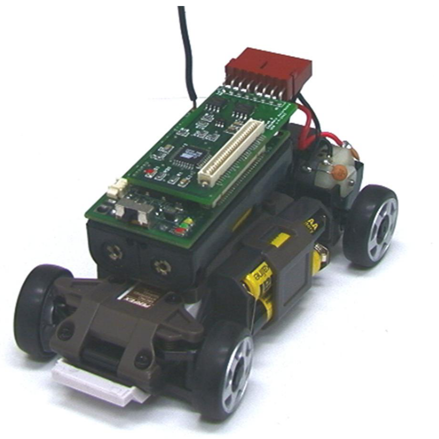
CotsBots Specifications as of 1-27-2002

1. Summary



The goal of the CotsBots project is to use commercial off-the-shelf (COTS) components to build and deploy inexpensive and modular robots. These robots will provide a convenient platform to investigate algorithms, cooperation and distributed sensing in large (> 50) robot networks.

Each robot is small (5”x2.5”x3”) and costs under $200 each (in quantity 50). Each is equipped with on-board processing and wireless radio communication. Currently, sensing boards associated with the TinyOS/NEST project, including a board equipped with light, temperature, buzzer/microphone, 2-axis magnetometer and accelerometer, are available commercially. A common bus allows for the design and addition of robot-oriented sensor boards (for odometry, obstacle avoidance, mapping, etc.). All parts are commercially available and require only minor modifications to build the robot.

This document is intended to provide an overview of the hardware and software that is used in CotsBots. Please see the Getting Started/Troubleshooting document to start using the CotsBots.

2. Processing and Communication

CotsBots are based on the Berkeley Mica Motes (and their derivatives). These motes, originally designed as the atomic components in a large-scale sensor network, are power-efficient, small, and provide computation and wireless radio connectivity. This provides a perfect off-the-shelf base from which to build small, autonomous robots.

2.1. Mica Mote

The Mica mote is the current version of the UC Berkeley mote available. As new motes are developed, it is hoped that they may be interchanged with the Mica mote on the CotsBots.

2.1.1. Processor

The Mica mote currently available from Crossbow Technology Inc. (http://www.xbow.com) is based on an Atmel ATmega128L microcontroller. The ATmega128L is a low-power AVR® 8-bit processor with 128K Bytes flash program memory, 4K Bytes EEPROM, and 4K Bytes internal SRAM. Several methods of serial communication are provided including the SPI interface which is used at the lower levels of the radio interface, I2C (Inter-Integrated Circuit) protocol, and two programmable USARTs. The current draw is approximately 5.5mA while active and < 20uA in sleep mode. Clock speed is set at 4MHz based on an external crystal.

An on-chip 8-channel 10-bit analog-to-digital converter (ADC) is also available (and can be used for differential channels as well). The ADC offers a simple interface for connecting sensors that a user might want to add to the CotsBots platform. In addition, several interrupt lines are also available for sensors.

For more information on the ATmega128L, see the online datasheet at http://www.atmel.com/atmel/acrobat/doc2467.pdf.

2.1.2. Radio

The radio used in the current versions of Mica is the RFM TR1000 operating at 916 MHz. Maximum speed is currently at 40kbps at a power of 0.75mW. Current draw during transmission is approximately 12mA and the radio draws 1.5mA in receive mode. When the radio is put to sleep, it draws less than 1.5uA. Range is approximately 100 feet, but this is highly dependent on the environment and where the mote is placed in that environment. For example, if the mote is placed on the ground, transmission range is much worse.

The other radio option currently in development is the ChipCon CC1000 radio. In testing, this radio has given a much longer transmission distance more reliably and provides multiple channels depending on the frequency selected. See the product web page for details.
http://www.chipcon.com/index.cfm?kat_id=2&subkat_id=12&dok_id=14

Data was found at:
http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA.pdf

2.1.3. Interesting Other Features

The Mica mote provides a 51-pin connector interface from which to connect daughter boards for sensing or actuation (such as the MotorBoard described below). See the MotorBoard schematic also included in the contrib.cotsbots.doc directory for pin definitions on this connector.

The primary debugging interface for the Mica mote if JTAG is not used are the three LEDs (red, green and yellow) provided on the mote interface. When loading an application on the mote, it is often useful to check the included README file to determine what various LEDs indicate.

A separate processor, the Atmel AT90LS2343 is provided as a co-processor to assist in reprogramming the mote over the radio. 4Mbit of EEPROM (external to the processor) are also provided for this use as well as data logging applications.

A voltage regulator is used to stabilize the power supply at 3.3V.

For more information, see the schematic provided at
http://webs.cs.berkeley.edu/tos/hardware/design/ORCAD_FILES/MICA/Mica_sch.pdf

3. Mobility

3.1. Robot Platform

For ease of prototyping and simplicity of hardware, the CotsBots are built from remote-controlled toy cars.

3.1.1. Kyosho® Mini-Z Racer

The Mini-Z Racer is equipped with proportional steering (approximately -26.7^{o} to 26.7^{o}), along with a differential drive and front/rear suspension. The top speed is 12.5 mph (scale top speed is over 250 mph). However, slower speeds are attainable as described later in section 3.2. Forward and reverse directions are both available as well. Clearance on the Mini-Z is approximately ¼", meaning that smooth and very short-hair carpeted floors are acceptable. The Mini-Z can also drive over door jams and small obstacles such as a small notebook lying on the floor. Further information on the Mini-Z Racer can be found at
http://www.kyosho.com/cars/kyod01x1.html

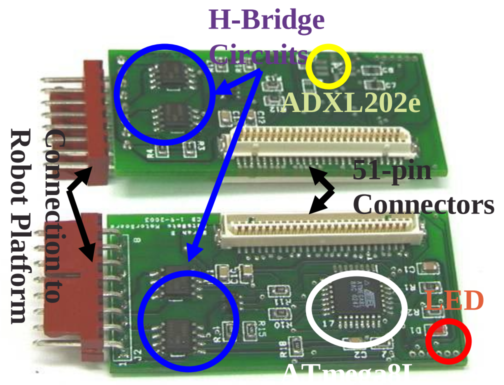
Mini-Z Racers can be ordered from hobby shops all over, but a good place to look would be
http://www.towerhobbies.com

3.1.2. Other Platforms Investigated

Several other platforms were investigated during the original CotsBots design phase. The Tyco Canned Heat RC car is a much cheaper platform ($15-$20) but only uses a solenoid for steering and can therefore not steer as deftly as the Mini-Z. The Canned Heat is available at several retail outlets. In addition, a differential drive platform was found in the Plantraco Desktop Rover. The Desktop Rover uses a tank-tread base, but is slow and rather noisy. It is available on the web for approximately $60 and more information is at Plantraco's website
http://www.plantraco.com/product_dtr.html

Since the MotorBoard described below in section 3.2 can control up to two motor channels, several other RC-car platforms may be adapted to build CotsBots as well. Several very small RC-cars (smaller than 2") have just been put on the market that would make an interesting CotsBots platform.

3.2. MotorBoard



The MotorBoard has been built to provide a means of actuation for the NEST motes. It has been designed with maximum flexibility to maintain compatibility with previous, current and future mote architectures for the NEST project.

The MotorBoard provides an interface to two H-bridge circuits, which can be used to provide speed and direction control for motors, positioning for servos, switching relays or solenoids and a variety of other actuation schemes. Because two H-bridges are provided, the motor board can be used to control two devices from those listed above. The board uses its own microcontroller to communicate with the NEST mote and provide software re-programmability on the motor board itself. Therefore, if new applications arise, appropriate software components may easily be added to the motor board.

3.2.1. Microcontroller

The microcontroller used on the MotorBoard is an ATmega8L available in both a 28-pin DIP design and 32-pin quad flat pack design. The ATmega8L contains 8K of flash program memory, 512 bytes internal EEPROM and 1K RAM. It uses an internal oscillator and consumes only 3.2mA in active mode at 4MHz and 1mA in idle mode. The microcontroller has an onboard 8 channel ADC (6 channels 10-bit accurate and 2 channels 8-bit accurate). There are 2 8-bit timers and 1 16-bit timer with 3 PWM lines available. Data buses include a two-wire serial interface, UART and SPI serial interface. A hardware multiplier is also available. 23 programmable I/O lines provide the flexibility for additional hardware to be added to the motor board.

3.2.2. H-Bridges

An H-Bridge circuit is generally made from 4 MOSFETs (usually two p-type and two n-type) in the configuration of a capital "H" to provide PWM and direction control to the motors. When the PWM is fast enough, the motor averages out the signal to provide an analog voltage for the motor to modify speed. Direction is changed by switching V_{motor} or ground to the opposite terminal of the motor.

The MotorBoard uses four Half-Bridge ICs to create two full H-Bridge circuits. The Half-Bridge ICs used are the Fairchild Semiconductor NDS8858H. These ICs support up to 4.5A at 30V if adequate heat-sinking is provided. For currents and voltages much past those required by standard toy motors, external heat-sinks will be required. For more information on this IC, see the datasheet at
http://www.fairchildsemi.com/ds/ND/NDS8858H.pdf

Several single chip H-Bridge solutions were also considered, but usually did not meet the logic supply constraints imposed by the Mica mote (3.3V and 90mA output).

3.2.3. Power System

Due to the rapid switching to control the speed of the motor, power supplies in systems that use motors are traditionally very noisy. To prevent the motor power supply from severely affecting the power supply for the NEST mote (and possibly resetting it), two separate power supplies are used. The NEST mote power supply (which is a regulated 3.3V in the case of the MICA mote) is used to power the digital circuitry on the motor board, while the power supply from the robot platform used (a Kyosho Mini-Z RC car is the standard platform as in Section 3.1.1) is used to control the motors. Bypass capacitors are used to clean up the motor supply as well.

3.2.4. Servo Capabilities

The Mini-Z robot platform (described in section 3.1.1) uses an Ackerman steering system, the traditional type of steering used by a car. One motor drives the back wheels and a servo sets the angle of the front wheels. The servo on the Mini-Z is simply a motor with a potentiometer on the shaft. Therefore, it is necessary to use readings from the potentiometer on the shaft to feedback position information which can be used to appropriately drive the motor. This information is fed back into the microcontroller through a resistor divider network into an ADC pin on the ATmega8L described above in Section 3.2.1.

It is also necessary to calibrate each servo since the shaft potentiometer is set slightly differently on each car. To do this in the cleanest way possible, a “straight” reference value is stored in the ATmega8L’s EEPROM so that in the future, this might be done automatically. A useful method of automatic calibration would be to use the accelerometer provided on the MotorBoard. By minimizing acceleration on the cross-axis the robot moves straighter. Of course, the servo may also be calibrated by hand using the RobotCmdGUI.

3.2.5. Accelerometer Circuit

An ADXL202e accelerometer (2-axis, +/- 2g) has been provided on the MotorBoard for future use in a low-cost inertial navigation system. Although the ADXL202e was designed to provide a PWM signal as output, the analog outputs are fed into the ATmega8L instead due to software interrupt considerations. A separate reference will be provided later on designing a low-cost inertial navigation system for the CotsBots.

More information on the ADXL202e can be found at
http://www.analog.com/UploadedFiles/Datasheets/567227477ADXL202E_a.pdf

3.2.6. Interface to Mica (or future motes)

The hardware interface to the NEST mote is provided by the 51-pin connector used on current and previous NEST motes. This connector is provided on both the top and bottom of the board to allow for maximum flexibility in sandwiching the two motes.

Lines are provided to connect through several communication channels on the NEST mote. Typically the NEST mote provides UART, I2C (inter-integrated circuit communication) and SPI. The corresponding lines have been connected on the motor board so that any of these methods can be used to communicate with the motor board. The motor board is reprogrammed in the same manner as the NEST mote and can be programmed using the same programming board.

The software used in this interface is described below in Section 0.

3.2.7. Interface to Mini-Z (or other platforms)

The motor board interface to the Mini-Z and other robot platforms will remain backward compatible with the Rene motor board. Wires are extended on the Mini-Z platform and attached to a Molex connector for which there is a compatible Molex header on the motor board. Extra prototyping pins are also provided as seen on the MotorBoard schematic.

The current NEST mote and sensor board requires that the sensor board be facing up for the buzzer and microphone to work properly. This provides difficulty in mounting the NEST stack (including hardware board) since the current MICA mote has a connector on only one side. Therefore, the current mounting procedure when a sensor board is used is to stack the motor board in the middle of the stack. However, mounting holes are provided to better mount the motor board to the Mini-Z platform in the case that only the motor board and NEST mote are used.

More information of this interface may be found in the Getting Started/Troubleshooting guide.

3.2.8. Future Expansion Capabilities

To ensure maximum flexibility with future NEST designs and interesting new sensors, some ADC lines and PW lines are connected to the corresponding pins on the 51-pin connector. Some prototyping area is also provided on the motor board as well (if space available).

4. Sensing

4.1. Odometry

Currently no odometry is available on the CotsBots. Current localization information is either obtained from a surrounding sensor network or from a camera overlooking the robots. However, as stated previously, an ADXL202e accelerometer has been included on the MotorBoard for the purpose of investigating its use in a low-cost inertial navigation unit. Although accelerometer drift will likely render the position estimates invalid in short amounts of time, the current plan is to provide global localization through a sensor network.

The standard means odometry on robots, using a wheel encoder, is not in our plans for development due to the very custom nature of such sensors. However, we would welcome the development of such sensors from a larger CotsBots community. In addition to the INU, plans for an optical mouse-like sensor are being investigated along with camera-like sensors that will be able to determine the relative orientation between the robots.

4.2. Obstacle Detection/Avoidance Sensors

A whisker board (not commercially available) that utilizes simple binary contact whisker sensors for obstacle avoidance has been developed for the CotsBots and schematics are online. This board was originally developed for the Rene Mote version of the CotsBots however, and several new features will likely instigate a re-design of this board. In general, the whisker board consists of a separate microcontroller used to constantly poll up to four binary whisker positions to check if these whiskers are touching a wall or not. An infrared board is also currently in the works to do simple obstacle detection.

4.3. Standard NEST Sensor Boards

All of the standard NEST sensor boards are available from Crossbow Technology (http://www.xbow.com) with the current exception of the Mica Weatherboard.

The Basic Sensorboard consists of a thermistor, photosensor and prototyping space to develop the user’s own sensors.

The Mica Sensorboard provides a thermistor, photosensor, buzzer/microphone pair, ADXL202e accelerometer, and a two-axis magnetometer. Documentation for the Mica Sensorboard is provided in tinyos-1.x/doc.

The Mica Weatherboard includes a thermopile, humidity sensor, light sensor, pressure sensor, and ADXL202e two-axis accelerometer. Documentation for this board is not currently available.

5. Cost

Cost for the CotsBots includes the Kyosho® Mini-Z Racer, MotorBoard, and Mica Mote.

Part	Cost (quantity 50)
Kyosho® Mini-Z Racer	$80.99
Mica Mote	$215
MotorBoard	$37.12
Parts	$14.82
Board	$6.30
Assembly	$16
Total	**$333.11**

It should be noted that using a Dust, Inc. Blue mote instead of the Mica will reduce the cost by $140 to under $200.

6. Software

6.1. TinyOS/NesC

TinyOS is an event-driven operating system designed for sensor network nodes with very limited resources. In an event-driven operating system the software responds to events which can be propagated from the interrupt level to much higher levels in the system. For example, contact made by a whisker sensor can signal a hardware interrupt which can further signal an obstacleDetect event in a higher (abstracted) level. Many robot architectures used to describe behavior are event-driven, making TinyOS an excellent operating system for robot algorithms.

TinyOS offers several other advantages including the ability to abstract software modules. For example, much of the software described below is written so that the application writer doesn't need to know what hardware they are driving and the specifics on how to drive it. Instead, abstracted functionality such as setSpeed, setDir and setTurn can be provided to simplify the software writing process.

The following is a tutorial in which to try out the TinyOS system:
http://today.cs.berkeley.edu/tos/tinyos-1.x/doc/tutorial/

Also included is a reference for the nesC language (a c variant used in TinyOS):
http://today.cs.berkeley.edu/tos/api/nesc/doc/ref.pdf

Software for the CotsBots can be divided into two major sub-sections: software that runs on the NEST mote, and software that runs on the MotorBoard. Version 0.5 of the CotsBots software is described below.

6.2. NEST Mote Software

In the nesC language, the primary means of abstraction is through interfaces. An interface defines commands that may be called and events that may be signaled. A component can implement or use interfaces. Hopefully, this will be demonstrated more fully in the descriptions below.

6.2.1. Interfaces

The primary interface of importance in writing CotsBots code on the NEST mote is Robot.nc.

```
interface Robot {
  command result_t init();
  command result_t setSpeed(uint8_t speed);
  command result_t setDir(uint8_t direction);
  command result_t setTurn(uint8_t turn);
  command result_t setSpeedTurnDirection(uint8_t speed, uint8_t turn, uint8_t dir);
}
```

This interface provides the basic features required to drive the robot from an application. Robot.init() is required to setup the communication components that will be described later in Section 6.4. Robot.setSpeed(speed) takes an argument from 0-255 (where 0 is OFF and 255 is the top speed). A typical, controllable speed lies from 20-100. Robot.setDir(direction) takes a direction argument (FORWARD = 1, REVERSE = 0). Robot.setTurn(turn) takes an argument from 0-60, where 0 is full left and 60 is full right (STRAIGHT = 30).

Because an application writer will often wish to change all three commands at once, a Robot.setSpeedTurnDirection(speed, turn, direction) command is also provided. This is a preferred method over using three separate commands in turn as will be described in the implementation below.

6.2.2. Components

The tos\system\RobotC.nc configuration and tos\system\RobotM.nc implementation provide the Robot interface to the application developer. In this case, RobotC.nc uses whichever means of communication is desired (UART, I2C, etc) to communicate the above commands in the Robot interface to the MotorBoard. More on this communication is described below in Section 6.4.

An important troubleshooting component to note in this implementation is the time required to send a message over the UART/I2C channel. Buffering is provided for two messages, but if the user tries to send three commands simultaneously, it is likely that the last one will be lost.

See the README in tos\system for more information about this and other components.

6.3. MotorBoard Software

The MotorBoard uses an Atmel ATmega8L microcontroller to interface the NEST mote with the motor drivers. To make use of pre-written code in the NEST community, the processor runs TinyOS. Several driver components and interfaces have been written specifically to abstract the MotorBoard functionality and are listed below.

The tos/lib/MotorBoard.h file sets up many of the definitions used in applications and in the messaging system that will be described in Section 6.4.

6.3.1. Interfaces

```
interface HPLMotor {
  command result_t init();
  command result_t setSpeed(uint8_t speed);
  command result_t setDir(uint8_t direction);
  command uint8_t getSpeed();
  command uint8_t getDir();
}
```

The HPLMotor interface provides that basic features with which one can drive a motor. An initialization using HPLMotor.init() is ***required***. HPLMotor.setSpeed(speed) takes an argument from 0-255 (where 0 is OFF and 255 is the top speed). A typical, controllable speed lies from 20-100. HPLMotor.setDir(direction) takes a direction argument (FORWARD = 1, REVERSE = 0). Get functions are also given to retrieve the current speed and direction.

```
interface Servo {
  command result_t init();
  command result_t setTurn(uint8_t turn);
  event result_t debug(uint16_t data);
}
```

The Servo interface provides the features with which to drive a servo motor. Notice that the interface abstracts the implementation details and I could therefore drive a standard hobby servo using this interface or the servo on the Mini-Z Racer. Servo.setTurn(turn) takes an argument from 0-60, where 0 is full left and 60 is full right (STRAIGHT = 30) for the implementation that will be described below in 6.3.2. However, an 8-bit argument is provided so that positions from 0 to 255 might be possible in a different implementation. An initialization using Servo.init() is ***required***.

In addition, an interesting feature is provided in the Servo interface: the debug event. Because it is often necessary to review the control signals given to the servo for calibration, etc. the debug event is provided to push whatever information is desired to a higher level component. This may then be sent back to the radio, or used directly in a higher component.

```
interface EEPROM {
  command result_t init();
  command uint8_t read(uint8_t address);
  command result_t write(uint8_t address, uint8_t data);
  event result_t writeDone();
}
```

This EEPROM interface differs from the EEPROMRead and EEPROMWrite interfaces provided in the standard TinyOS-1.x release. This is done for simplicity in accessing the on-chip EEPROM v. the off-chip EEPROM that those interfaces are designed to handle. EEPROM.read(address) will return the byte read from a particular memory address. EEPROM.write(address) will write a value to this address. EEPROM.init() is ***required***.

An important use for this interface is in the RealMain.nc component. Because the internal RC oscillator on the ATmega8L, it is necessary to calibrate it well enough to send messages over the UART and I2C lines. This calibration value is always stored in address 0 of the EEPROM and is read and stored into the calibration register as the first thing in RealMain.nc.

```
interface ServoCalibration {
  command result_t setKp(uint8_t Kp);
  command result_t setKi(uint8_t Ki);
  command result_t setStraight(uint8_t straight);
  command result_t setDebug(uint8_t state);
}
```

ServoCalibration provides an interface for which to calibrate the Mini-Z Racer servo. ServoCalibration.setStraight(straight) writes a new "straight" reference value to the EEPROM. ServoCalibration.setKp(Kp) and ServoCalibration.setKi(Ki) allow a user to set the gain parameters for the PI control loop controlling the Mini-Z Racer servo. ServoCalibration.setDebug(state) allows the user to decide whether the Servo.debug() event is fired or not.

These functions may be accessed through the RobotCmdGUI.

6.3.2. Components

The *HPLMotor1/2* components implement the HPLMotor interface on two separate channels. Motors are controlled using the PWM lines on the ATmega8L's Timer1. PWM is currently set up at 8-bit, phase correct, TOP mode and the frequency is currently set at approximately 500Hz. This may be slowed down or sped up as applications require. HPLMotor.setSpeed(speed) takes an argument from 0-255 (where 0 is OFF and 255 is the top speed). A typical, controllable speed lies from 20-100. HPLMotor.setDir(direction) takes a direction argument (FORWARD = 1, REVERSE = 0).

MZServo implements both the Servo and ServoCalibration interfaces. MZServo uses a PI control loop (the parameters for which can be set using the ServoCalibration interface) to match the current position read from the ADC to the desired turn radius. Servo.setTurn(turn) takes an argument from 0-60, where 0 is full left and 60 is full right (STRAIGHT = 30).

HPLEEPROM implements the EEPROM interface. For more information on how this is done, please see the ATmega8L datasheet.

The *MotorClock* component implements the standard Clock interface (although not with all of the features provided). The MotorClock is very rudimentary and is run off of Timer0 which is only equipped with an overflow interrupt. Documentation is provided within the MotorClock.nc component itself on achieving different speeds.

MotorTest runs through a set sequence of speeds to test each motor. It implements a StdControl interface and may be started and stopped through the RobotCmdGUI. See the component file for more information.

6.4. MotorBoard Messaging

Because the MotorBoard controls the motor features through the ATmega8L, it is necessary to define a messaging protocol for communication between the NEST mote and MotorBoard.

6.4.1. Message Structure

The message structure used to communicate to and from the motor board is defined as follows:

```
typedef struct MOTOR_Msg {
  uint8_t addr;
  uint8_t type;
  uint8_t data[MOTORDataLength];
} MOTOR_Msg;
```

Each motor board may be assigned a specific address using the TOS_LOCAL_ADDRESS field (or make motor install.*x*). This allows the motor boards to be stacked to achieve more than two channels if required.

The type of message could include anything from accelerometer data to a setSpeed command, etc. Some commonly used commands are defined in the MotorBoard.h file for convienence.

The data array provides arguments for commands or other data corresponding to the message type. The length of this array is variable dependent on the MOTORDataLength field also in MotorBoard.h. For example, if MOTORDataLength = 2 (Default), the data array is two bytes. Two bytes has been enough for everything used thus far, although a variable length packet structure is being considered for a future CotsBots release.

Because the data array can be cast to whatever types and size the programmer wishes, this provides the most flexible means of sending data

6.4.2. Interfaces

```
interface MotorReceiveMsg {
  event MOTOR_MsgPtr receive(MOTOR_MsgPtr m);
}
```

The MotorReceiveMsg interface provides only one event – MotorReceiveMsg.receive(m). This interface is very similar to the standard TinyOS-1.x ReceiveMsg interface, but uses MOTOR_Msg's instead of TOS_Msg's. It is required that a pointer to a MOTOR_Msg buffer be returned on completion of this event.

```
interface MotorSendMsg {
  command result_t send(MOTOR_MsgPtr msg);
  event result_t sendDone(MOTOR_MsgPtr msg, result_t success);
}
```

MotorSendMsg is also analogous to its SendMsg cousin and provides both a MotorSendMsg.send(msg) command and a MotorSendMsg.sendDone(msg, success) event for split-phase operation. Notice that these interfaces are abstract from whichever bus is being used to send and receive messages.

Please see some of the examples below in Section 7 for proper usage of these interfaces.

6.4.3. Components

MotorPacket provides the MotorReceive/SendMsg interfaces as well as a StdControl interface from which the communication subsystems must be initialized. Please see comments in the component code to get a better idea of what this component does.

The UARTMotorPacket component is the configuration component above the MotorPacket module. It sets up the use of the UART instead of another ByteComm interface such as I2C.

6.4.4. UART v. I2C

Unfortunately, as of the initial release, only the UART communication is available. However, an I2CMotorPacket component should be available shortly.

7. Sample Applications

7.1. TestMotorBoard

TestMotorBoard tests the interface to the MotorBoard. It receives and sends messages over the radio interface as well as over the motor board interface. Incoming radio packets are transformed into motor packets. In addition, packets sent from the MotorBoard (which may contain accelerometer or servo debugging information) are relayed back over the radio.

LED Debugging:
RED_LED -- toggles when a message is received from the motor board.
GREEN_LED -- toggles when a message is sent to the motor board.
YELLOW_LED -- can be toggled from a motor board command or externally to test communication.

As hardware drivers and new capabilities are added to the motor board, they should also be added to this component for testing as well as commands/message types in tos/lib/Robot/MotorBoard.h. The current implementation of this component is by no means complete.

To send messages to this application, use the RobotCmdGUI provided in the contrib.cotsbots.tools directory. In the contrib.cotsbots.tools directory (if you haven't already done so), type

```
$ make
... /home/sbergbre/tinyos-1.x/contrib/cotsbots/tools
make[1]: Entering directory `/home/sbergbre/tinyos-1.x/contrib/cotsbots/tools/RobotCmd'
... /home/sbergbre/tinyos-1.x/contrib/cotsbots/tools/RobotCmd
mig java -java-classname=RobotCmd.RobotCmdMsg ../../tos/lib/RobotCmdMsg.h
RobotCmdMsg >RobotCmdMsg.java
mig java -java-classname=RobotCmd.RobotAccelMsg ../../tos/lib/RobotCmdMsg.h
RobotAccelMsg >RobotAccelMsg.java
```

```
mig java -java-classname=RobotCmd.NavigationMsg ../../tos/lib/NavigationMsg.h
NavigationMsg >NavigationMsg.java
javac RobotCmdGUI.java
make[1]: Leaving directory `/home/sbergbre/tinyos-1.x/contrib/cotsbots/tools/RobotCmd'
```

Make sure the SerialForwarder is running, and type:

```
$ java RobotCmd/RobotCmdGUI 74 &
[2] 1656

Administrator@DRAGONFLY /home/sbergbre/tinyos-1.x/contrib/cotsbots/tools
$ RobotCmdGUI: Using group ID 74
```

Try selecting various features to send messages. It should be noted that the Navigate command is not currently functional on the TestMotorBoard program.

7.2. Figure8

Figure8 runs the tos/lib/Figure8M component with the default values. The robot should move in a figure8 pattern, although this depends tremendously on surface, battery voltage, etc. Because there is no position feedback currently being used, this pattern is completely open-loop and must be calibrated. Calibration is possible through tools/java/net/tinyos/cotsbots/RobotCmdGUI and the TestMotorBoard application.

It would be nice if the calibration values obtained above were stored in EEPROM, but currently they are not.

Many features could be added to help close the loop (localization, a battery/speed lookup table, etc.).

7.3. TestNavigation

TestNavigation tests the NavigationC/M component. It accepts a radio message that tells the robot its current x-y position in cm, its current heading in radians (-pi,pi) and the position it should go to. TestNavigation calls the Navigation.navigate command to move the robot from point A to point B. In addition, debugging capability is provided in the Navigation component to check the calculations at various steps. Currently there are no debugging LEDs in this application.

It should be noted, that like Figure8, Navigation is an open-loop component in its current implementation. The speed needs to be set so that it corresponds to approximately 40cm/sec on whatever surface is being used at whatever battery voltage you are currently at.

To use the TestNavigation application, use the RobotCmdGUI as in the TestMotorBoard description above.

Select “Navigate” in the drop-down box, enter the information for navigation, and go!