __n;
		__i--;

		TimeIndex next = TimeIndex::null();
		if (__n != dataSet.end()) {
			next = __n->first;
			//qDebug("OLD %u - %u", __k, next);
		}

//		qDebug("Set Find Post");

		return complete_type(value_type(&__i->first, &__i->second),
					next);
	}

	value_type last()
	{
		MapType::iterator __v = dataSet.end();
		__v--;

		return value_type(&__v->first, &__v->second);
	}

	bool have_next()
	{
		return dataIterator != dataSet.end();
	}

	value_type next()
	{
		dataIterator--;
		return value_type(&dataIterator->first, &dataIterator->second);
	}
};
```

16.3.4.1.8 sendSignals.hh

```
#include <qobject.h>
#include "types.hh"

#include "gpsCord.hh"

class SendMessage : public QObject {
      Q_OBJECT
private:
      SendMessage();
protected:
      unsigned char ID;
public:
      SendMessage(int);
};

class Send_null : public SendMessage {
      Q_OBJECT
public:
      Send_null(int val) : SendMessage(val) {}
public slots:
      void setValue();
};

class Send_u8 : public SendMessage {
      Q_OBJECT
public:
      Send_u8(int val) : SendMessage(val) {}
public slots:
      void setValue(int);
};

class Send_GPSCordinate : public SendMessage {
      Q_OBJECT
public:
      Send_GPSCordinate(int val) : SendMessage(val) {}
public slots:
      void setValue(GPSCordinate);
};
```

16.3.4.1.9 sendSignals.cc

```
#include <qapplication.h>
#include <qslider.h>

#include "common.hh"
#include "sendSignals.hh"
#include "arbitrator.hh"


SendMessage::SendMessage(int val) : ID(val) {}

void Send_null::setValue()
{
      unsigned char buffer[1];
```

```
	buffer[0] = ID;

	arbitrate.writeMessage(buffer, 1);
}

void Send_u8::setValue(int val)
{
	unsigned char buffer[2];

	buffer[0] = ID;
	buffer[1] = (u8)val;

	arbitrate.writeMessage(buffer, 2);
}

void Send_GPSCordinate::setValue(GPSCordinate data)
{
/*	struct {
		unsigned char id;
		long  latitude;
		long  longitude;
		} cord  __attribute__((packed));*/

	unsigned char buffer[9];

	buffer[0] = ID;

	long lat = data.getLatitude().getNumber() - 20430000;
	long lon = -data.getLongitude().getNumber() - 64140000;

	printf("%d, %d\n", data.getLatitude().getNumber(),
		data.getLongitude().getNumber());

	*((long *)&buffer[1]) = *((s32*)(&lat));
	*((long *)&buffer[5]) = *((s32*)(&lon));

/*	cord.id = ID;
	cord.latitude = 0; /((s32*)(&lat));
	cord.longitude = 0; /((s32*)(&lon));*/

	arbitrate.writeMessage(buffer, 9);
};

Send_null start_action(1);
Send_GPSCordinate ball_cord(3);
Send_GPSCordinate hole_cord(4);
Send_u8     output(5);

void initSend()
{
	QObject::connect(mainWin, SIGNAL(startAction()),
			&start_action, SLOT(setValue()));

	QObject::connect(mainWin, SIGNAL(ballSet(GPSCordinate)),
			&ball_cord, SLOT(setValue(GPSCordinate)));
	QObject::connect(mainWin, SIGNAL(holeSet(GPSCordinate)),
```

```
                  &hole_cord, SLOT(setValue(GPSCordinate)));

      QObject::connect(mainWin->Set_Output1, SIGNAL( valueChanged(int) ),
                  &output, SLOT(setValue(int)));
}
```

16.3.4.1.10 gpsCord.hh

```
#ifndef _H_GPSCORD
#define _H_GPSCORD

class SubCord {
private:
      long  cord;

      static const long _invalid = 0x80000000;

public:
      SubCord() { cord = _invalid; }
      SubCord(long val) : cord(val) {}
      SubCord(int deg, float min)
      {
            bool negative;
            if (negative = deg < 0)
                  deg *= -1;

            cord = (deg * 600000) +
                  (long)(min * 10000.0 + 0.5);

            if (negative)
                  cord *= -1;
      }

      SubCord(int deg, unsigned int min, float sec)
      {
            bool negative;
            if (negative = deg < 0)
                  deg *= -1;

            cord = (deg * 600000) +
                  (min * 10000) +
                  (long)(sec * 10000.0/60.0 + 0.5);

            if (negative)
                  cord *= -1;
      }

      bool invalid() { return cord == _invalid; }

      int getDeg() const { return cord / 600000; }
      unsigned int getIntMin() const { return (abs(cord) / 10000) % 60; }
      float getFloatMin() const { return (float)(abs(cord) % 600000) /
10000.0; }
      float getSec() const { return (float)(abs(cord) % 10000) *
(60.0/10000.0); }
```

```
      long getNumber() const { return cord; }
};

class GPSCordinate {
protected:
      SubCord     latitude;
        SubCord   longitude;
//    long  height;

public:
      GPSCordinate() {}
      GPSCordinate(SubCord _lat, SubCord _lon) :
            latitude(_lat), longitude(_lon) {}

      SubCord getLatitude() const { return latitude; }
      SubCord getLongitude() const { return longitude; }
};

class GPSCordinateAngle : public GPSCordinate {
protected:
      unsigned short    angle;

public:
      GPSCordinateAngle() {}
      GPSCordinateAngle(SubCord _lat, SubCord _lon, long _angle) :
            GPSCordinate(_lat, _lon), angle(_angle) {}

      long getAngle() const { return angle; }
};

#endif
```

16.3.4.2 Data Handlers

16.3.4.2.1 arbitrator.hh

```
#include <qobject.h>
#include <qsocketnotifier.h>
#include <qthread.h>

class CIOArbitrator
      : public QThread
{
private:
      int               confd;

      char outbuffer[256];
      unsigned char in, out;
      bool empty;

public:
      virtual void run();

      void writeMessage(void *, int len);

      void flowRecv();
```

```
private:
      void dataReceived();
};

extern CIOArbitrator arbitrate;
```

16.3.4.2.2 arbitrator.cc

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <unistd.h>

#include <sys/socket.h>


#include "arbitrator.hh"

#include "recvAtomic.hh"

//bool init = true;

void CIOArbitrator::run()
{
      struct termios newtio;

      confd = open("/dev/tts/0", O_RDWR | O_NOCTTY);

      newtio.c_cflag = B38400 | CS8 | CLOCAL | CREAD;
      newtio.c_iflag = IGNPAR;
      newtio.c_oflag = 0;
      newtio.c_lflag = 0;
      newtio.c_line = 0;

      newtio.c_cc[VINTR]    = 0;     // Ctrl-c
      newtio.c_cc[VQUIT]    = 0;     // Ctrl-\
      newtio.c_cc[VERASE]   = 0;     // del
      newtio.c_cc[VKILL]    = 0;     // @
      newtio.c_cc[VEOF]     = 4;     // Ctrl-d
      newtio.c_cc[VTIME]    = 0;     // inter-character timer unused
      newtio.c_cc[VMIN]     = 1;     // blocking read until 1 character
arrives
      newtio.c_cc[VSWTC]    = 0;     // '\0
      newtio.c_cc[VSTART]   = 0;     // Ctrl-q
      newtio.c_cc[VSTOP]    = 0;     // Ctrl-s
      newtio.c_cc[VSUSP]    = 0;     // Ctrl-z
      newtio.c_cc[VEOL]     = 0;     // '\0'
      newtio.c_cc[VREPRINT] = 0;     // Ctrl-r
      newtio.c_cc[VDISCARD] = 0;     // Ctrl-u
      newtio.c_cc[VWERASE]  = 0;     // Ctrl-w
      newtio.c_cc[VLNEXT]   = 0;     // Ctrl-v
      newtio.c_cc[VEOL2]    = 0;     // '\0'
      newtio.c_cc[17]       = 0;
```

```
	newtio.c_cc[18]       = 0;

	tcflush(confd, TCIFLUSH);
	tcsetattr(confd, TCSANOW, &newtio);

	char byte = 0xFF;
	write(confd, &byte, 1);

	in = 0;
	out = 0;
	empty = true;


/*	confd = ::socket(AF_UNIX, SOCK_STREAM, 0);

	  struct sockaddr_un address;
	  address.sun_family = AF_UNIX;
	  strcpy(address.sun_path, "../socket");

	  ::connect(confd, (struct sockaddr *)&address, sizeof(address));*/

	while (1)
		dataReceived();

}

void print(unsigned char *buffer, unsigned char pos)
{
	for (unsigned char i = 0; i < pos; i++)
		printf("%02x,", buffer[i]);
	printf("\n");
}

unsigned int offset = 0;
unsigned char buffer[256];

void CIOArbitrator::dataReceived()
{
	int len = read(confd, buffer + offset, 256 - offset);

/*	if (init) {
		for (int i = 0; i < len; i++) {
		  if (buffer[i] == 0xFF) {
			pre_count++;
			if (pre_count == 16) {
				init = false;
				printf("INIT COMPLETE");
				goto out;
			}
		  else
			pre_count = 0;
		  }
		}

		return;
	}
```

```
	out:*/

//	printf("RECV %d: ", offset);
//	print(buffer + offset, len);

	len += offset;
	offset = 0;

	for (int i = 0; i < len; i++) {
		AtomID ID = buffer[i];
		if (ID >= (NumRecvAtomics * 2)) {
			qFatal("BAD MESSAGE ID");
			print(buffer, len);
		}

		unsigned char size = size_recv(ID);

		if (i + size >= len) {
			for (int j = i; j < len; j++) {
				buffer[j - i] = buffer[j];
			}
			offset = len - i;
			break;
		}

		i++;

//		printf("MSG %x:", ID); print(buffer + i, size);
		call_recv(ID, (void *)((char *)buffer + i));

		i += size - 1;
	}
}

void CIOArbitrator::writeMessage(void *ptr, int len)
{
//	if (in + 1 == out)
//		qFatal("WRITE BUFFER FULL");

	printf("Send "); print((unsigned char *)ptr, len);

	memcpy(&outbuffer[in], ptr, len);
	in += len;

	if (empty) {
		flowRecv();
		empty = false;
	}

//	write(confd, ptr, len);
}

void CIOArbitrator::flowRecv()
{
	if (in == out) {
```

```
            if (empty)
                  qFatal("No MORE DATA TO SEND");
            else {
                  printf("SEND TERM\n");
                  empty = true;
                  return;
            }
      }

      printf("SEND BYTE %d,%d - %02x\n", in, out, outbuffer[out]);

      write(confd, &outbuffer[out++], 1);
}

CIOArbitrator arbitrate;
```

16.3.4.2.3 timeFlow.hh

```
#include <qobject.h>
#include <qtimer.h>
#include <multimap.h>

#include "recvAtomic.hh"

const int   play_divisor = 4;

class TimeFlow : public QObject {
      Q_OBJECT
private:
      int         play_speed;

      unsigned int      start_time;
      unsigned int      base_time;

      typedef multimap<TimeIndex, RecvAtomic *> times_type;
      times_type  times;

      QTimer            timer;
      QTimer            periodic;

      friend void registerUpdate(TimeIndex, RecvAtomic *);

      TimeIndex currentTime();

      void clearTimer();
      void setupTimer();

      void updateTime();

private slots:
      void timerEvent();
      void periodicEvent();

public slots:
      void setTime(unsigned int val);
      void setSpeed(int val);
```

```
public:
      TimeFlow() :
            play_speed(play_divisor),
            start_time(0), base_time(0)
      {
            connect(&timer, SIGNAL(timeout()),
                  this, SLOT(timerEvent()));
            connect(&periodic, SIGNAL(timeout()),
                  this, SLOT(periodicEvent()));
            periodic.start(100);
      }

      virtual bool event(QEvent *e);
};
```

16.3.4.2.4 timeFlow.cc

```
#include <qthread.h>
#include <qslider.h>
#include <qlabel.h>

#include "common.hh"
#include "timeFlow.hh"

QTime TimeIndex::base;
unsigned char *ParseMessage::ptr;

QMutex RecvAtomic::mutex;
RecvAtomic *RecvAtomic::_head = 0;
RecvAtomic *RecvAtomic::_tail = 0;

TimeFlow    flow;

void RecvAtomic::queue()
{
//    qDebug("PRE QUEUE");
      mutex.lock();
      if (queueable &&
          !_next) {
            if (_head)
                  _tail->_next = this;
            else {
                  _head = this;
                  QThread::postEvent(&flow, new QEvent(QEvent::User));
            }
            _tail = this;
      }
      mutex.unlock();
//    qDebug("POST QUEUE");
}

RecvAtomic *RecvAtomic::dequeue()
{
//    qDebug("PRE DEQUEU");
      mutex.lock();
```

```
	RecvAtomic *ret = 0;
	while (_head) {
		ret = _head;

		_head = ret->_next;
		ret->_next = 0;

		if (ret->queueable)
			break;
	}
	mutex.unlock();
//	qDebug("POST DEQUEU");
	return ret;
}


void registerUpdate(TimeIndex index, RecvAtomic *atomic)
{
	qDebug("Register Pre %d - %x", index, atomic);

	for (TimeFlow::times_type::iterator i = flow.times.lower_bound(index);
	     i != flow.times.upper_bound(index);
	     i++) {
		if (i->second == atomic) {
			qFatal("Register Post Duplicate");
			return;
		}
	}

	flow.times.insert(TimeFlow::times_type::value_type(index, atomic));
	qDebug("Register Post");
}




TimeIndex TimeFlow::currentTime()
{
	unsigned int elapsed = TimeIndex();

	unsigned int current =
		((elapsed - start_time) * play_speed)
			/ play_divisor
		+ base_time;

	if (current > elapsed) {
		//play_speed = play_divisor;
		mainWin->Slider_Speed->setValue(play_divisor);
		base_time = start_time = current = TimeIndex();
	}

	return current;
}

void TimeFlow::periodicEvent()
{
```

```
	TimeIndex current;
	mainWin->Value_CurrentTime->setText(current.getTime());
	mainWin->Slider_Time->setMaxValue(current);
	mainWin->Slider_Time->setValue(currentTime());
}


void TimeFlow::clearTimer()
{
	timer.stop();
	times.clear();
}

void TimeFlow::setupTimer()
{
	times_type::iterator i;

	TimeIndex current(0);

	while ((i = times.begin()) != times.end() &&
	       i->first <= (current = currentTime())) {
		qDebug("TIMER UPDATE");
		i->second->update(current);
		times.erase(i);
	}

	if (i != times.end() && play_speed) {
		int val = ((i->first - currentTime()) *
			   play_divisor) / play_speed;
		qDebug("TIMER START %d", val);
		timer.start(val, true);
	}
}

bool TimeFlow::event(QEvent *e)
{
	if (e->type() != QEvent::User)
		qFatal("EVENT");

//    qDebug("PRE EVENT");

	RecvAtomic *item;

	while ((item = RecvAtomic::dequeue()))
		item->update(currentTime());

//    qDebug("POST EVENT");
	setupTimer();
//    qDebug("POST TIMER");

	return true;
}


void TimeFlow::timerEvent()
{
```

```
//    qDebug("Timer Event");
      setupTimer();
}


void TimeFlow::setTime(unsigned int val)
{
      TimeIndex time = val;

      clearTimer();

      for (unsigned int index = 0; index < NumRecvAtomics; index++) {
            recv_class_array[index]->update(time);
      }

      base_time = val;
      start_time = TimeIndex();

      setupTimer();
}

void TimeFlow::setSpeed(int val)
{
      qDebug("---SET SPEED---");
      base_time = currentTime();
      start_time = TimeIndex();
      play_speed = val;
      setupTimer();
}


void initTime()
{
/*    QObject::connect(mainWin->Slider_Time, SIGNAL(valueChanged(int)),
                  &flow, SLOT(setTime(unsigned int)));
*/
      QObject::connect(mainWin->Slider_Speed, SIGNAL(valueChanged(int)),
                  &flow, SLOT(setSpeed(int)));

      TimeIndex::base = QTime::currentTime();

      TimeIndex current;
      mainWin->Value_StartTime->setText(current.getTime());
}
```

16.3.4.3 Qt

16.3.4.3.1 main.cpp

```
#include <qapplication.h>
#include <qstring.h>
#include <qtable.h>
#include <qlabel.h>
#include <qbuttongroup.h>

#include "widgets.hh"
#include "common.hh"
```

```
#include "arbitrator.hh"

#include "canvasItems.hh"

MainWindow *mainWin;

int main( int argc, char ** argv )
{
    QApplication a( argc, argv );
    mainWin = new MainWindow;
    mainWin->show();
    a.connect( &a, SIGNAL( lastWindowClosed() ), &a, SLOT( quit() ) );

    initSend();
    initTime();

    arbitrate.start();

    BooleanSignalAnd ButtonMux;

    QObject::connect(mainWin->EditLatitude, SIGNAL(valid(bool)),
                &ButtonMux, SLOT(inputA(bool)));
    QObject::connect(mainWin->EditLongitude, SIGNAL(valid(bool)),
                &ButtonMux, SLOT(inputB(bool)));
    QObject::connect(&ButtonMux, SIGNAL(result(bool)),
                mainWin->ButtonGroup, SLOT(setEnabled(bool)));



    GreenCanvasControl canvasControl(mainWin->GreenCanvas);
    QObject::connect(mainWin, SIGNAL(holeSet(GPSCordinate)),
                &canvasControl, SLOT(holeSet(GPSCordinate)));
    QObject::connect(mainWin, SIGNAL(ballSet(GPSCordinate)),
                &canvasControl, SLOT(ballSet(GPSCordinate)));
/*    QObject::connect(mainWin, SIGNAL(robotSet(GPSCordinate)),
      &canvasControl, SLOT(robotSet(GPSCordinate)));*/


    QObject::connect(mainWin->ButtonGroup_Units, SIGNAL(clicked(int)),
                &CordLabelArbitrator, SLOT(unitsChanged(int)));


    initRecv(&canvasControl);

    return a.exec();
}
```

16.3.4.3.2 mainwindow.ui.h

```
/****************************************************************************
** ui.h extension file, included from the uic-generated form implementation.
**
** If you wish to add, delete or rename slots use Qt Designer which will
** update this file, preserving your code. Create an init() slot in place of
** a constructor, and a destroy() slot in place of a destructor.
```

```
*****************************************************************************
/

bool haveHole = false;
bool haveBall = false;

void MainWindow::ballClick()
{
    GPSCordinate cord(EditLatitude->getGPS(),
                  EditLongitude->getGPS());

    Cord_Ball_Lat->setCord(cord.getLatitude());
    Cord_Ball_Lon->setCord(cord.getLongitude());

   haveBall = true;
    if (haveHole)
      PushButton_Action->setEnabled(true);

    emit ballSet(cord);
}

void MainWindow::holeClick()
{
    GPSCordinate cord(EditLatitude->getGPS(),
                  EditLongitude->getGPS());

    Cord_Hole_Lat->setCord(cord.getLatitude());
    Cord_Hole_Lon->setCord(cord.getLongitude());

    haveHole = true;
    if (haveBall)
      PushButton_Action->setEnabled(true);

    emit holeSet(cord);
}

void MainWindow::robotClick()
{
    GPSCordinate cord(EditLatitude->getGPS(),
                  EditLongitude->getGPS());

    Cord_Robot_Lat->setCord(cord.getLatitude());
    Cord_Robot_Lon->setCord(cord.getLongitude());

    emit robotSet(cord);
}


void MainWindow::actionClick()
{
    emit startAction();
}
```

16.3.4.3.3 canvasItems.hh

```
#include <qobject.h>
```

```
#include <qcanvas.h>
#include <qimage.h>
#include <qpixmap.h>

#include <stdio.h>

#include "widgets.hh"

// Copied from QT Canvas example
class ImageItem : public QCanvasRectangle
{
public:
    ImageItem( QImage img, QCanvas *canvas);
protected:
    void drawShape( QPainter & );
private:
    QImage image;
    QPixmap pixmap;
};


class CCanvasView : public QCanvasView
{
      Q_OBJECT
private:
      bool needrePosition;

public:
      CCanvasView( QWidget * parent = 0, const char * name = 0, WFlags f = 0
);
      void rePosition();
};


const long absXmax = 648000000;
const long absYmax = 324000000;

const long centerX = -64144434;
const long centerY = 20439804;
const long scale = 2;
const long regionWidth = 1500;
const long regionHeight = 1500;

const long canvasWidth = scale * regionWidth;
const long canvasHeight = scale * regionHeight;

const long edgeOffset = 1;

const long numberOfImages = 90;

class GreenCanvasControl : public QCanvas
{
      Q_OBJECT
private:
      QCanvasPixmapArray      *robot_array;
```

```
      QCanvasSprite           *robot;

      ImageItem *ball;
      ImageItem *hole;

      CCanvasView *view;

      QImage image;
      QPixmap pixmap;

      void zoom();

      inline long transX(long val)
      {
            long ret = 2 * (val - centerX) + canvasWidth/2;
            printf("X: %d\n", ret);
            return ret;
      }
      inline long transY(long val)
      {
            long ret = 2 * (-val + centerY) + canvasHeight/2;
            printf("Y: %d\n", ret);
            return ret;
      }


public:
      GreenCanvasControl(CCanvasView *);
      QRect getBounds();

public slots:
      void ballSet(GPSCordinate);
      void holeSet(GPSCordinate);
      void robotSet(GPSCordinate);
      void robotRotate(int);
};
```

16.3.4.3.4 canvasItems.cc

```
#include <qpainter.h>

#include "canvasItems.hh"

ImageItem::ImageItem( QImage img, QCanvas *canvas)
      : QCanvasRectangle( canvas ), image(img)
{
    setSize( image.width(), image.height() );

    pixmap.convertFromImage(image, OrderedAlphaDither);
}

void ImageItem::drawShape( QPainter &p )
{
    p.drawPixmap( int(x()), int(y()), pixmap );
}
```

```
CCanvasView::CCanvasView( QWidget * parent = 0, const char * name = 0, WFlags
f = 0 )
      : QCanvasView(parent, name, f) {}

void CCanvasView::rePosition()
{
      QRect rect = static_cast<GreenCanvasControl *>(canvas())->getBounds();

      double scale;

      if ((long long)rect.height() * (long long)visibleWidth() >
          (long long)rect.width() * (long long)visibleHeight())
            scale = (double)visibleHeight() / (double)rect.height();
      else
            scale = (double)visibleWidth() / (double)rect.width();

      QWMatrix mx;
      mx = mx.scale(scale, scale);
      setWorldMatrix(mx);

      int x, y;
      mx.map(rect.center().x(), rect.center().y(), &x, &y);

      printf("SCALE %g %d,%d\n", scale, x, y);

      center(x, y);
}

void GreenCanvasControl::zoom()
{
//    view->rePosition();
      update();
}

QRect GreenCanvasControl::getBounds()
{
      long Xmin = 0x7FFFFFFF;
      long Ymin = 0x7FFFFFFF;
      long Xmax = 0x80000000;
      long Ymax = 0x80000000;

      QCanvasItemList list = allItems();
      QCanvasItemList::Iterator it = list.begin();
      for (; it != list.end(); ++it) {
            QCanvasItem *i = (*it);
            if (i->isVisible()) {
                  QRect rect = i->boundingRect();
                  if (rect.left() < Xmin)
                        Xmin = rect.left();
                  if (rect.right() > Xmax)
                        Xmax = rect.right();
                  if (rect.top() < Ymin)
                        Ymin = rect.top();
                  if (rect.bottom() > Ymax)
                        Ymax = rect.bottom();
```

```
            }
      }

      printf("BOUNDS: %d,%d %d,%d\n",
             Xmin, Ymin, Xmax, Ymax);

      return QRect(Xmin - edgeOffset, Ymin - edgeOffset,
                  Xmax - Xmin + 2*edgeOffset, Ymax - Ymin + 2*edgeOffset);

}

/*QRect GreenCanvasControl::getBounds()
{
      return QRect(0, 0, 5000, 5000);
}*/

GreenCanvasControl::GreenCanvasControl(CCanvasView *val)
//    : QCanvas(absXmax * 2, absYmax * 2), view(val)
      : QCanvas(canvasWidth, canvasHeight), view(val) //, Xtrans(0),
Ytrans(0)
{
      image = QImage("images/background.png");
      pixmap = QPixmap(image);
      setBackgroundPixmap(pixmap);

      hole = new ImageItem(QImage("images/flag.png"), this);
      ball = new ImageItem(QImage("images/ball.png"), this);

//    robot = new ImageItem(QImage("images/robot.png"), this);

//    robot_array = new QCanvasPixmapArray("images/robot%1.png",
numberOfImages);
      QPixmap robot_pixmap(QImage("images/robot.png"));

      QValueList<QPixmap> list;

      for (int i = 0; i < numberOfImages; i++) {
            QWMatrix matrix;
            matrix.rotate((double)i * 360.0/((double)numberOfImages));
            list.append(robot_pixmap.xForm(matrix));
      }

      robot_array = new QCanvasPixmapArray(list);
      robot = new QCanvasSprite(robot_array, this);

      view->setCanvas(this);

/*    ball->move(750, 750);
      ball->show();
      hole->move(750, 750);
      ole->show();*/
//    robot->move(750 - 100, 750 - 100);
//    robotRotate(90);
//    robot->setFrame(45);
      robot->show();
}
```

```
void GreenCanvasControl::ballSet(GPSCordinate cord)
{
      ball->move(transX(cord.getLongitude().getNumber() - 18),
               transY(cord.getLatitude().getNumber()) - 4);
      ball->show();
      zoom();
}
void GreenCanvasControl::holeSet(GPSCordinate cord)
{
      hole->move(transX(cord.getLongitude().getNumber()) - 161,
               transY(cord.getLatitude().getNumber()) - 198);
      hole->show();
      zoom();
}
void GreenCanvasControl::robotSet(GPSCordinate cord)
{
      robot->move(transX(cord.getLongitude().getNumber() - 100),
                transY(cord.getLatitude().getNumber()) - 100);
      robot->show();
      zoom();
}
void GreenCanvasControl::robotRotate(int val)
{
      robot->setFrame((val - 360/(2*numberOfImages))/(360/numberOfImages));
//    robot->show();
}
```

16.3.4.3.5 widgets.hh

```
#ifndef CWIDGETS_H
#define CWIDGETS_H

#include <qregexp.h>
#include <qvalidator.h>
#include <qlineedit.h>
#include <qlabel.h>
#include <qstring.h>

#include "gpsCord.hh"

class CCordLineEdit : public QLineEdit
{
      Q_OBJECT
public:
      CCordLineEdit( const QRegExp & rx, QWidget * parent, const char * name
= 0);

      SubCord getGPS();

      QRegExpValidator localvalidator;
signals:
      void valid(bool);

private slots:
      void privateTextChanged(const QString &str);
```

```
};


class CLatLineEdit : public CCordLineEdit
{
public:
      CLatLineEdit( QWidget * parent, const char * name = 0 );
};


class CLonLineEdit : public CCordLineEdit
{
public:
      CLonLineEdit( QWidget * parent, const char * name = 0 );
};


class BooleanSignalAnd : public QObject
{
      Q_OBJECT
private:
      bool A, B, state;

      inline void sendSig();

public:
      BooleanSignalAnd() : A(false), B(false), state(false) {}

public slots:
      void inputA(bool);
      void inputB(bool);

signals:
      void result(bool);
};


class CCordLabel : public QLabel
{
      Q_OBJECT
public:
      CCordLabel( QWidget * parent, const char * name = 0, WFlags f = 0 );

public slots:
      void setCord(SubCord);

private:
      friend class CCordLabelArbitrator;

      enum Units { seconds, minutes };
      static Units units;

      SubCord cordinate;

private slots:
      void updateText();
```

```
};


class CCordLabelArbitrator : public QObject
{
      Q_OBJECT
signals:
      void updateAll();

public slots:
      void unitsChanged(int val);
};

extern CCordLabelArbitrator CordLabelArbitrator;

#endif
```

16.3.4.3.6 widgets.cc

```
#include <qobject.h>

#include "widgets.hh"

CCordLineEdit::CCordLineEdit( const QRegExp & rx, QWidget * parent, const
char * name = 0)
      : QLineEdit(parent, name), localvalidator(this)
{
      localvalidator.setRegExp(rx);
      setValidator(&localvalidator);
      connect(this, SIGNAL(textChanged(const QString &)),
            this, SLOT(privateTextChanged(const QString &)));
}

void CCordLineEdit::privateTextChanged(const QString &val)
{
      int pos = cursorPosition();
      QString str = val;
      bool res = validator()->validate(str, pos) == QValidator::Acceptable;

      emit valid(res);
}

SubCord CCordLineEdit::getGPS()
{
//    QRegExp rx("(-?\\d*)d(\\d*)\"(\\d*\\.?\\d*)");
/*    QRegExp rx("(-?\\d*)d((\\d*\\.\\d*)|((\\d*)\"(\\d*\\.?\\d*)))");
      rx.search(text());
      return SubCord(rx.cap(1).toInt(),
                  rx.cap(2).toUInt(),
                  rx.cap(3).toFloat());

      QStringList list = rx.capturedTexts();

      printf("NUM %d %s:%s:%s:%s\n", list.count(),
             list[0].ascii(),
```

```
                list[1].ascii(),
                list[2].ascii(),
                list[3].ascii());

        int deg = list[1].toInt();
        if (list.count() == 3)
                return SubCord(deg,
                                list[2].toFloat());
        else if (list.count() == 4)
                return SubCord(deg,
                                list[2].toUInt(),
                                list[3].toFloat());
        else
                return SubCord();
*/

        QRegExp rx("(-?\\d*)d(\\d*\\.?\\d*)\"(\\d*\\.?\\d*)");
        rx.search(text());

        int deg = rx.cap(1).toInt();
        if (rx.cap(2).find('.') == -1)
                return SubCord(deg,
                                rx.cap(2).toUInt(),
                                rx.cap(3).toFloat());
        else
                return SubCord(deg,
                                rx.cap(2).toFloat());
}


CLatLineEdit::CLatLineEdit( QWidget * parent, const char * name = 0 )
//    : CCordLineEdit(QRegExp("-?[0-8]?\\d?d[0-5]?\\d?\"[0-
5]?\\d?((\\.\\d{0,3}')|')$"),
        : CCordLineEdit(QRegExp("-?[0-8]?\\d?d[0-5]?\\d?((\\.\\d{0,4}\")|(\"[0-
5]?\\d?((\\.\\d{0,3}')|')))$"),
                        parent, name)
{}

CLonLineEdit::CLonLineEdit(QWidget * parent, const char * name = 0 )
//    : CCordLineEdit(QRegExp("-?((1[0-7]\\d)|(\\d{0,2}))d[0-5]?\\d?\"[0-
5]?\\d?((\\.\\d{0,3}')|')$"),
        : CCordLineEdit(QRegExp("-?((1[0-7]\\d)|(\\d{0,2}))d[0-
5]?\\d?((\\.\\d{0,4}\")|(\"[0-5]?\\d?((\\.\\d{0,3}')|')))$"),
                        parent, name)

{}


inline void BooleanSignalAnd::sendSig()
{
        bool val = A & B;

        if (val != state) {
                state = val;
                emit result(val);
        }
```

```
}

void BooleanSignalAnd::inputA(bool val)
{
      A = val;
      sendSig();
}

void BooleanSignalAnd::inputB(bool val)
{
      B = val;
      sendSig();
}


CCordLabel::Units CCordLabel::units;

CCordLabel::CCordLabel( QWidget * parent, const char * name = 0, WFlags f = 0
)
      : QLabel(QString("UNKOWN"), parent, name, f)
{
      setAlignment( int( QLabel::AlignCenter ) );
      connect(&CordLabelArbitrator, SIGNAL(updateAll()), SLOT(updateText()));
}

void CCordLabel::setCord(SubCord cord)
{
      cordinate = cord;
      updateText();
}

void CCordLabel::updateText()
{
      if (cordinate.invalid()) {
            setText("UNKOWN");
            return;
      }

      QString tmp, result;

      result.append(tmp.setNum(cordinate.getDeg()));
      result.append(QChar(0x00B0));

      if (units == seconds) {
            result.append(tmp.setNum(cordinate.getIntMin()));
            result.append('\"');
            result.append(tmp.setNum(cordinate.getSec()));
            result.append('\'');
      } else {
            result.append(tmp.setNum(cordinate.getFloatMin()));
            result.append('\"');
      }
      setText(result);
}
```

```
void CCordLabelArbitrator::unitsChanged(int val)
{
      CCordLabel::Units newval = (CCordLabel::Units)val;

      if (newval != CCordLabel::units) {
            CCordLabel::units = newval;
            emit updateAll();
      }
}

CCordLabelArbitrator CordLabelArbitrator;
```

Index

T