CSC560 - Real time operating systems
Tanks! - Assignment 5

Mike Krazanowski

August 10, 2010

Contents

List of Tables

List of Figures

0.1 Document Overview

This document is structured to aid in understanding and development of this application. As such each chapter in this document describes a different perspective on the design and implementation.

Section 1 describes the overview and motivation for this application. It doesn't contain any design information other than to orient the user to understand in layman's terms the desired outcome of this design.

Section 2 describes the high-level design of the application outlined in section 1. This section describes the desired goals of the application, describes the high-level states that the tanks and the base station must have implemented, it also describes any reactionary behaviour or PID controls that are needed for this tanks application.

Section 3 describes the hardware components used in this application and a brief description of how it is used and how it is connected to the other components used in this application.

Section 4 describes how the electronics of this application are connected, either through a wire or through wireless connections.

Section 5 describes the communication and timing requirements for this application to succeed.

1 High-level design

The high-level design of this application is to have iRobot devices be able to traverse an environment, to detect the other iRobot devices and to be able to emulate a tank's behaviour (i.e. aim and shoot).

For this to work effectively, the iRobot devices need to be able to take commands from a remote base station, as well as acquire sensor data and report the sensor data's status back to the base station. With this simple communication, the iRobot devices should be able to emulate basic tank and operator behaviour.

The user can control one or many of the "tanks" by moving it around the real world with a video game controller, aim the tank at their opponent and fire. If there is no human controlling the tank, a computer will control it. For a tank to open fire on the other tanks in the arena, it simply needs to point itself at the other tank and fire. In this way the iRobot tank is emulating a real-world fixed-hull mounted tank.

The intent of this application is to create a real-world implementation of the many "tanks" video games that exist such as the Wii-Play: Tanks! game shown in figure 1.

Figure 1: Wii Play! Tanks

1.1 High-level states

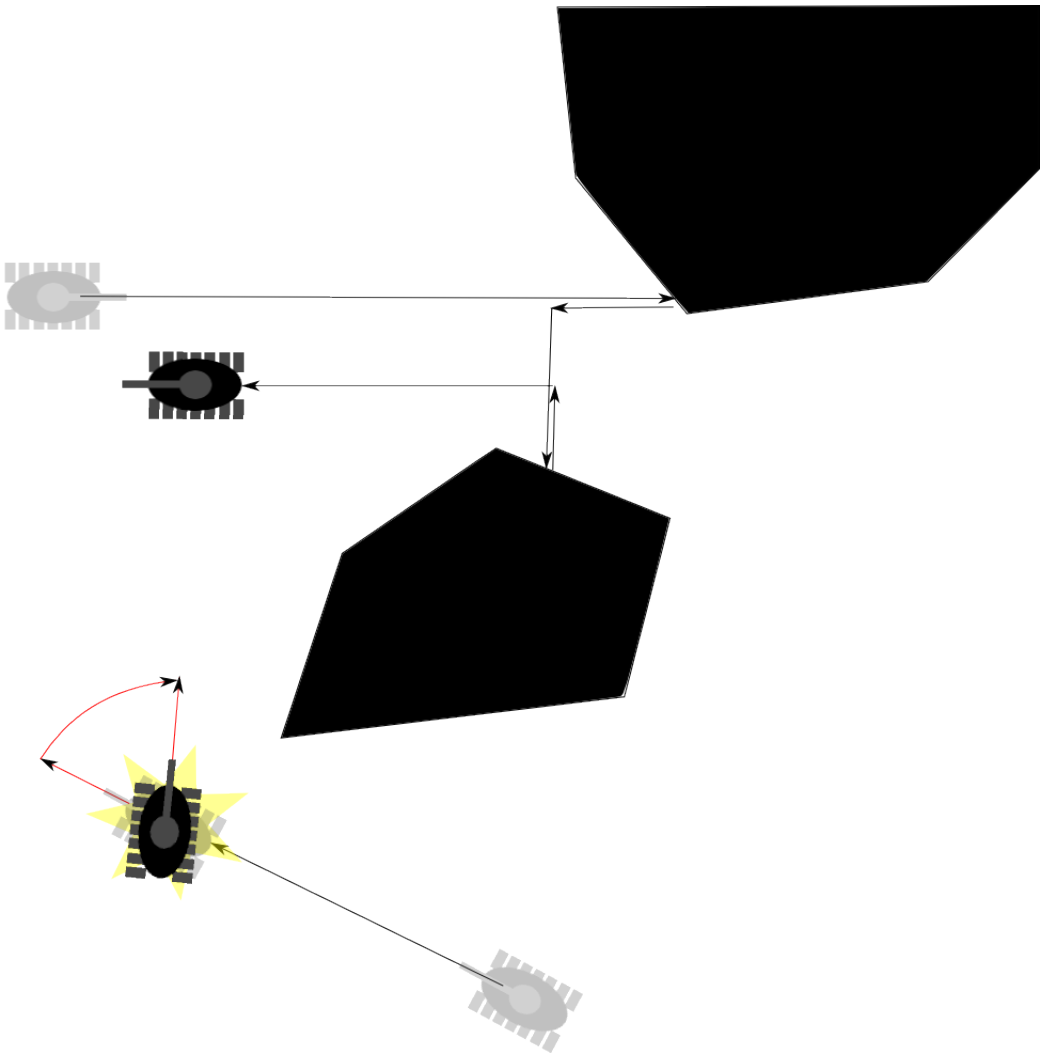


Figure 2: Tank roaming and targetting

The tank should be able to take basic commands and be able to execute them deterministically. The high level commands that will be required are:

1. **Receive and translate command:** Accept and parse a command sent from the base station.
2. **Roam:** Generate a sequence of move elements that will allow the tank to move from one point in space to another.
3. **Detecting Enemy:** Determine the relative direction of the enemy tank.
4. **Targeting Enemy:** Stop tank's translation movement and change this tank's orientation to point at and fire upon the other tank.
5. **Report Status:** Collect and report the status of the tank and its view of the world back to the base station.
6. **Plan:** In the situation where the tank has not received a command from the base station for a while, the tank itself will attempt to decide the best course of action.

7. **Dead:** The tank has been damaged and can no longer continue playing the game.

8. **Restart:** The tanks should restart playing the game.

1.2 Actions

For the desired goals to be carried out, there needs to be a set of actions defined to assist the states defined previously. The following list of actions will be used to assist with this end. The following list of actions is not a complete list of functions used in this application, but is a good coverage of the interesting actions used, and are referenced in the rest of this document.

1. **ProcessUpdateTankStatus:** Receive and buffer the roam status, determine the threat level, determine the direction of the other tank, determine the status of engagement for this tank, collect any additional necessary iRobot status data.

2. **ProcessRoamAndSeekEnemy:** Define commands to execute on the iRobot: Stop, ArcLeft, TurnLeft, ArcRight, TurnRight, Forward, Backup, ArcBackLeft, ArcBackRight.

3. **ProcessDetectingEnemy:** Determine the relative direction of the other tank.

4. **ProcessTargettingEnemy:** Rotate the tank (TurnLeft, TurnRight) until this tank is pointing at the enemy tank.

5. **ProcessEngagingEnemy:** Start the firing sequence (it is be beneficial to add a delay to firing to allow for the other tank to have a fighting chance to escape. Because of this, the firing sequence needs to charge up before firing.).

6. **ProcessTranslateCommand:** Receive and store the sequence of move commands for the tank from the base station.

7. **ProcessReportTankStatus:** Send the status information of the tank to the base station over the wireless communication link.

8. **ProcessPlan:** Calculate the desired course of action based on the current tank status (and any historical data stored).

9. **OutputCommandToRoomba:** Output the commands to the UART in a format that the Roomba can understand.

10. **GetNextCommandFromRoamBuffer:** Get the next command from the command buffer.

11. **GetRelativeEnemyDirection:** Calculate a weighted average of a small window of recent IR samples taking into account the servo angle and the compass direction (n, s, e, w).

12. **GetTurnDirectionFromRelativeDirection:** Use the relative enemy direction to calculate which direction the tank needs to turn to point at the enemy.

13. **IsEnemyTargetted:** Determine if the tank is angled close enough to be able to consider the enemy as targeted.

14. **AIRandomRoam:** Gets a random tank command to act as a stub for a more featureful AI process.

15. **InitializeRoombaSongs:** Define the MIDI songs used to identify the various states in this application. For example, the tanks play the Jaws theme song on start up, there are 3 firing level songs that are played while firing, there is a random sounds song to identify when the tank has fired and finally the tank that has just died plays a wa-wa-wa song upon death.

1.3 High-level state machine

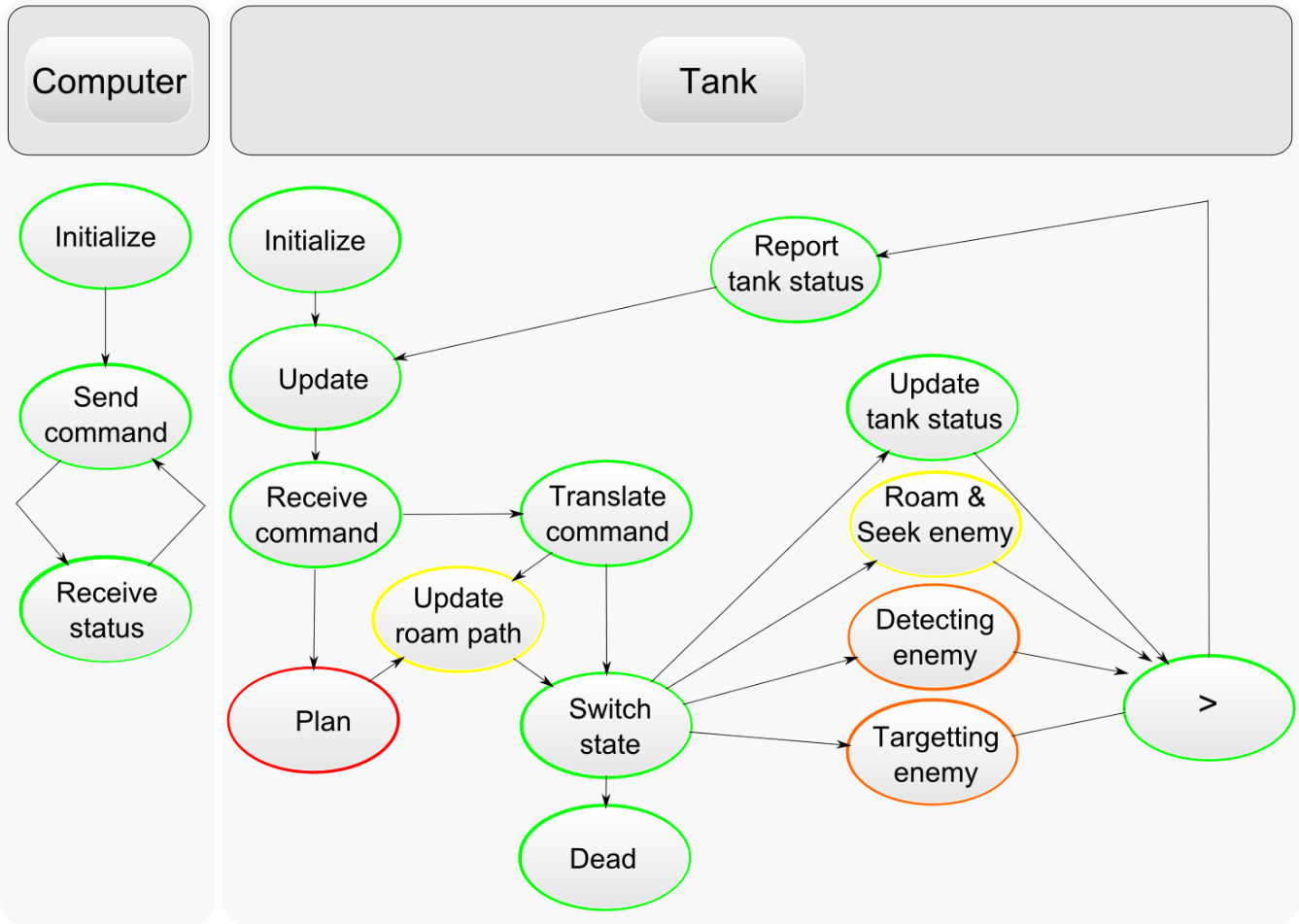


Figure 3: State Machine for Tanks application

2 3-Layered design

This application fits the behavioural robotics design paradigm. All sensor acquirement and PID controls fall into the react layer. The roam command sequencing as well as the targeting enemy behaviour represents the execute layer. The plan state, including complex seeking and evading behaviours, fall nicely into the plan and think layer of the 3-layered design methodology.

2.1 Basic tank control and reactions

2.1.1 Update Tank Status

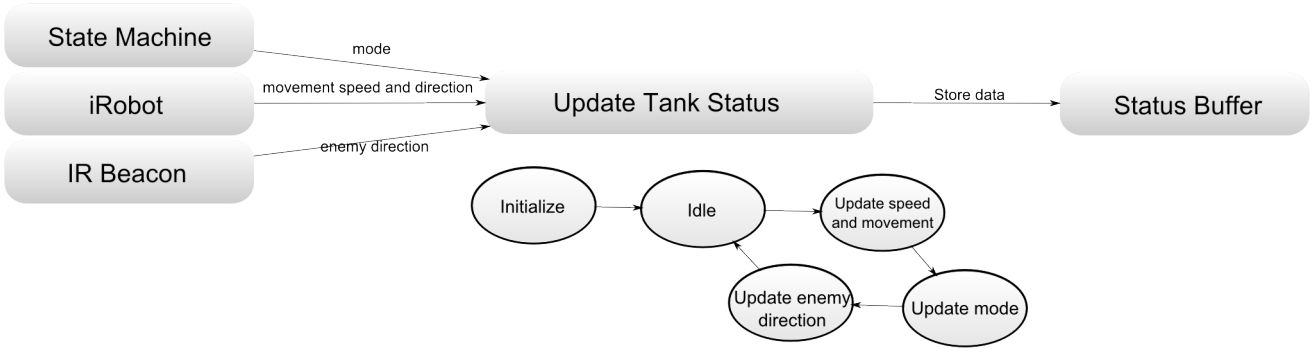


Figure 4: Behaviour for updating the status of the tank

The *update tank status* state describes how the status of the tank will be collected. Its intention is simply to probe all the devices that can return state information that is useful for this tank application and to store the values in a status buffer.

2.1.2 Report Tank Status

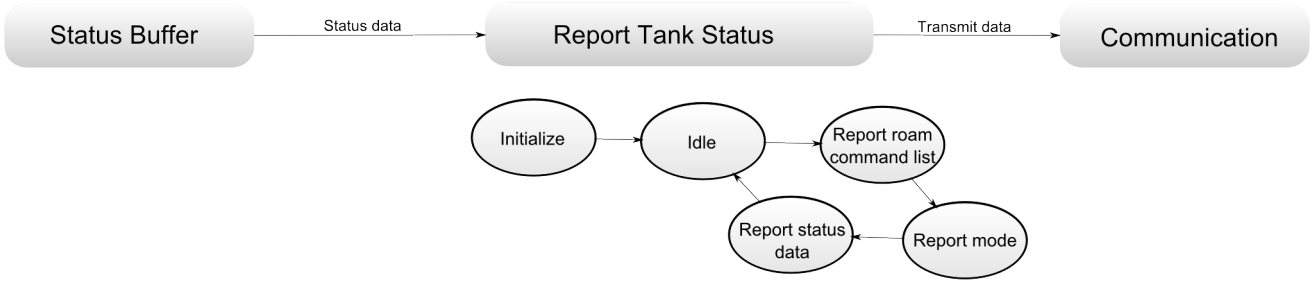


Figure 5: Behaviour for reporting the status of the tank

The *report tank status* state describes how the tank will report the relevant status of itself back to the base station.

2.1.3 Roam and Seek Enemy

The *roam and seek enemy* state describes how the tank will receive and execute movement commands. For the most part this state is simply receiving commands from the roaming buffer and executing them. There is some definition of how to deal with collisions if they arise when a human

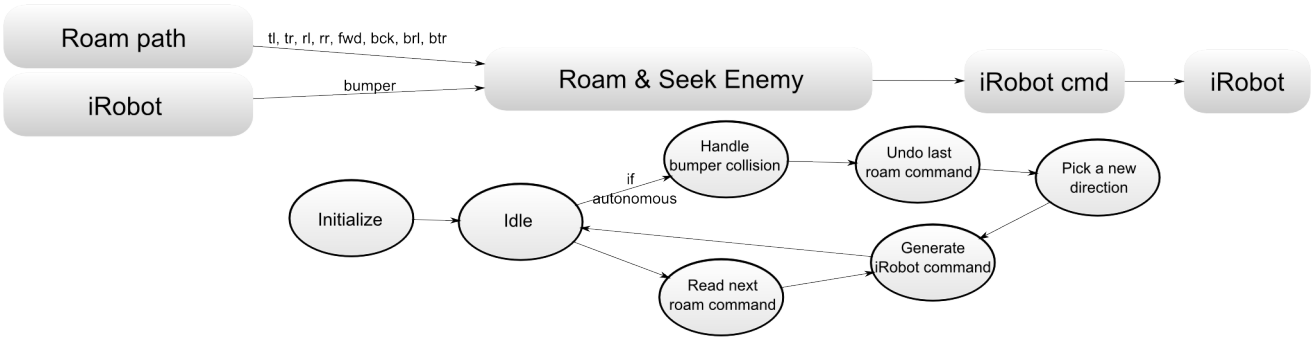


Figure 6: Behaviour for roaming and seeking enemy tanks

is not controlling the tank.

2.1.4 Detecting Enemy

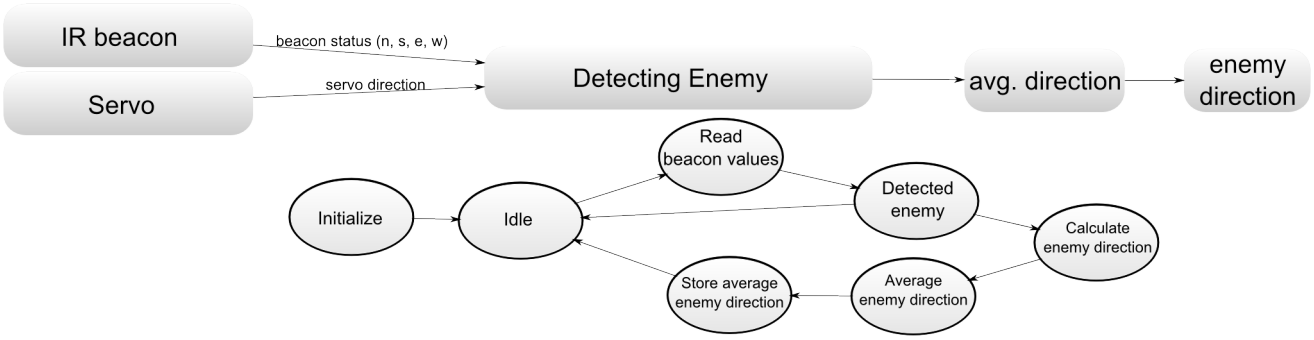


Figure 7: Behaviour for detection of enemy tanks

The *detecting enemy* state describes how the tank should be able to detect another tank.

2.1.5 Dead

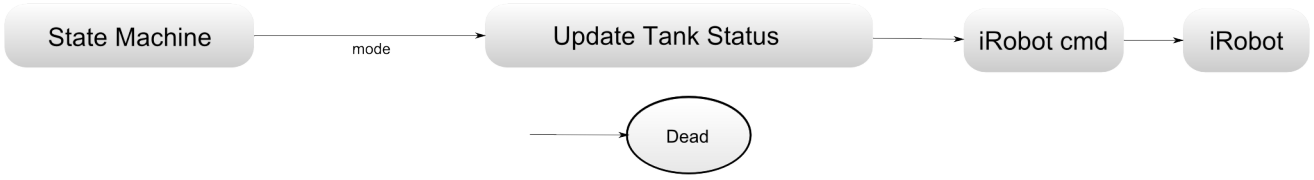


Figure 8: Behaviour for a dead tank

The *dead* state simply disallows all other states to execute. It is the dead-end state.

2.2 Controlled behaviour elements

2.2.1 Targeting Enemy

The *targeting enemy* state describes how to engage another tank. The intended behaviour of the tank is to stop (translational) movement, rotate the tank to point at the enemy, then fire upon the

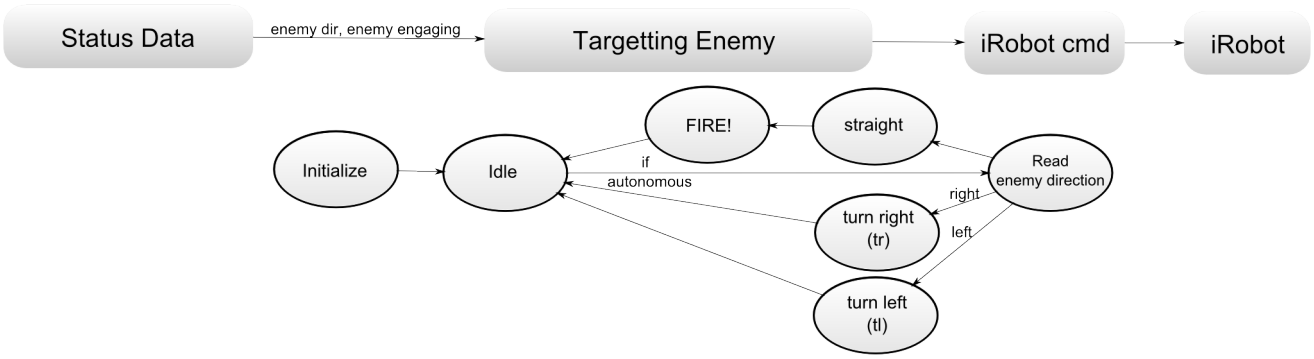


Figure 9: Behaviour for autonomous targeting of enemy tanks

enemy tank.

2.2.2 Plan

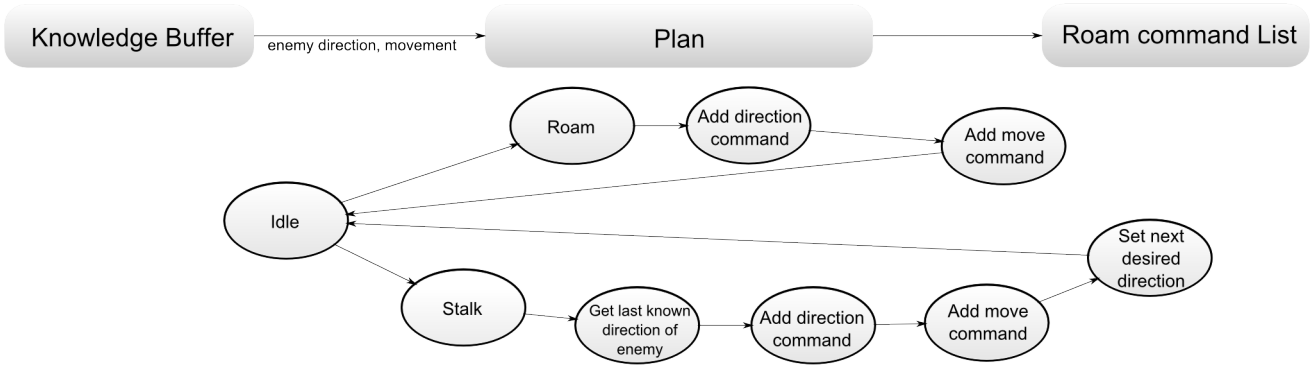


Figure 10: Behaviour for autonomous planning of tank actions

The *plan* state describes how to autonomously determine the next course of action if the base station is not sending commands.

The intended behaviour of this state is to generate a small set of actions that allow for the tank to get within range of being able to fire upon the enemy. As such, this state simply tries to roam the environment searching for the enemy or tries to stalk the enemy by seeking out the last known position of the enemy tank. (Actually simply tries to move in the direction of the last known sighting of the enemy tank).

3 Hardware

The equipment used for this application is as follows:

3.1 PC

A computer is used to provide an interface between the video game controllers and the tanks (transitively through the base station). It could also be used to perform any AI required for uncontrolled tanks in the field.

The computer connects to the Wii remote (wiimote) through a bluetooth connection. Once the Wiimote is paired with the computer, it can be connected to the base station using its FTDI FT232RL connection.

Figure 11: Computer

I implemented a C# program to bridge the connected Wiimote to the RS232 connection on the Arduino board (as shown in figure 12). This program uses the coding4fun library to connect to the first available Wiimote connected to the computer and sends the state of the controller to the specified COM port. Doing this will send the state of the Wiimote for use on the Arduino micro controller.

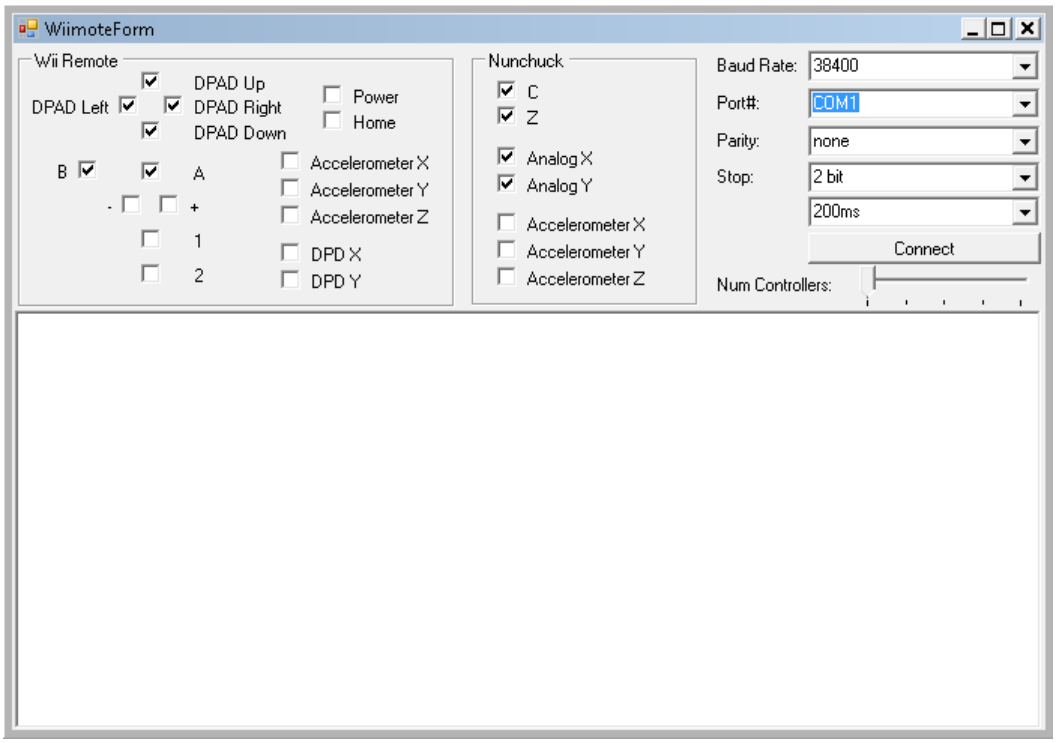


Figure 12: WiimoteForm

3.2 iRobot

The iRobot devices are used to simulate the mechanical parts of the tanks allowing for movement in the world as well as carrying the necessary sensors, electronics and microcontrollers. The iRobot has 2 motors for movement, a bumper for detecting movement collisions, a speaker for playing MIDI songs, several other sensors and brushes (that aren't used for this application) and a Mini-DIN port for serial communication.

For the purposes of this application, the AVR2560 will be connected (through the STK600) with the iRobot through the Mini-DIN connection using the Roomba's Serial Command Interface (SCI). (More details on the iRobot Roomba SCI specification can be found at www.irobot.com).

Figure 13: iRobot Roomba Discovery

To interface with the Roomba, the user simply needs to remove the Mini-DIN port cover (located in figure 13 at roughly the 4:00 position) and plug a Mini-DIN plug into it. It is this connection that is used to provide power to the STK600 and the Pololu IR sensors. The Mini-DIN connection also is used to communicate between the Roomba and the STK600.

For the purposes of this application, the Mini-DIN cable has 5 wires that are potentially used, a power wire, a ground wire, a TX and a RX wire and a device detect line.

The power (Vpwr) line produces roughly 16v of unregulated power which needed to be reduced and regulated to a 9-15v for the STK600. The power is regulated using the lm7805c chip wired with two 1k resistors as is shown in figure 21 to produce roughly 13v. The pololu uses a 6v-16v supply as well, so the output of the lm7805c is used to power it as well.

The RX/TX lines already use to 5v.

3.3 ATMega2560

This is the microcontroller that will is used to control the iRobots and acquire data from the sensors and to interface to the wireless electronics on the tanks. The ATMega2560 is mounted on a daughter

board and attached to the STK600 interface board.

On this ATMega2560 is loaded the RTOS that I implemented in assignment 3, and it is within this RTOS that the tank's communication and behaviour code is implemented. The code makes heavy use of the periodic process plan (PPP) to schedule and coordinate the various behaviours defined in section 2 of this document.

Since the original design didn't specify (intentionally) how those behaviours are to be implemented, some of them are defined as stand-alone processes whereas some are defined as substates within one higher level process. For example most of the communication "behaviours" that are defined for this application are implemented in separate processes, whereas most of the movement and AI behaviours are all defined as procedures and executed within a single "update" process.

Figure 14: ATMega2560 and STK600

The STK600 provides MANY IO pins to allow for interfacing to the ATMega2560 chip. For the purposes of this document, the IO pins defined in the code, in the wiring diagram (fig. 21) and elsewhere correspond to the pin configuration shown in figure 15.

A quick overview of the IO interfacing is as follows:

1. The **Roomba** is interfaced through the UART RX/TX pins on the STK600.

2. The **Servo** is controlled through the pe4 pin on the STK600 using the pwm functionality on the ATMega2560 to control the duty cycle for a 20ms period and a varying signal from 1ms to 2ms. The servo is powered through one of the many VTG and GND pins on the STK600.

3. The **Pololu IR beacon** sends the directionality signal (north, south, east, west) to the STK600 through the pj0-3 pins respectively. The pololu is powered with the regulated power source also used for the STK600 itself.

4. The **Radio** is connected through various pins on the STK, for the most part the pb1-5 pins and the pd0 pin as shown in the wiring diagram (fig. 21). The radio is powered through one of the many VTG and GND pins on the STK600.

3.4 Arduino, Seeduino

Due to the short supply of STK600 interface boards, an Arduino mega micro controller was used in place of the 3rd ATMega2560 for the base station. This device is used to manage the game state, communicate with the tanks over the radio connection, manage the battlefield status and to perform any AI that may be needed. It is also used to capture the Wiimote state values through the UART channel, transform those values to tank commands and send the commands to the tanks over the

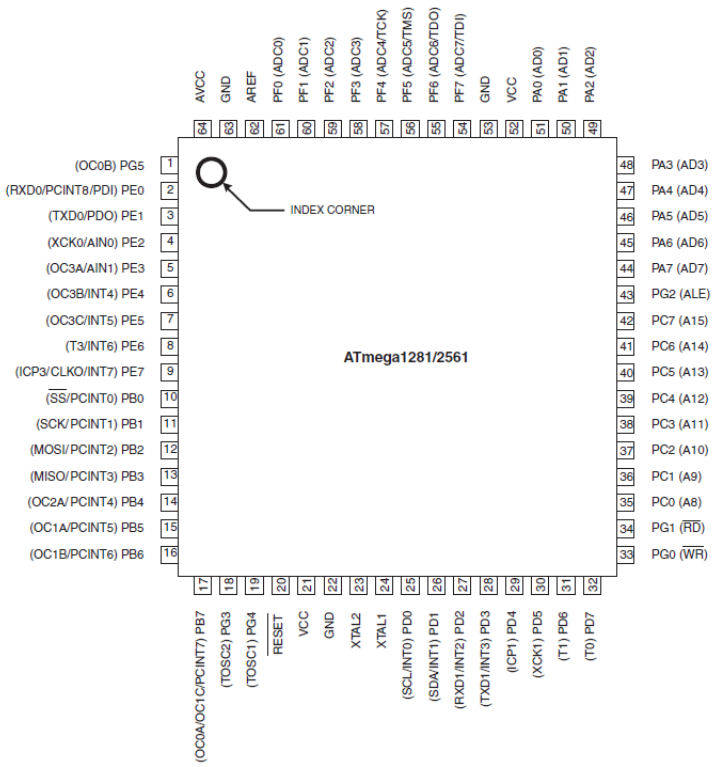


Figure 15: ATMega256X/128X IO Pin Diagram

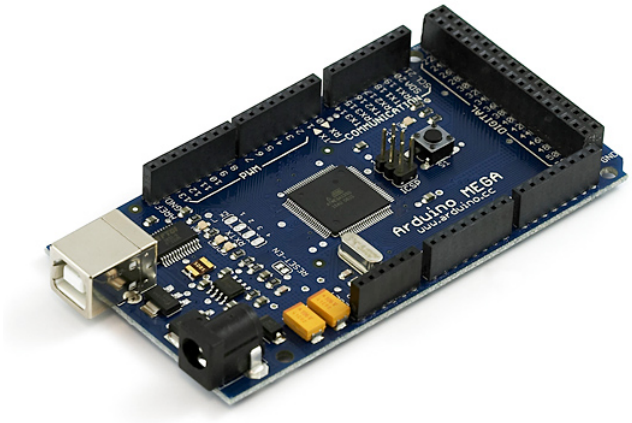


Figure 16: Arduino Mega

radio.

From an IO perspective, the Arduino and Seeduino have different labeling of the ports, numbered from 0 to 53 mainly. Since these micro controllers have an ATMega1280 chip on it, the pins map to the same IO pins described in the pins shown in figure 15. The actual pin mapping is described in the Arduino-mega-schematic.pdf.

Code for the basestation was written using the RTOS I implemented and compiled using AVRStudio (like the Tanks) but I could not get it to upload to the Arduino and Seeduino boards through AVR studio. I was able to create a batch file using the avrdude utility and the following

command line.

```
avrdude.exe -cstk500v1 -P//./COM4 -b57600 -pm1280 -Uflash:w:BaseStation\default\
   BaseStation.hex
```

As an aside, the AI code proved to be easier to develop on the Tank side of the radio communication channel due to the more readily available information available to the tank itself. It should be stated that it doesn't matter where the AI is performed, it could easily be sent from the base station instead of being managed by the tank itself, but due to production time line concerns, it was simply easier to implement on the tank's side.

3.5 nRF24L01 Wireless

The wireless device to allow communication between the tanks and the base station. This device communicates on a 2.4GHz ISM band and can communicate up to 2Mbps. These devices were used to transfer commands from the base station to the tanks. These devices were also used to transfer tank status information from the tanks back to the base station.

I used the drivers provided that were written by students from previous iterations of this class as the interface to the NRF24L01.

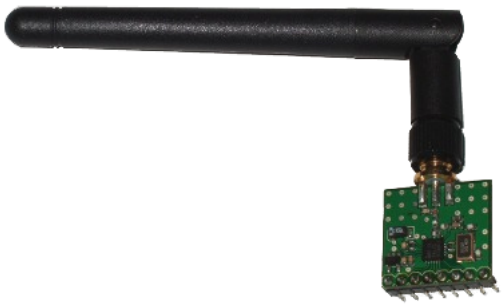


Figure 17: NRF24L01 Radio

3.6 Pololu IRB02 IR beacon

The Pololu IR beacons are infrared emitters and sensors that allows two such devices to detect each other. They are mounted with sensors that detect which direction (north, south, east, west) the other Pololu beacon device is relatively. The beacon will pull low one of the direction lines for the direction that had the highest IR response from the other beacon.

I mounted the device upside down to allow for clearance between the capacitor sensors and IO ports on the Pololu. Also because of the positioning of the capacitor on the Pololu device, I was required to point the east direction in the tank's forward direction. This makes the south point to

the left and the north point to the right.

I mounted one of these devices on a servo motor and swept it across a 90 degree span with reasonably high frequency. I added the angular direction the Servo had moved the beacon in to get better precision for the direction of the other device.

As a signal switched from one of the compass directions to another, it was possible to add the relative 90 degrees to the incoming signal to collect the relative forward direction.

Taking all of the factors above into account I used a weighted average of a small window of historical direction values from the Pololu device to determine with higher precision which direction the other device is in (relatively). By averaging the signal values with added compass offsets and adding the servo angle, it was possible to reduce the detected direction of the other IR beacon to less than 30 degrees with reasonable accuracy.

Unfortunately having such precise detection of the other tank made for difficult targeting for the AI tank. It was way too easy for a human to evade the AI tank by simply moving out of the 30 degree range. In the end I bumped the angle of detection up to roughly 80 degrees, but game play tuning could benefit from more experimentation.

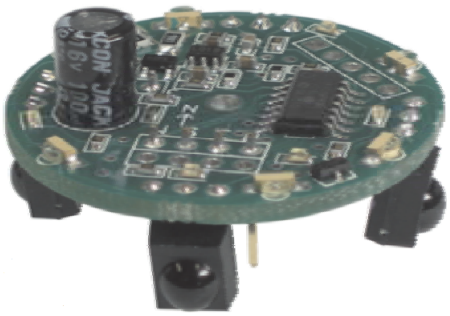


Figure 18: Pololu

3.7 HS-55 Sub-Micro Servos

Used to these micro servos to control the direction of the IR beacons device for each tank and were swept some percentage of the -45 to 45 degree range. As described in the Pololu section, the percentage that was finally decided upon was from 10% to 90% resulting in a sweep from -36 to 36 degrees.

As in previous assignments, the pwm signal needed to control the servos were set using the output capture unit (OCU) on the ATMega2560 through the pe4 IO pin. The servo itself was also powered using one of the pwr and gnd pins.

The Pololu IR beacon device was mounted on the servo and its direction was defined by the position of the servo.

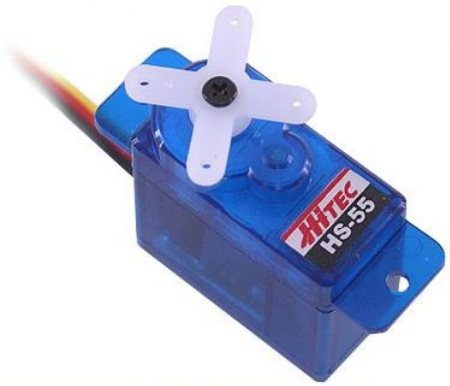


Figure 19: HS-55 Sub-micro servo

3.8 Wii remote

Finally the Wii remote (Wiimote) was used to allow a human to control one of the tanks. The data was acquired from the Wiimote on the computer (through a Bluetooth connection) and sent to the base station using a serial connection. I.E. A COM port on the computer's side, and a UART connection on the base station's side.

Figure 20: Wiimote

4 Wiring Diagram

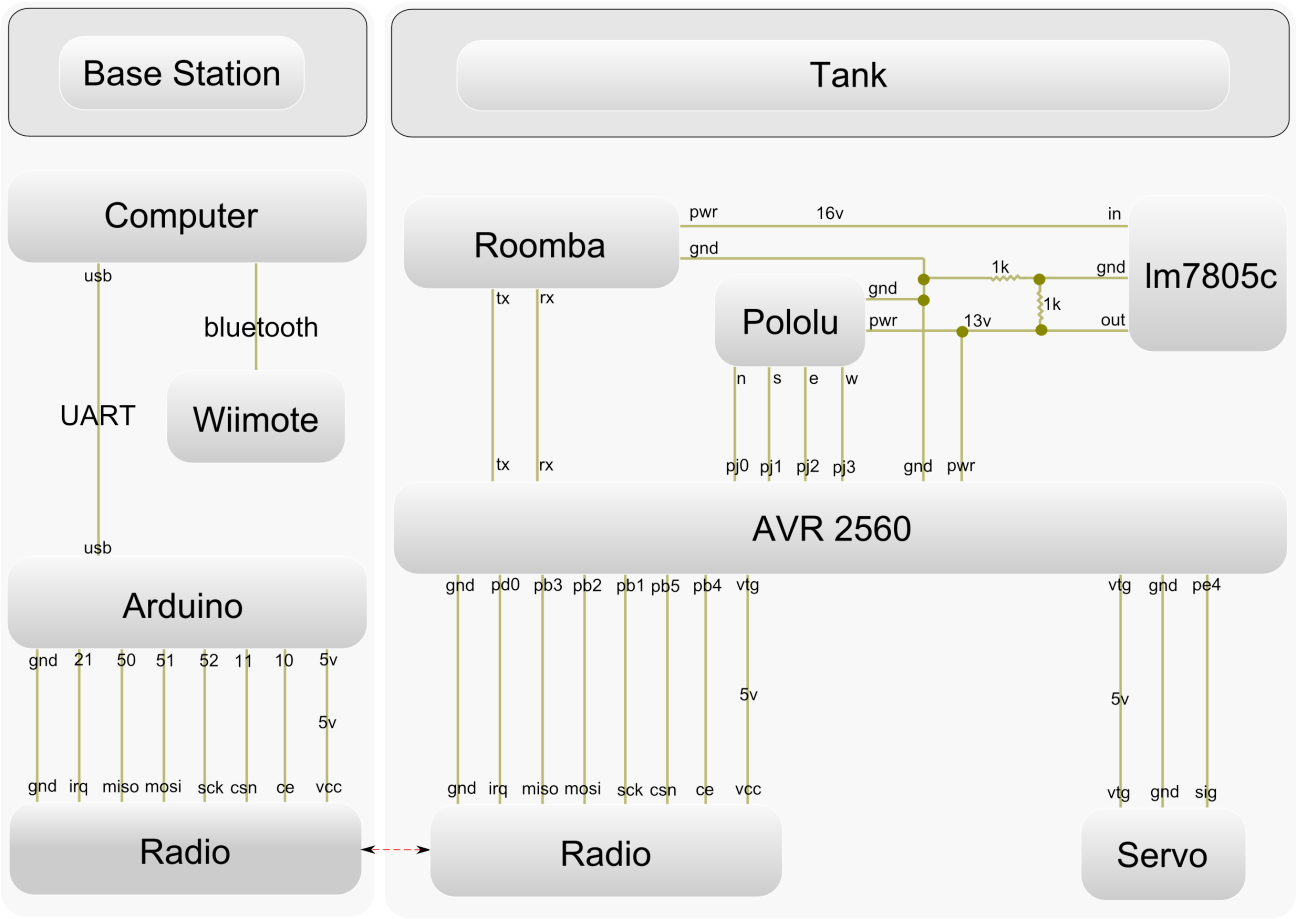
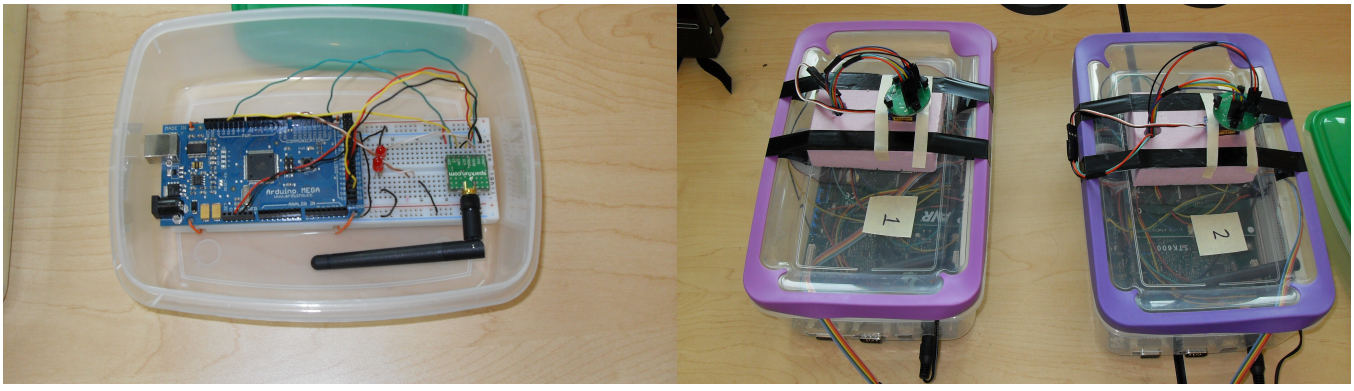


Figure 21: Wiring diagram



(a) The base station (b) The tanks

Figure 22: Hardware pictures

Figure 23: The tank in action

5 Communication and Timing

This section describes the communication of the tanks application discussed in the rest of this document. For the purposes of this document, the majority of the communication design relates to how the tank communicates with the base station.

5.1 Communication Overview

For the tanks application the communication of interest occurs between the tanks and the base station. To this end, this chapter describes the necessary communication and timing requirements that are necessary to allow the tanks application execute as expected.

An additional requirement is to ensure that the communication between the base station and the micro controller that is handling the communication is sufficient.

5.2 Communication Data

First and foremost we need to define the set of data that is to be passed between the various devices. At a high level, the base station sends sequences of commands and battlefield status information to each of the tanks separately. The tanks return to the base station any status information that is necessary for the base station to know about as well as requests for more information.

5.2.1 Tank Commands

The following section describes the set of commands the base station sends to each tank on the battlefield. For clarity sake, the following commands are intended as private messages to a particular tank object.

```
typedef enum eTankCommands
{
        /* Not received */
        COMMAND_UNDEFINED = 0,

        /* Movement commands (Idle) */
        COMMAND_MOVE_STILL,

        /* Movement commands (Roaming) */
        COMMAND_MOVE_FORWARD,
        COMMAND_MOVE_BACKWARD,
        COMMAND_MOVE_TURNLEFT,
        COMMAND_MOVE_TURNRIGHT,
        COMMAND_MOVE_ROTATELEFT,
        COMMAND_MOVE_ROTATERIGHT,
        COMMAND_MOVE_BACKWARDSTURNLEFT,
        COMMAND_MOVE_BACKWARDSTURNRIGHT,
```

```
        /* Attack commands (Targetting) */
        COMMAND_AIM,
        COMMAND_FIRE,
        COMMAND_DIE,

        /* Retreat command (Evading) */
        COMMAND_EVADE,

        /* System leve commands */
        COMMAND_RESTART,
        COMMAND_OFF,
}
TankCommandEnumeration;
```

An observation of this list shows that some of these commands are simple atomic instructions to tell the tank how to move around the world, whereas the rest imply tank-state information.

Commands like COMMAND_AIM, COMMAND_FIRE, COMMAND_DIE and COMMAND_EVADE tell the tank which state it should be in, but not necessarily what to do in that state. These commands map more closely with the state definitions, which are simply abstractions of controls for the tank and have a more complex mapping to controlling the tank.

The COMMAND_MOVE_X commands tell the tank exactly what to execute and have a 1:1 translation to the state of the wheel motors.

The COMMAND_RESTART and COMMAND_OFF are more game play abstractions that are not necessary for the simulation of a tank, but provide more convenience for starting and stopping a game play sequence.

It should be noted that the source of these commands may come directly from user input (via a video game controller), may come from some sort of AI system controlling an unmanned tank or may be simply a broadcast of critical game state information. From the perspective of the tank and its communication method, it doesn't matter what the source of the commands are. This is critical to allow for flexible abstraction between user controlled tanks, AI controlled tanks and possibly a hybrid of both.

5.2.2 Tank Status

The Tank status is reported back to the base station at the end of the update loop. As is shown in the structure below, there are 4 data elements that comprise this status package, the mode that the tank is in, the command that was executed in the update loop, the direction that the enemy tank was detected in and a set of flags for general state.

```
typedef enum eTankModes
{
        TANKMODE_IDLE,
```

```
        TANKMODE_ROAMING,
        TANKMODE_TARGETTING,
        TANKMODE_ENGAGING,
        TANKMODE_FIRING,
        TANKMODE_EVADING,
        TANKMODE_DEAD,
}
TankModes;

typedef enum eEnemyDirection
{
        ENEMYDIRECTION_UNDEFINED = 0,
        ENEMYDIRECTION_RIGHT,
        ENEMYDIRECTION_LEFT,
        ENEMYDIRECTION_STRAIGHT
}
EnemyDirection;

typedef struct sTankReportStruct
{
        uint8_t Mode;
        uint8_t CommandExecuted;
        uint8_t EnemyDirection;
        uint8_t Flags;
}
TankStatusReportStruct;
```

5.3 Wireless usage

With a quick analysis of the amount of data to be sent across the wireless channels, it is relatively easy to see that this application doesn't even come close to filling the allowed 2Mbps bit rate that the nRF24L01 can handle.

Each Tank needs to send/receive 3 different communications per update, with 4 updates occuring (roughly) per second. With a request packet (1 byte), a command packet (3 bytes) and a status packet (4 bytes) the total amount of data per update loop is 8 bytes. This results in 32 bytes per second, which is easily within the 2Mbps limit, even when taking into account the overhead of device identification and other sources of overhead for sending a packet wirelessly.

5.4 Timing diagram

This section describes the timing requirements of the tank application from the perspective of the tank. The following describes the expected timing behaviour of the communication and execution of the previously defined behaviours.

Since the timing requirements for this application is heavily periodic, the tank code makes heavy use of the PPP in the RTOS. The following is the actual PPP used for the tank code.

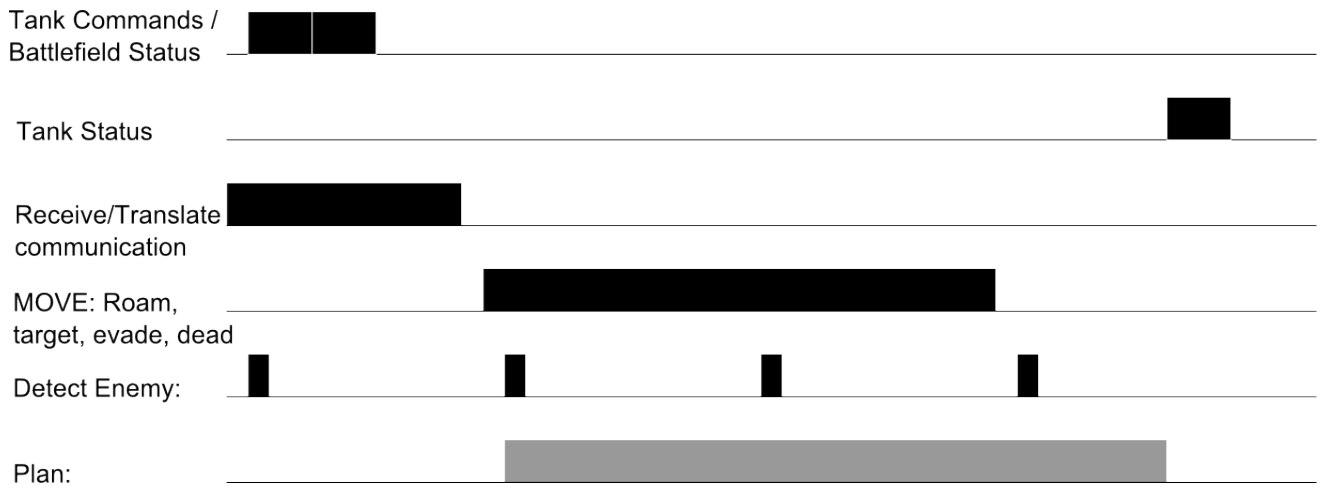


Figure 24: The high-level timing diagram for the tanks and their communication with the base station

```
enum { Request=1, Receive, Update, Report, UpdateIR };

const unsigned int PT = 8;
const unsigned char PPP[] = {
         UpdateIR, 1,
         Request, 2,
         Receive, 3,
         UpdateIR, 1,
         Update, 5,
         UpdateIR, 6,
         UpdateIR, 1,
         Report, 5
};
```

Where *Request* corresponds to the tank requesting a command update, *Receive* corresponds to accepting the response from the base station, *Update* does all of the behavioural code execution, *Report* corresponds to the tank's reporting back to the base station and *UpdateIR* corresponds to the servo sweep update and acquisition of the IR beacon status.

The response time for communication to the tank is not critical for the most part. Since many of the commands will result in activating motors or other form of physical response, it is not necessarily possible to get high responsiveness anyways. As such, the period of this communicate/execute loop can be as long as 200-500ms without being considered a failure.

This application is going to aim for a 250ms loop period, which should give adequate responsiveness while allowing for a reasonable likelihood of meeting all the necessary timing requirements.

5.4.1 Tank Command Communication

All tank commands require at most 24 bits to be communicated (per tank). There is an 8-bit command component as well as an 8-bit duration parameter and an identifier to specify this packet

as a command represented as an 8-bit quantity.

5.4.2 Tank Status Communication

All tank status packets report 4 bytes of data (as shown in the Tank Status section).

5.5 Message sequencing chart

This section describes the simple protocol for the tank to communicate with the base station.

First of all the tank will notify the base station that it is ready for an update by sending a "REQUEST_COMMAND" message. The base station will respond with acknowledgement of reciept of the request with the set of commands from the base station. After 20ms the tank will start processing the response from the base station.

The tank has reserved 50ms to execute the update process. All of the behaviours described in this document needs to complete executing within the 50ms time slice. After the update process is complete the tank status report could occur at any time. As the code is currently set up, the PPP waits a large period of time to allow for a reasonable amount of time to elapse to allow for the wheel motors to be able to execute.

The delay is tuned to match the 250ms period as possible, set to a 240ms period.

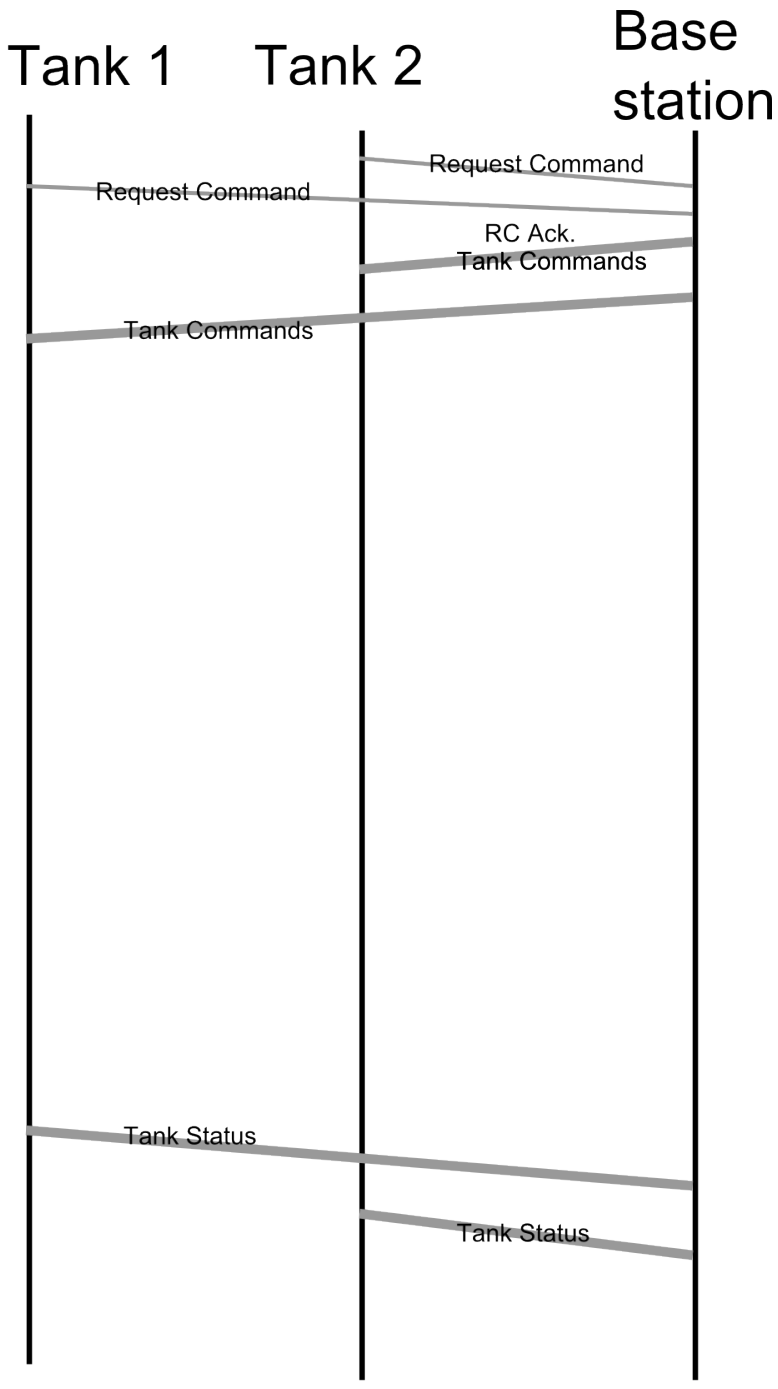


Figure 25: The message sequencing chart