CENG 499 Technical Project:
Design and Implementation of an Autonomous Mobile Robot

Submitted to:
Dr. Kin Li
August 7, 2002

Joel Fieber       98-09286
Anthony Bergelt   98-08462
Abstract

A mobile robot operates in an unpredictable environment. To construct a controlling system for a mobile robot presents an in-depth study of the challenges associated with creating concurrent, real-time systems. This document is a detailed report on the design and implementation of a mobile robot.

The robot was intended to autonomously navigate a grid—while avoiding a competing robot on the grid—to locate and retrieve objects. This specified objective of the robot was clearly defined, but the requirements given for our system provided considerable freedom in our design. While the physical chassis for the robot was provided, our project required developing the entire real-time operating system (RTOS), the drivers for the associated sensors and actuators, and the application software to give the robot meaningful behaviour. We attempt to outline not only the problem and our solution, but also the rationale and insights behind our solution and the final outcome of all our design decisions. The intent is to illustrate the fundamental issues of a non-trivial embedded application. It is expected the reader has a basic knowledge of real-time systems and design formalisms such as finite state machines (FSM).
# Table of Contents

ABSTRACT

LIST OF FIGURES

1.0 INTRODUCTION

2.0 PROBLEM SPECIFICATION

2.1 HARDWARE

2.2 SOFTWARE

3.0 PROBLEM SOLUTION

3.1 COMMUNICATION

3.1.1 Goals

3.1.2 Events

3.2 HIGH LEVEL CONTROL

3.2.1 Finding Coins on the Grid

3.2.2 Taking Coins Home

3.2.3 Getting Home when Lost

3.3 LOW-LEVEL CONTROL

3.3.1 Go Forward to Next Intersection

3.3.2 Turn Left

3.3.3 Turn Right

3.3.4 Drop Coin Left

3.3.5 Drop Coin Right

3.3.6 Resume Previous Interrupted Goal

3.3.7 Cry Lost

3.3.8 Victory Song

4.0 IMPLEMENTATION

4.1 RTOS

4.1.1 Architectural Overview

4.1.2 Scheduling

4.1.3 Timing

4.1.4 RTOS Conclusions

4.2 HIGH LEVEL CONTROL IMPLEMENTATION

4.2.1 Serial Port Read/Write

4.2.2 Coordinator

4.3 LOW LEVEL CONTROL IMPLEMENTATION

4.3.1 Serial Port Read/Write

4.3.2 Coordinator

4.3.3 Driving

4.3.4 Coin and Bumper
5.0 CONCLUSION ........................................................................................................... 26

6.0 REFERENCES ........................................................................................................... 29

APPENDIX A – SOURCE CODE ................................................................................ A-1

FILE CONTENTS ........................................................................................................ A-1
A.1 RTOS CODE ......................................................................................................... A-3
A.2 68HC11 CODE .................................................................................................... A-28
A.3 PALM CODE ........................................................................................................ A-58
A.4 SHARED FILES .................................................................................................. A-76

List of Figures

Figure 2.1 Allowable Home Starting Positions............................................................. 2
Figure 3.1 System Hierarchy and Communication Alphabets ..................................... 5
Figure 3.2 Top Level State Machine Behaviour ....................................................... 8
Figure 3.3 Finding Coins on the Grid ..................................................................... 10
Figure 3.4 Taking Coins Home ............................................................................... 11
Figure 3.5 Go Forward to Next Intersection ............................................................ 14
Figure 3.6 Turn Left ................................................................................................. 15
Figure 3.7 Drop Coin Left ...................................................................................... 16
Figure 4.1 High-Level Control Module Decomposition ........................................ 22
Figure 4.2 Low Level Control Module Decomposition ......................................... 24
1.0 Introduction

An autonomous mobile robot operates in a dynamic, unpredictable environment. The robot must be able to respond to these changes in the environment instantaneously. Because the robot hardware must interface imperfectly to the real world, the controlling system must be tolerant to faults in the sensors and immune to noise. This suggests a reactive, yet extremely robust system—typical requirements of many embedded real-time applications, such as those found in automobiles, fly-by-wire aircraft, telecommunication switching equipment, etc. [1]

This paper describes the design and implementation of an autonomous mobile robot able to “intelligently” navigate and retrieve objects. The robot was designed using a pre-assembled “Rug Warrior” robot described in the book by Jones and Flynn: Mobile Robots: Inspiration to Implementation [2] linked to a Palm PDA containing an embedded Motorola DragonBall processor (MC68EZ328) [3]. The robot’s task was to manoeuvre itself around a grid, pick up ‘coins’ and return them home. The robot must compete against (and possibly avoid) a similar device navigating on the same track. A more detailed description of the problem is provided in the following section.

2.0 Problem Specification

Given a 4x4 grid of black electrical tape and starting from a home position in one corner, our robot must navigate the grid to retrieve coins and return them to its home position. The coins are located on the intersections of the tape grid. While navigating the grid, our robot may at any time encounter its opponent—whose goal is the same, but has a home position directly opposite our robot. A robot may decide to turn around and follow a different path, or bump the other robot off track and continue its way. When a robot has completely lost where it is or can no longer track the tape, the robot may emit a loud sound to inform its owner it is lost. A lost robot is returned to its home position where it

---

must suffer a sixty second penalty. Whichever robot has more coins in its home position after 10 minutes is the victor. The robot must be able to move without hesitation. The following figure depicts the playing field and the possible home starting positions. The two competitors will have home positions diagonally opposite.

2.1 Hardware

The chassis of each robot is already fully assembled. To interface with their environment, the Rug Warrior kits contain:

- Two IR emitter/detectors to detect the edge of the black tape
- Two PWM DC motors for forward/reverse propulsion
- One PCM servo motor for raising and lowering the magnet
- One contact switch to detect when a coin has been picked up
- One front bumper to detect collisions
- One piezo-electric buzzer to make sound
- One microphone to detect sound
The Rug Warrior boards (MC68HC11 based) also contain a serial port interface. To handle all the high-level “intelligent” behaviours of our robot, the mobile unit will also carry an embedded Motorola MC68EZ328 (“DragonBall”) processor (contained in a Palm PDA) that communicates via UART to the 68HC11.

### 2.2 Software

The final constraint is each robot must run a common OS API. The specification for this real-time operating system (RTOS) was provided by the competition co-ordinator Dr. Mantis Cheng and is attached in this report as "os.h" (See Appendix A.1). The RTOS is a custom-built, prioritized, pre-emptive, multi-tasking microkernel, accommodating inter-process communication (IPC) through first-in first-out (FIFO) message queues.

While the hardware specifications were fairly fixed, the implementation of the OS, the behavioural design, the software architecture and the implementation of our controlling system were unconstrained. This allowed us to make critical design decisions in order to realize our system.

The following section presents our solution to the tape-tracking robot problem and the design decisions we made. This includes a detailed specification of the robot’s behaviours as well as the mechanisms used to achieve each behaviour. Section Four (see page 17) describes an implementation of the tape-tracking robot, from the custom RTOS to the control application built on top of the microkernel. The final section (see page 26) provides a summary of the project and draws conclusions about our successes/failures and which ideas would need revising for a future design.
3.0 Problem Solution

Our mobile robot system is composed of three parts: High Level Control, Low Level Control, and Robot Hardware. The specifications for these are as follows:

- **High Level Control** – A component of the software running on a Palm IIIx DragonBall Processor. The Palm runs an RTOS that schedules processes to perform high-level decision-making. These decisions control the actions of the mobile robot through the Low Level Control.

- **Low Level Control** – A component of the software running on a Motorola 6811 microcontroller. The 6811 will also run an RTOS to schedule the processes and drivers to handle low-level decision-making and Robot Hardware control. These decisions control the reactionary behaviour of the mobile robot, and interact with the Robot Hardware to create the necessary actions.

- **Robot Hardware** – A Rug Warrior mobile robot provides a base for the tape-tracking robot behaviour to run on. The robot supports input for: left and right tape sensors, front bumper, coin sensor, and microphone. The robot supports outputs for: left and right drive trains, magnet arm, and buzzer.

This project only deals with the design of the first two components, not the robot hardware. Design of the robot hardware was provided with the project, and robot functionality will be discussed only as it relates to the design of the first two components.

To understand its location, the robot must start at a specific starting location and orientation. For consistency, the robot will be started on the grid so that it is on the line segment of its home quadrant such that the grid is in front of it and to its right side (see Fig 2.1 for possible positions).
3.1 Communication

To allow the high-level behaviour to control the robot, the Low Level Control provides a system for specifying goals to it, and retrieving events from it. Goals are used to inform the Low Level Control what the high-level behaviour wants it to do. Goals are only sent to the Low Level Control when an event has been received from it. Events communicate information to the high-level behaviour about what is happening in the environment. The following diagram illustrates system communication and alphabets.

Goals:
- Go Forward to Intersection
- Turn Left
- Turn Right
- Drop Coin Left
- Drop Coin Right
- Resume
- Cry

Events:
- At Intersection with Coin
- At Intersection without Coin
- Completed Turn
- Bumper Hit
- Lost

Control Signals: Specific low level signals for controlling the motors and actuators.

Input Signals: Specific low level inputs reporting the state of the robot sensors.

Figure 3.1 System Hierarchy and Communication Alphabets
3.1.1 Goals

These are all of the goals that may be sent from the high-level behaviour to the low level control:

- **Go Forward to Next Intersection** – Drive the robot along the tape-track until it reaches the next intersection.
- **Turn Left** – Turn the robot to the left when on an intersection. After completing the turn, the robot should be well centered enough to simply invoke the Go Forward behaviour and stay on the track.
- **Turn Right** – Turn the robot to the right when on an intersection. After completing the turn, the robot should be well centered enough to simply invoke the Go Forward behaviour and stay on the track.
- **Drop Coin Left** – Drop the coins the robot is holding while executing a left turn. The robot must be on an intersection to do this.
- **Drop Coin Right** – Drop the coins the robot is holding while executing a right turn. The robot must be on an intersection to do this.
- **Resume Previous Interrupted Goal** – If the robot’s previously specified goal was interrupted by an asynchronous event such as a Lost Timer, or Bumper Hit then this tells the robot to complete the previous goal by resuming from where it was interrupted.
- **Victory Song** – Play a victory song on the buzzer. Used to indicate that a coin has been dropped in the home corner.
- **Cry Lost** – Play a cry song on the buzzer. Used to indicate that the robot has lost the tape-track and should be returned to the home corner by the owner.

The procedure for physically carrying out each of these goals will be discussed in the section on Low Level Control.
3.1.2 Events

These are all of the events that may be sent from the low level control to the high-level behaviour:

- **At Intersection with Coin** – Indicates that the robot has arrived at an intersection and that it is holding a coin.
- **At Intersection without Coin** – Indicates that the robot has arrived at an intersection and that it is NOT holding a coin.
- **Turn Complete** – Indicates that the robot has completed turning and aligning.
- **Bumper Hit** – Indicates that the bumper on the front of the robot has been hit.
- **Lost Timer** – Indicates that the robot has not reached the next intersection in the minimum amount of time, and it has not detected any part of the tape-track.

The response of the high-level to these events is discussed in the description of each high-level behaviour.

3.2 High Level Control

The purpose of the High Level Control is to create behaviour for the robot that is successful within the game environment. To create this behaviour an RTOS is used to schedule processes that interact and form an “intelligent” high-level behaviour and provide goals to the Low Level Control. The high-level behaviour handles three main tasks, described here and depicted in Fig. 3.2:

- **Finding Coins on the Grid** – This behaviour causes the robot to find an optimum solution for grid traversal under unpredictable conditions. Visited grid intersections are tracked and the most favourable shortest path is always chosen. The one step path is calculated at each intersection taking obstacles into account.
- **Taking Coins Home** – This causes the robot to take the shortest path to the home corner when carrying one or more coins. The path home is calculated one step at a time, backtracking when obstacles are hit.

- **Getting Home when Lost** – If during any of the preceding behaviours the robot loses track of the tape grid and cannot determine its position it will attempt to get home. The robot will not attempt to find the tape grid because if it did it would be at an unknown location on the grid. To get home the robot will cry until its owner takes it home and hits the bumper to reset its known position.

![High Level Control State Diagram](image)

**Figure 3.2 Top Level State Machine Behaviour**

The three behaviours in the High Level Control work together to enable the robot interact with its environment. The waiting state is to put the High-Level in a known start state and to synchronise communication with the Low-Level. From this state, the High-Level will only become active after a bumper event. As events are received from the Low Level Control they are handled by the active behaviour and are reacted to by generating a new goal. Some events may cause the active behaviour to change before it generates the new goal. Following are in-depth descriptions of the three main behaviours.
3.2.1 Finding Coins on the Grid

To find coins on the grid the robot uses a flexible searching algorithm to traverse grid intersections. The algorithm is biased to search grid intersections near the home corner to allow it to retrieve the closest coins first. Each time the robot arrives at an intersection without a coin and receives the “At Intersection without Coin” event a new destination intersection is chosen. The algorithm for choosing the destination intersection is as follows:

Go to the adjacent intersection that is closest to the home corner and has not yet been visited. If two intersections are of equal distance then choose the one that you can reach with the least number of 90-degree turns. If adjacent intersections have already been visited then go to the closest intersection not yet visited. If two intersections are of equal distance then choose the one that requires the least number of 90-degree turns, and the closest to the home corner.

If all of the intersections have been searched and you aren’t currently carrying a coin, re-mark all of the intersections as having not been visited and continue searching.

This is specified formally by the following FSM:
After deciding which intersection to visit the algorithm generates the next goal. The goal is one of: Go Forward to Next Intersection, Turn Left, or Turn Right. Each time another event is received the next necessary movement is determined and the corresponding goal is generated.

If while traveling to the destination intersection, the robot’s front bumper is hit, the destination intersection is recorded as being visited but the robot will not abort its current action. If an “At Intersection with Coin” event is received the robot switches to the “Taking Coins Home” behaviour.
3.2.2 Taking Coins Home

To get a coin back to the home corner the robot uses a simple shortest path home algorithm. The algorithm tries to minimize the amount of turning needed to get home.

The algorithm for taking a coin home is as follows:

If going to the adjacent intersection in front of the robot gets it closer to the home corner then go there. Otherwise go to the intersection to the left or the right depending on which one brings the robot closer to the home corner.

This is shown visually by the corresponding FSM.

![High Level Control State Diagram for Take Coin Home State](Image)

Figure 3.4 Taking Coins Home

After deciding which intersection to visit the algorithm generates the next goal. The goal is one of: Go Forward to Next Intersection, Turn Left, or Turn Right. Each time another event is received the next necessary movement is determined and the corresponding goal is generated.
When the robot reaches the home corner it will generate a “Drop Coin Left” or “Drop Coin Right” goal depending on the orientation of the robot to the home corner. This will drop the coins in the home corner. Once the next event, “At Intersection without Coin” has been received the robot switches to the “Finding Coins on Grid” behaviour. If while returning home an “At Intersection without Coin” event is received the robot switches to the “Finding Coins on Grid”.

### 3.2.3 Getting Home when Lost

If the robot receives a “Lost Timer” event or the high-level behaviour determines the robot is lost then the robot must get help to return to the home corner. To get help a “Cry Lost” goal is generated. The owner then re-places the robot at its home position. Once a “Bumper Hit” event is received, the robot switches behaviours to the “Finding Coins on the Grid” behaviour if it is not holding a coin, or the “Taking Coins Home” behaviour if it is holding a coin.

Each of the high-level behaviours generates different goals, depending on its interpretation of the current environment. It is left to the Low-Level Control to carry out each goal.

### 3.3 Low-Level Control

The purpose of the Low Level Control is to create a bridge between the planning of the high-level behaviour and the hardware of the mobile robot. An RTOS is run in the Low Level Control to schedule processes to implement this bridge. The purpose of the bridge is to map goals received into controls for the robot, and input from the robot hardware into events for the high-level behaviour.

To allow the Low Level Control to run in a consistent state the robot must begin the game with the magnet arm down and oriented as specified earlier.
The following sections discuss in detail the steps that are taken in response to each goal received and what events are sent in response.

### 3.3.1 Go Forward to Next Intersection

To achieve this goal the robot drives forward at full speed until both tape-tracking sensors read black, indicating an intersection. The robot then continues driving forward until one or both tape-tracking sensors read white.

Because wheel speeds are independent—and even when driven with the same power each wheel will not in general spin at the same rate—the robot will always veer to the right or left. Constantly stopping and pivoting back onto the tape when one tape sensor reads black causes abrupt jerks in the motion of the robot and can lead to uncontrollable oscillations. To overcome this, our system will never completely stop either wheel. Instead, when one side is detected to be on the tape, we attempt to smoothly curve our way back onto the tape. The algorithm is as follows:

- If both tape-tracking sensors read white then continue going forward at full speed.
- If one of the tape-sensors read black and the other sensor reads white, then cut the speed of the black side by three quarters. Incrementally increase the speed of this wheel until both tape sensors read white again.

If both tape-tracking sensors read black then you are crossing an intersection. As soon as either side detects white again, send an “At Intersection with Coin” or “At Intersection without Coin” to the high-level control, depending on whether or not a coin has been picked up.

This is illustrated visually by the following FSM diagram.
To turn, the robot circles on the spot. To do this to the left, the left wheel is put in reverse, and the right wheel is driven forward. To complete the goal correctly the robot is assumed to have started at a point sufficiently passed the intersection, such that rotating on a dime will place it nearly aligned on the new tape track. The robot reaches this sufficient point by a combination of the “Go Forward...” behaviour and inertia.

Because we identify having crossed an intersection as reading both tape sensors reading black, followed by either tape sensor reading white (see Fig 3.5), the turn left goal must handle the cases where the turn begins with the left sensor on white or on black. The algorithm and FSM (see Fig. 3.6) are as follows:

Drive the right wheel at half the regular forward speed. Drive the left wheel at half the regular reverse speed. When the left tape sensor reads black after first reading white stop the wheels and issue a turn completed event.
3.3.3 Turn Right

The turning right goal is the dual of Turn Left. For the sake of brevity, the steps to turn will not be repeated.

3.3.4 Drop Coin Left

This goal must be sent when the robot is positioned on an intersection. The motor control sequence is the same as executing a left turn. However, when the robot has turned completely off the tape, the servo motor raises the magnet and drops the coin. Once the left tape sensor detects the tape again, the servo motor is returned to the default lowered position. The robot is now ready to track the tape again. The FSM is given below.
3.3.5 Drop Coin Right

The “Drop Coin Right” objective is the dual of the “Drop Coin Left” described above. The process is the same with the perception of left and right switched.

3.3.6 Resume Previous Interrupted Goal

When an asynchronous event occurs (e.g. Bumper Hit or Lost Timer), the state of the current goal is saved. To accomplish a “Resume Previous Interrupted Goal” the robot attempts to resume execution of the goal where it was interrupted.

3.3.7 Cry Lost

This goal is accomplished by emitting a high-pitched wailing sound from the buzzer. The robot stops all motion and continues to ‘cry’ until the front bumper switch is contacted. When the bumper is contacted, a “Bumper Hit” event is sent to the High Level Control and the screaming stops. When the robot begins crying, it is assumed the owner will recover the robot and reset it to its home position. Thus, only the owner will trigger the robot’s bumper once it has been placed in a known reset position.
3.3.8 Victory Song

To accomplish this task, the robot emits a sequence of tones to the buzzer. Since, this goal is only generated after successfully completing a drop coin goal, the robot will always be positioned on an intersection. When the sequence is completed, report an “At Intersection without Coin” event. The melody is to be determined. If the robot’s front bumper is hit while singing, stop singing and send a “Bumper Hit” event.

4.0 Implementation

Before even designing the application software to give the robot behaviour, an implementation of the OS API needed to be created and tested. This was one of the toughest portions of the entire project. A multi-tasking real-time kernel brings about many crucial, interrelated considerations. A description of our implementation of the RTOS with the specified interface semantics is given in Section 4.1 and the source code is listed in Appendix A.1.

When the stable, tested kernel was ready and our robot control system designed, we began transfer of our design to code. As we became more familiar with the hardware we adapted our design to better reflect the physical properties of robot’s environment. The translation of the final design (presented in Section 3 above) to code is highlighted in Sections 4.2 and 4.3 with the complete source code supplied as Appendix A.2 and A.3.

4.1 RTOS

The specifications for our real-time operating system stipulate the standard basic operating system primitives: process creation, process termination, and yield. However, instead of any blocking message passing or communication protocol, inter-process communication (IPC) is achieved with first-in first-out queues (FIFOs). The kernel supports three levels of priority in its pre-emptive scheduling policy: device, periodic, and sporadic. In the design and construction of our solution we learned many intricacies involved with general operating system design as well as real-time scheduling. A brief description of the operation of our OS is given in the following sections. For a detailed account, please see the well-commented source code provided in Appendix A.1.
4.1.1 Architectural Overview

The RTOS is made up of two main parts: the scheduler/API and the Back End (BE). The code in os.c is responsible for scheduling the processes and providing an API to the user. The BE that is implemented in backend.c is responsible for managing the data structures necessary for the RTOS to function.

The main concern of the RTOS is timing and scheduling, and so will be discussed later after the different types of processes have been discussed.

The BE is the heart of the code, responsible for supporting the functionality of the RTOS. Its main purpose is to manage the data structures needed by the RTOS. The RTOS needs two main types: Process Descriptors and FIFO Descriptors. Process Descriptors encapsulate everything related to a process. FIFO Descriptors do the same for FIFOs. The data structures that the BE uses are: a sporadic process queue, periodic process plan, device process delta queue, process registry, and FIFO registry.

Each of these data structures has a different responsibility in the BE. The process registry contains all current processes and dead processes. Whenever a new process is to be created and added to one of the three process queues it is assigned from this process registry. Each of the three process data structures is described later in its corresponding process type section. The BE also maintains the FIFO registry that keeps track of the status of all FIFOs. When a FIFO is initialized it is assigned from this FIFO registry. Writing and reading of FIFOs is also managed by the registry within the BE. FIFOs are implemented using a bounded circular buffer.

4.1.2 Scheduling

The three level scheduling policy allows for handling of periodic, aperiodic, and rate-based device processes. All sporadic (aperiodic) processes are executed first-come-first-served (FCFS) when the CPU is idle. Periodic processes are given scheduled time slices according to a plan fixed at compile time. Rate-based device processes are highest
priority and will pre-empt other processes until they choose to yield or terminate. To implement this policy according to the specifications provided entailed resolving many anticipated and some completely unexpected issues. The scheduling levels and primitives were implemented and tested in increments, beginning with the simplest, sporadics (lowest priority), and finishing with the most timing sensitive, devices (highest priority).

### 4.1.2.1 Idle Process

The Idle process is a unique process. It is specially created by the BE to fill in empty CPU time. When no other process is ready to run, the Idle process is run until another one becomes ready.

### 4.1.2.2 Sporadic Processes

Sporadic processes only execute when the CPU is idle. The CPU is idle when there are no devices to run and either a periodic has yielded without using its maximum time slice, or, explicit idle time is specified in the periodic process scheduling plan (PPP).

As sporadic processes are created, they are added to the sporadic queue maintained by the operating system. Because sporadics are FCFS and run to completion, new sporadic processes are always added to the end of the sporadic queue. This property also makes the scheduling simple. Any time the CPU is idle, run the process at the front of the queue. If a higher priority process pre-empts a sporadic process, the sporadic remains at the front of the queue. If a sporadic process yields its control of the CPU it is put to the end of the sporadic queue.

### 4.1.2.3 Periodic Processes

The cyclic, fixed-order periodic scheduling plan is declared at application compile time. It specifies the name of periodic processes to be run and the maximum time slice each is allowed. No two processes will exist with the same name simultaneously, but if a process is terminated another can be created later on to occupy that place in the periodic plan. A periodic process may occupy more than one location in the plan.
When a periodic process is scheduled, it can execute for up to its allowable maximum time slice. A periodic process may yield before its maximum time has expired, at which point its remaining execution time becomes idle, and a ready sporadic process can be run. If a higher priority device process needs to be scheduled during execution of a periodic process, the execution state of the periodic is saved and the device is allowed to complete. The time required to execute the device process is lost from the allowed execution time of any pre-empted periodic processes.

The BE is responsible for preserving the sequence of the periodic scheduling plan. To do this it maintains an index into the user-defined PPP. Because periodic process are stored by name in the PPP, the BE optimizes the lookup of process descriptors by maintaining a direct mapping of names into process descriptors. At the end of every period, the RTOS notifies the BE to allow it to update the current periodic process.

### 4.1.2.4 Device Processes

Device processes execute for a finite amount of time at a fixed rate. They are the highest priority process and interrupt any other non-device process when the time of their next needed execution occurs.

Device processes are maintained in a delta queue. Each device process in the queue stores the time of how long after the previous device process it should run (delta time). This strategy reduces the amount of updates and searches that must be done on the queue of devices. By keeping the queue sorted by delta time, the next device process that must be run is always at the front of the queue. When a device completes its execution by yielding, it is re-inserted into the delta queue in sorted order based on its rate. When a process is added to or removed from the delta queue, the delta time of the following device process is updated.
4.1.3 Timing

The timing and context switching is managed by the scheduler. It coordinates with the BE to manage to the state of the processes and FIFOs. Context switching can occur when a process yields, a timer expires, or upon creation/termination of a process. In a real-time system where timing is sensitive, the timer must always activate the correct instant.

A timer event occurs to signify the end of a period in the scheduling plan, or the start of a device process. If a device process is pending, the timer starts the device process. Otherwise, if a period has expired, the timer selects the new process to run and calculates the time until the next periodic event. Then, if there was no device pending, the next timer event is scheduled for the earliest of: the next end of a period or the next start of a device process.

When a process is created or terminated the RTOS may have to change the process that is currently running and/or change the next scheduled timer event.

When a process yields, the hardware traps into the RTOS. Depending on which type of process yielded and the current pending processes, the RTOS will schedule the next process and possible timer event accordingly.

4.1.4 RTOS Conclusions

To ease development, the RTOS was developed in two main pieces. One part is responsible for providing the API to the user applications and handling program and timer events. The BE is responsible for maintaining the state of all necessary data structures. While they interact through simple function calls, their interrelation can be quite complex. To implement even the basic functionality of the specified OS created complex scheduling concerns.
4.2 High Level Control Implementation

The following diagram illustrates the modular breakdown of our High-Level Control implementation. Arrows from the RTOS indicate creation of processes. Information transferred via function call or FIFO is shown as a labelled arc.

The High Level Control of the mobile robot was implemented on a Palm IIIx handheld computer containing a Motorola DragonBall processor. The Palm ran the RTOS developed for this project to schedule the necessary processes. The High Level Control is made up of three processes: Serial Write, Serial Read, and Coordinator.

4.2.1 Serial Port Read/Write

The Serial Write and Serial Read processes are periodic processes that are responsible for bridging communication between the Palm and the Low Level Control. The Serial Write process provides a FIFO that allows processes to send data. Each time the process runs it checks if there is any data in the FIFO, and if so it sends it out through the serial port. The Serial Read process is the opposite of the Serial Write process. Each time it runs it
checks the serial port for any incoming data. If any data is received it is placed in a FIFO that any other process can read from. These two processes allow any process to independently read or write from the serial port.

4.2.2 Coordinator

The Coordinator process is the High Level Control behaviour described in Section 3.2. It implements the High Level Control state machines that respond to messages from the Low Level Control. To respond to messages it uses the Serial Read and Serial Write global FIFOs. The process is scheduled as a sporadic process. This allows the process to run in any idle time left over from the periodic Serial Read and Serial Write processes. Extra idle time is also specified in the periodic schedule to ensure the Coordinator process has a specific minimum amount of time.

For the complete translation of our system design, please see the attached fantastically readable source code (see Appendix A.3).

4.3 Low Level Control Implementation

The following diagram illustrates the modular breakdown of our Low Level Control implementation. Arrows from the RTOS indicate the type of the created process. Communication and connectivity are shown as labelled arcs.
The Low Level Control of the mobile robot is implemented using a Motorola 68HC11 micro-controller on a Rug Warrior mobile robot platform. The Low Level Control is run using the RTOS developed for this project. To create the required functionality of the Low Level Control many different processes run and interact. These processes are: Serial Write, Coordinator, Driving, Tape Sensor, and Coin and Bumper. There are also interrupt handlers for Serial Read, Motors and Timer, and Servo Control.

### 4.3.1 Serial Port Read/Write

The Serial Write process is a driver process that transfers any data from its input FIFO across the serial channel. Any process can use this FIFO to send data to the High Level Control. The driver is scheduled periodically. Each time it runs the process checks for new data in the FIFO and sends it the serial port. The Serial Write driver is
complemented by the Serial Read device driver. Serial Read is a hardware interrupt, generated whenever new data is received by the 68HC11 UART. This interrupt handler reads the new data from the serial port and puts it in a FIFO for any process to use. Serial Read is an interrupt because the 68HC11 micro-controller can only buffer one byte, and so if the buffer were polled it is possible messages would be lost.

### 4.3.2 Coordinator

The Coordinator process is a sporadic process that handles incoming messages from the High Level Control and delegates commands to the other processes. The process is also allocated specific idle time so that it will run at least a minimum amount of time per periodic schedule, while still taking advantage of the other processes idle time. This process is the control centre for the Low Level Control, and coordinates all received and sent messages. It sends the appropriate stimuli to the Low Level Control state machines to achieve the required behaviours. The Coordinator also receives the responses and relays the proper events to the High Level. To communicate with the other processes, the Coordinator uses: two FIFOs to send and receive data to/from the Driving process and two FIFOs to receive data from the Coin and Bumper process.

### 4.3.3 Driving

The periodically scheduled Driving process implements all the robot’s movement goals. The Driving process receives all its relevant messages from the high-level via the Coordinator. When a new message is received through its FIFO, the driving process invokes the associated FSM, through a function call to: goForwardToIntersection, turnLeft, turnRight, dropCoinLeft, or dropCoinRight. Each function is the translation of its corresponding FSM, described previously in Section 3.3. To stay on track, the Driving process receives the current status of tape sensors from the tape sensor driver.

The Driving module requests readings from the tape sensor driver. The tape sensor driver is scheduled periodically to wake, and if requested, read the current input values. The process does some filtering and averaging to overcome faults and noise in the sensors before informing the Driving module of the current tape sensor reading.
4.3.4 Coin and Bumper

A single periodic process drives all the operation of the A/D converter. Because the bumper and the coin detector are connected to the analog inputs, their digital value must be read from the A/D subsystem of the 68HC11. The periodic CoinBump process initiates a sequence of conversions on the input pins for the coin detector and the bumper. To compensate for any faults or noise in the coin sensor, the process maintains a window of the previous readings. The majority reading within the window confirms a valid input. CoinBump informs the Coordinator whenever the state of one of these A/D inputs changes, i.e. when NoCoin changes to Coin or NoBump changes to Bump.

The complete, gruesome detail of the translated Low Level Control FSMs is available as Appendix A.2.

5.0 Conclusion

To create productive behaviours we designed a two-level architecture to run on top of the mobile robot OS and hardware. To achieve the objective of collecting coins and bringing them home, a balance of planning and reactive control is needed. In an unpredictable, real-time environment not every situation can be planned for and a purely reactive robot would lack purpose. In our design this required partitioning the system into two communicating controllers: the High Level Control and the Low Level Control.

The High Level Control decides which behaviour to exhibit and the goals required to accomplish the behaviour. The High Level Control sequentially sends each goal to the Low Level Control as events occur. The Low Level Control is responsible for execution of the defined steps required to complete each goal. The Low Level Control receives goal messages from and sends event messages to the high level. This allows us to model a purposeful behaviour able to adapt and interact with a dynamic environment.

These control systems run on separate processors, each running our custom RTOS. The RTOS provides support for three classes (priorities) of processes: device, periodic, and
sporadic. Developing a stable kernel was the most challenging portion of the project. However, the RTOS facilitated development of multi-threaded user applications like the High Level Control and the Low Level Control.

Once our design was finalized into code, the robot operated very effectively. The physical tracking and manoeuvring abilities of the mobile robot were better than anticipated. The robot’s motion was smooth and steady. Even turning, the slowest operation for the robot, was quick and reliable. While communication between controllers was at an apparently slow 9600 baud, the High-Level was able to respond and direct the Low-Level virtually instantly. The robot never hesitated as it navigated the grid.

We did however encounter many difficulties on the way to designing our mobile robot. We originally intended to make more use of device level processes in our application. However, the instabilities present in our implementation of device process made simply rescheduling with periodic processes a more time-efficient option. Some portions of our original design needed to be revised once the hardware limitations were explored; the effects of inertial motion was never really considered in our first draft. Also, our initial design provided a behaviour reverse along the tape grid. However, the physical implications proved to daunting and the time was need on more fundamental behaviours. We also struggled to write code that worked well on any Rug Warrior. Because the hardware was provided and stored in the lab, it was never guaranteed which robot you would be working on. Our driving is scheduling and timing dependant and each robot moved differently. Eventually we tweaked the driving for a specific robot and consequently had trouble running our application on other hardware.

The implementation was also designed to be expandable, allowing brand new or improved functionality to be added later. Code to allow the use of the Palm PDA buttons is incorporated but currently commented out. Currently the robot tracks the tape very well. However, code has been written (but commented out) to adapt the ratio of power supplied to each wheel and compensate for the veering caused by wheel speed imbalance.
With the current architecture, adding more detailed communication between the high and low level would be easy. Adding new sensors would just require defining new messages to notify the high level of sensor events. As well, defining new goal actions, such as turning 180 degrees or reversing along the track would fit easily into our design. While our design turned out excellent, there is always more features or improvements that can be made.

In conclusion, this project was pain-staking, frustrating, and intense from the beginning, but very rewarding to see the final creation in action. The design and implementation of an entire real-time system (OS, drivers, application) is not a trivial task. It brings to light the fundamental issues of concurrency and real-time scheduling in a complex control system. The project emphasized requirements analysis and design specification of an embedded real-time system. It was an impressive challenge, but I would do it again in a second (and do it better!).
6.0 References


Appendix A – Source Code

File Contents

RTOS Code (listed in Appendix A.1)

Os.h: Contains all the function declarations and interface semantics of the RTOS API.
Os.c: Contains our implementation of the specified RTOS.
Backend.c and Backend.h: Contains all the data structures and methods to manipulate them required by the OS

6811 Low-Level Controller Code (listed in Appendix A.2)

Rproj5.c: Starts the RTOS and creates the coordinator process.
Sched.h: Definitions for the periodic process scheduling.
Rcomm.c and Rcomm.h: Contains the process and interrupt handler for interfacing FiFos with the serial port.
Rcoord.c and Rcoord.h: Contains the coordinator process for the 6811 that implements the 6811 state machine.
Driving.c and Driving.h: Contains the process for controlling the motors and responding to the tape sensors.
Gofwd.c: Contains the state machine for driving forward, used by the driving process.
Turnleft.c and Turnright.c: Contains the state machines for turning, used by the driving process.
Dropleft.c and Dropright.c: Contains the state machines for dropping coins, used by the driving processes.
Tapesen.c and Tapesen.h: Contains the process that provides tape sensor data to the driving process.
Coinbump.c and Coinbump.h: Contains the process for using the A/D converter to use the coin sensor and bumper.
Magnet.c and Magnet.h: Provides support for using the servo to control the magnet.
Motors.c and Motors.h: Provides support for using the motors.
Support.c and Support.h: Provides commonly used operations.

The other files not discussed are taken from the sample code provide on the CSC 460 web page.
Palm High-Level Controller Code (listed in Appendix A.3)

Pproj5.c: Starts the RTOS and creates the coordinator process.
Pcomm.c and Pcomm.h: Contains the processes for interfacing FiFos with the serial port.
Pcoord.c and Pcoord.h: Contains the coordinator process that implements the Palm state machine.
Nav.c and Nav.h: Provides functions for calculating orientation and direction to the destination.
Sched.h: Provides definitions for periodic scheduling.
Keys.c and Keys.h: Provides support for using the buttons on the Palm.

The other files not discussed are taken from the sample code provide on the CSC 460 web page.

Shared (listed in Appendix A.4)

Messages.h: Contains definitions for the messages passed between the Palm and the 6811.
A.1 RTOS Code

```c
#ifndef _OS_H_
#define _OS_H_

/* MHMC UVic/CS (May 22/2002) */
/* DO NOT EDIT THIS FILE! */

/*==================================================================
*             T Y P E S   &   C O N S T A N T S
*==================================================================
*/
/*==================================================================
*             L I M I T S
*==================================================================
#define MAXPROCESS 16     /* max. # of processes supported */
#define MAXFIFO 16       /* max. # of FIFOs supported */
#define FIFOSIZE 8       /* max. # of data elements per FIFO */
#define WORKSPACE 512    /* workspace of each process in bytes */

/*==================================================================
*             I N V A L I D   C O N S T A N T S
*==================================================================
#define INVALIDPID 0      /* id of an invalid process */
#define INVALIDFIFO 0     /* an invalid FIFO descriptor */

/*==================================================================
*             S C H E D U L I N G   L E V E L S
*==================================================================
#define SPORADIC 2        /* first-come-first-served, aperiodic */
#define PERIODIC 1        /* cyclic, fixed-order, periodic */
#define DEVICE 0          /* time-driven cyclic device drivers */

/*==================================================================
*             W E L L - K N O W N   P R O C E S S   N A M E
*==================================================================
#define IDLE -1           /* name of an IDLE process */

/*==================================================================
*             P R E - D E F I N E D   C O N S T A N T S
*==================================================================
#define TRUE 1            /* Boolean */
#define FALSE 0           /* Boolean */

/*==================================================================
*             P R E - D E F I N E D   T Y P E S
*==================================================================
typedef unsigned int FIFO;
typedef unsigned int Pid;
typedef unsigned int bool;

/*==================================================================
*             P E R I O D I C   P R O C E S S   S C H E D U L I N G   P L A N
*==================================================================
extern int PPPLen;    /* length of PPP[] */
extern int PPP[];      /* PERIODIC process scheduling plan */
extern int PPPMax[];   /* max CPU in msec of each process in PPP */

/*==================================================================
*             A C C E S S   P R O C E D U R E S
*==================================================================
*/

/*==================================================================
*             O S   I N I T I A L I Z A T I O N
*==================================================================
void OS_Init();
void OS_Start();
void OS_Abort();

/*==================================================================
*             P R O C E S S   M A N A G E M E N T   P R I M I T I V E S
*==================================================================
Pid OS_Create(void (*f)(void), int arg, unsigned int level, unsigned int n);
void OS_Terminate(void);
void OS_Yield(void);
int OS_GetParam(void);
```

/* FIFO primitives */
FIFO OS_InitFiFo();
void OS_Write( FIFO f, int val );
bool OS_Read( FIFO f, int *val );

/*==================================================================
*        S T A N D A R D   I N L I N E    P R O C E D U R E S
*==================================================================
*/

/* inline assembly code to disable/enable maskable interrupts
  * (N.B. Use with caution.)
  */
#define OS_DI()    asm(" sei ")  /* disable all interrupts */
#define OS_EI()    asm(" cli ")  /* enable all interrupts */

/*==================================================================
*          O S   I N T E R F A C E    S E M A N T I C S
*==================================================================
*/

/* GLOBAL ASSUMPTIONS:
   * - OC4 and SWI interrupts are reserved for use by the OS.
   * - OC4 is used for pre-empting processes when their alloted time is used up.
   * - SWI is used for kernel calls.
   * - All runtime exceptions (where assumptions are violated) or other
     unrecoverable errors get handled by calling OS_Abort().
   * - PPP[] and PPMax[] must be of the same length, PPPLen.
   * - Each entry in PPMax[] must be between 1 and 10 msec inclusive.
   * - PPPLen = 0 is the same as PPPlan = [IDLE] and PPMax = [infinity].
   * - All unspecified errors have undefined behaviours.
   *
   * SCHEDULING POLICY:
   *
   * There are three levels for processes: DEVICE, PERIODIC and SPORADIC.
   * SPORADIC processes are scheduled in a FCFS manner. PERIODIC processes
   * are scheduled according to a cyclic fixed-order plan. PERIODIC
   * processes have higher priority than SPORADIC processes. DEVICE
   * processes have a predefined fixed rate of execution. They have the
   * highest priority and pre-empt both SPORADIC and PERIODIC processes.
   * Two DEVICE processes coincide on the time interval will be scheduled
   * in an arbitrary order. We assume DEVICE processes have very short
   * execution time.
   *
   * PERIODIC PROCESSES:
   *
   * PERIODIC processes are scheduled according to a cyclic fixed-order
   * scheduling plan, defined by the PPP[] array. When a PERIODIC process
   * is created, it is assigned a name "n" (an index between 0 and
   * MAXPROCESS-1).
   *
   * This name is fixed and can NEVER be changed again. The Pid of this
   * process is not the same as this name. No two processes are allowed
   * to have the same name at the same time. The same name may be reused
   * over time.
   *
   * The PPP[] array is a sequence of names which specifies the execution
   * order of all PERIODIC processes. The name of every PERIODIC
   * process must appear in PPP[] array at least once, but may be more
   * than once.
For example, if we create three PERIODIC processes with names A, B and C out of three functions P(), Q() and R() respectively. Then,
\[ \text{PPP}[] = \{ A, B, A, C \} \]
means executing A, then B, then A again, then C, then A again, and so on. If P() terminates, but the name A is later assigned to another instance of P() with a different Pid, then A will be executed again according to PPP[] order. In a sense, the PPP[] specifies at least a single execution cycle of all PERIODIC processes.

In addition, each PERIODIC process is assigned a maximum CPU time, given by the array PPPMax[]. The process name IDLE is reserved for introducing explicit CPU idle time. For example,
\[ \text{PPP}[] = \{ A, \text{IDLE}, B, \text{IDLE}, A, C, \text{IDLE} \}; \]
\[ \text{PPPMax}[] = \{ 5, 2, 3, 3, 10, 7, 10 \}; \]
After completing A within 5 msec, the processor idles 2 msec, then starts B for 3 msec, then idles another 3 msec, then starts A again for 10 msec, then starts C for 7 msec, then idles 10 msec, then repeats all over again. The total cycle time of all PERIODIC processes is thus 40 msec. Each PERIODIC process is guaranteed to execute exactly according to the specified order and period of PPP[] and PPPMax[].

CPU IDLE TIME:

The CPU is "idle" if:
1. a PERIODIC process yields/blocks before its max CPU time is exhausted; or
2. the IDLE process is the current PERIODIC process.

SPORADIC PROCESSES:

SPORADIC processes are allowed to run only during CPU idle time. A ready SPORADIC process runs at the next earliest CPU idle time. When a SPORADIC process is pre-empted, that is by the next PERIODIC process, it re-enters its level at the front. When a SPORADIC process yields, or resumes after being unblocked, it re-enters its level at the end.

IMPORTANT NOTE:
Once, OS_Start() has been called, PPP[] and PPPMax[] arrays must not be changed.

DEVICE PROCESSES:
DEVICE level processes are designed to read/write I/O devices at specific data rates. They run for a very short time when activated. They move data in/out of I/O devices. They must be run at a precise rate. When a DEVICE process is created, its "n" specifies its rate of execution, e.g., 20 (i.e., once every 20 milliseconds).
Multiple DEVICE processes may collide at the same time interval. For example, one at every 20 msec and one at every 50 msec will collide once every 100 msec. Their execution order is not specified as long as both are executed eventually. Assume that DEVICE processes have very short execution time, then the jitter caused by collision should be acceptable. DEVICE processes have the highest priority. When they are ready, they are executed immediately, i.e., they pre-empt all other processes. All DEVICE processes run to completion, i.e., they run until they yield or terminate. When they yield, they will be resumed until next period.

ACCESS CALLS:
#include <stdio.h>

void OS_Init(void)
{
    The function main() will be called first by crt11.s. Before any calls
can be placed to the OS, main() must call OS_Init() to initialize the OS.
main() can then create processes and initialize the PPP[] and PPPMax[]
arrays. To boot the OS, main() must call OS_Start() which never returns.
Before the call to OS_Start(), the only calls that may be placed to the
OS are OS_Create().
Assumption: OS_Init() is called exactly once at boot time.
}

void OS_Start()
{
    OS_Start() will only be called once and only after the PPP[] and PPPMax[]
arrays have been initialized and OS_Init() has been called.
}

void OS_Abort()
{
    stop the OS immediately due to an unrecoverable error.
}

Pid OS_Create( void (*f)(void), int arg, unsigned int level,
                unsigned int n )
{
    A new process "p" is created to execute the parameterless
    function "f" with an initial parameter "arg", which is retrieved
    by a call to OS_GetParam(). If a new process cannot be
    allocated, INVALIDPID is returned; otherwise, p's Pid is
    returned. The created process will belong to scheduling "level",
    which is DEVICE, SPORADIC or PERIODIC. If the process is PERIODIC, then
    the "n" is a user-specified index (from 0 to MAXPROCESS-1) to be used
    in the PPP[] array to specify its execution order.
    Assumption: If "level" is SPORADIC, then "n" is ignored. If "level"
    is DEVICE, then "n" is its rate.
}

void OS_Terminate( void )
{
    Terminate the calling process; when a process returns, i.e., it executes
    its last instruction in the associated function/code, it is
    automatically terminated.
}

void OS_Yield( void )
{
    Reschedule the calling process; that is, the calling process
    voluntarily gives up its share of the processor.
}

int OS_GetParam(void)
{
    Retrieve the parameter ( "arg" ) provided by OS_Create().
}

INTERPROCESS COMMUNICATION:
    FIFOs are first-in-first-out bounded buffers. Elements are read in the
    same order as they were written. When writes overtake reads, the first
    unread element will be dropped. Thus, ordering is always preserved.
    "read" and "write" on FIFOs are atomic, i.e., they are indivisible, and
    they are non-blocking. All FIFOs are of the same size. All data elements
    are assumed to be unsigned int.

FIFO  OS_InitFiFo()
{
    Initialize a new FIFO and returns a FIFO descriptor. It returns
    INVALIDFIFO when none is available.
}

void OS_Write( FIFO f, unsigned int val )
{
    Write a value "val" into the FIFO "f". A write always succeeds. When
    a FIFO is full, the first unread element is dropped.
}

bool OS_Read( FIFO f, unsigned int *val )
{
    Return the first unread element in "f" if it is unavailable. If the FIFO
    is empty, it returns FALSE. Otherwise, it returns TRUE and the first
    unread element is copied into "val".
}"
// Implement the API specified by os.h

#include "os.h"
#include "newsystem.h"
#include "backend.h"
#include "MC68EZ328.h"
#include "uart.h"
#include <string.h> //for NULL

// interrupt vector table
static void **ramvec;

//Current Process
static ProcessDescriptor* CurrentProcess;

//Idle Process
extern ProcessDescriptor IdleProcess;

//time until...
static unsigned short TimeUntilNextDevice;
static unsigned short TimeUntilNextPeriodic;

//time ... started
static unsigned short TimeDeviceStarted;
static unsigned short TimePeriodicStarted;

//current time, but assigned in function calls
static unsigned short StableTCN;

//scheduling info
static bool DeviceWaiting;
static bool PeriodicWaiting;

static bool DeviceTimer;
static bool PeriodicTimer;

//global parameters used to get values into
//create interrupt handler
static void (*create_f)(void);
static int create_arg;
static unsigned int create_level;
static unsigned int create_n;
static Pid create_pid;

//os started flag
static bool OSStarted;

//forward declarations
static inline void ChooseNextTimer(void);
static inline void SetAndEnableNextTimer(void);

//yield interrupt trap
INTERRUPT_HANDLER(yield)
{
    unsigned short TimeDeviceTook;
    int temp;
    StableTCN = TCN;

    CurrentProcess->sp = sp;
if (CurrentProcess->level == DEVICE) {
  // double check this calculation for overflow errors
  // calculate how long the device took to run
  TimeDeviceTook = StableTCN - TimeDeviceStarted;

  // handle overflow wrap around
  if (StableTCN < TimeDeviceStarted)
    TimeDeviceTook -= 1;

  // re-insert into the delta queue
  YieldCurrentDeviceProcess(TimeDeviceTook);

  // update time of next device
  TimeUntilNextDevice = GetNextDeviceTime();

  // if there are periodic processes waiting
  if (PeriodicWaiting == TRUE) {
    // check if we missed the period end
    if (TimeDeviceTook >= TimeUntilNextPeriodic) {
      CurrentPeriodCompleted();
      // calculate end of next period
      temp = (int)(GetNextPeriodicTime()) -
        (int)(TimeDeviceTook - TimeUntilNextPeriodic);
      if (temp < 0)
        OS_Abort();
      else
        TimeUntilNextPeriodic = GetNextPeriodicTime() -
        (TimeDeviceTook - TimeUntilNextPeriodic);
    } else  // update time until end of period
      TimeUntilNextPeriodic -= TimeDeviceTook;

    // figure out if device or periodic comes next
    if (TimeUntilNextPeriodic < TimeUntilNextDevice) {
      PeriodicTimer = TRUE;
      DeviceTimer = FALSE;
    } else {
      PeriodicTimer = FALSE;
      DeviceTimer = TRUE;
    }
    TimePeriodicStarted = StableTCN;
  } else // no periodics waiting
  {
    PeriodicTimer = FALSE;
    DeviceTimer = TRUE;
  }
  SetAndEnableNextTimer();

  CurrentProcess = GetCurrentPeriodicProcess();
  if (CurrentProcess == NULL || CurrentProcess->level == IDLE_LEVEL)
    CurrentProcess = GetCurrentSporadicProcess();
} else if (CurrentProcess->level == PERIODIC)
YieldCurrentPeriodicProcess();
CurrentProcess = GetCurrentSporadicProcess();
} else if(CurrentProcess->level == SPORADIC)
{
    YieldCurrentSporadicProcess();
    CurrentProcess = GetCurrentSporadicProcess();
}

CurrentProcess->state = RUNNING;
RTI(CurrentProcess->sp);

//timer interrupt handler
INTERRUPT_HANDLER(timer)
{
    unsigned short TimeTook;
    StableTCN = TCN;

    CurrentProcess->sp = sp;
    if(CurrentProcess->level == PERIODIC)
        InterruptCurrentPeriodicProcess();
    else if(CurrentProcess->level == SPORADIC)
        InterruptCurrentSporadicProcess();

    //this timer is for the end of a period
    if(PeriodicTimer == TRUE)
    {
        //update device time
        if(DeviceWaiting == TRUE)
            TimeUntilNextDevice -= TimeUntilNextPeriodic;

        CurrentPeriodCompleted();
        TimePeriodicStarted = StableTCN;
        CurrentProcess = GetCurrentPeriodicProcess();
        TimeUntilNextPeriodic = GetNextPeriodicTime();
    }

    //this timer is for the start of a device
    else//(DeviceTimer == TRUE)
    {
        //update periodic time
        if(PeriodicWaiting == TRUE)
        {
            //calculate how long the device took to run
            TimeTook = StableTCN - TimePeriodicStarted;
            //handle overflow wrap around
            if(StableTCN < TimePeriodicStarted)
                TimeTook -= 1;
            //update time until next periodic
            TimeUntilNextPeriodic -= TimeTook;
        }

        //were running the device now, so it's delta should be zero
        TimeDeviceStarted = StableTCN;
        UpdateCurrentDeviceDelta(0);
        TimeUntilNextDevice = 0;

        CurrentProcess = GetCurrentDeviceProcess();

        IMR |= ISR_TMR;
    }
}
TSTAT &= 0;
//if not starting a device, setup the next timer
if(CurrentProcess->level != DEVICE)
{
    ChooseNextTimer();
    //SetAndEnableNextTimer();
    if(DeviceTimer == TRUE)
        TCMP = TCN + TimeUntilNextDevice;
    else //periodic timer
        TCMP = TCN + TimeUntilNextPeriodic;
}

CurrentProcess->state = RUNNING;
RTI(CurrentProcess->sp);
}

//process create interrupt trap
INTERRUPT_HANDLER(create)
{
    unsigned short TimeSinceDelta;
    int temp;
    StableTCN = TCN;
    if(OSStarted == TRUE)
    {
        CurrentProcess->sp = sp;
        if(create_level == DEVICE)
        {
            //figure out if we need to offset the rate of the device
            if(GetCurrentDeviceProcess() == NULL || OSStarted == FALSE)
            {
                TimeDeviceStarted = StableTCN;
                TimeSinceDelta = 0;
            }
            else
            {
                TimeSinceDelta = StableTCN - TimeDeviceStarted;
                //handle overflow wrap around
                if(StableTCN < TimeDeviceStarted)
                    TimeSinceDelta -= 1;
            }
        }
        //calculate ticks
        create_n *= TICKS_PER_MS;
        create_pid = AddDeviceProcess(create_f, create_arg, create_n,
                                        create_n + TimeSinceDelta);
        //update the timers
        if(create_pid != INVALIDPID && OSStarted == TRUE)
        {
            //update the device timer if needed
            if(CurrentProcess->level == DEVICE)
                TimeUntilNextDevice = 0;
            else if(DeviceWaiting == TRUE)
                TimeUntilNextDevice -= TimeSinceDelta;
            else   //else initialize the device timer
                TimeUntilNextDevice = (unsigned short)create_n;// +
            DeviceWaiting = TRUE;
        }
        if(PeriodicWaiting == TRUE)
        {
            //check if we missed the period end
            }
if(TimeSinceDelta >= TimeUntilNextPeriodic)
{
    CurrentPeriodCompleted();
    temp = (int)(GetNextPeriodicTime()) -
    (int)(TimeSinceDelta - TimeUntilNextPeriodic);
    if(temp < 0)
        OS_Abort();
    else
        TimeUntilNextPeriodic =
        GetNextPeriodicTime() - (TimeSinceDelta - TimeUntilNextPeriodic);
}
else
    TimeUntilNextPeriodic -= TimeSinceDelta;
}
ChooseNextTimer();
SetAndEnableNextTimer();
}
else if(create_level == PERIODIC)
{
    create_pid = AddPeriodicProcess(create_f, create_arg, create_n);
    //if you recreated a dead periodic and that is the current period
    //don't switch until the next period it should run
    /*
    if(create_pid != INVALIDPID && OSStarted == TRUE)
    {
        {
            CurrentProcess = GetCurrentPeriodicProcess();
        }
    }
    */
    else //if(create_level == SPORADIC)
    {
        create_pid = AddSporadicProcess(create_f, create_arg);
        if(create_pid != INVALIDPID && OSStarted == TRUE)
        {
            {
                CurrentProcess = GetCurrentSporadicProcess();
            }
        }
    }
    if(OSStarted == TRUE)
    {
        CurrentProcess->state = RUNNING;
        RTI(CurrentProcess->sp);
    }
    else
        RTI(sp);
}
void OS_Init()
InitBackEnd();
uart_init(BAUD_9600);

// vector table starts at address 0
ramvec = 0;

// install trap interrupt handler for yield
ramvec[33] = yield;

// install trap interrupt handler for create
ramvec[34] = create;

// setup timer interrupt
ramvec[30] = timer;
TCTL = 0x0115;   // timer clock source 1MHz
TPRER = 0x0063;   // prescalar 100
              // TPRER = 0x0009;   // prescalar 10

TimeUntilNextPeriodic = 0;
TimeUntilNextDevice = 0;
TimeDeviceStarted = 0;
TimePeriodicStarted = 0;

DeviceWaiting = FALSE;
PeriodicWaiting = FALSE;

DeviceTimer = FALSE;
PeriodicTimer = FALSE;

CurrentProcess = NULL;
OSStarted = FALSE;

void OS_Start()
{
    OSStarted = TRUE;

    // get the first periodic
    CurrentProcess = GetCurrentPeriodicProcess();
    // if we are idle and there is a sporadic, get it
    if(CurrentProcess == NULL || CurrentProcess->level == IDLE_LEVEL)
        CurrentProcess = GetCurrentSporadicProcess();

    // device process will never be the first process to run

    // see if we need a timer for devices
    if(GetCurrentDeviceProcess() != NULL)
    {
        DeviceWaiting = TRUE;
        TimeUntilNextDevice = GetNextDeviceTime();
    }

    // see if we need a timer for periodics
    if(GetCurrentPeriodicProcess() != NULL)
    {
        PeriodicWaiting = TRUE;
        TimeUntilNextPeriodic = GetNextPeriodicTime();
    }

    // figure out which timer comes first
    ChooseNextTimer();
}
//use to start near timer overflow
//while(TCN < 63500 || TCN > 64500);

CurrentProcess->state = RUNNING;
StableTCN = TCN;
SetAndEnableNextTimer();

TimeDeviceStarted = StableTCN;
TimePeriodicStarted = TimeDeviceStarted;

//goto the first process
RTI(CurrentProcess->sp);

}
/if there are periodic processes waiting
if(PeriodicWaiting == TRUE)
{
    //check if we missed the period end
    //if yes, update time until next end of period
    if(TimeDeviceTook >= TimeUntilNextPeriodic)
    {
        CurrentPeriodCompleted();
        temp = (int)(GetNextPeriodicTime()) -
            (int)(TimeDeviceTook - TimeUntilNextPeriodic);
        if(temp < 0)
            OS_Abort();
        else
            TimeUntilNextPeriodic =
                GetNextPeriodicTime() - (TimeDeviceTook - TimeUntilNextPeriodic);
    }
    else  //update time until end of period
        TimeUntilNextPeriodic -= TimeDeviceTook;
}
ChooseNextTimer();

//get the next process
CurrentProcess = GetCurrentPeriodicProcess();
if(CurrentProcess == NULL || CurrentProcess->level == IDLE_LEVEL)
    CurrentProcess = GetCurrentSporadicProcess();
SetAndEnableNextTimer();
}
else if(CurrentProcess->level == PERIODIC)
{
    RemoveCurrentPeriodicProcess();
    //because the current periodic is now idle
    CurrentProcess = GetCurrentSporadicProcess();
}
else if(CurrentProcess->level == SPORADIC)
{
    RemoveCurrentSporadicProcess();
    CurrentProcess = GetCurrentSporadicProcess();
}
//if idle wants to terminate... the OS is done
else// if(CurrentProcess->level == IDLE_LEVEL)
{
    OS_Abort();
}
CurrentProcess->state = RUNNING;
EI();
RTI(CurrentProcess->sp);
}

void OS_Yield(void)
{
    //goto the yield interrupt trap
    asm ("trap #1");
}

int OS_GetParam(void)
{
    return CurrentProcess->arg;
}
FIFO OS_InitFiFo()
{
    FIFO new_fifo;
    DI();
    new_fifo = InitFifo();
    EI();
    return new_fifo;
}

void OS_Write( FIFO f, int val )
{
    DI();
    WriteFifo(f, val);
    EI();
}

bool OS_Read( FIFO f, int *val )
{
    bool ReadStatus;
    DI();
    ReadStatus = ReadFifo(f, val);
    EI();
    return ReadStatus;
}

// uses DeviceWaiting, PeriodicWaiting, TimeUntilNextDevice, and //
// TimeUntilNextPeriodic to decide which of the two timer flags to set: //
// devicetimer or periodictimer or neither
static inline void ChooseNextTimer(void)
{
    // figure out which comes first
    if(DeviceWaiting == TRUE & PeriodicWaiting == TRUE)
    {
        // if the period switch comes before the device
        if(TimeUntilNextPeriodic < TimeUntilNextDevice)
        {
            PeriodicTimer = TRUE;
            DeviceTimer = FALSE;
        } else
        {
            PeriodicTimer = FALSE;
            DeviceTimer = TRUE;
        }
    } else
    { // use the timer for the device
        if(DeviceWaiting == TRUE)
        {
            DeviceTimer = TRUE;
            PeriodicTimer = FALSE;
        } else
        { // use the timer for the periodic
            if(PeriodicWaiting == TRUE)
            {
                DeviceTimer = FALSE;
                PeriodicTimer = TRUE;
            } else
            {
                DeviceTimer = FALSE;
            }
        }
    }
}
PeriodicTimer = FALSE;
}
}

static inline void SetAndEnableNextTimer(void)
{
  // clear any previous timer event
  TSTAT &= 0;

  // set time of next device
  if(DeviceTimer == TRUE)
  {
    // clear any previous timer event
    // TSTAT &= 0;
    // TCMP = StableTCN + TimeUntilNextDevice;
    TCMP = TCN + TimeUntilNextDevice;
    // enable timer - only affects timer mask bit!
    IMR &= ~ISR_TMR;
  }

  // set time of next period
  else if(PeriodicTimer == TRUE)
  {
    // clear any previous timer event
    // TSTAT &= 0;
    // TCMP = StableTCN + TimeUntilNextPeriodic;
    TCMP = TCN + TimeUntilNextPeriodic;
    // enable timer - only affects timer mask bit!
    IMR &= ~ISR_TMR;
  }
  else
  {
    // clear any previous timer event
    // TSTAT &= 0;
    IMR |= ISR_TMR;
  }
}

///////////////////////// backend.h
// backend.h Contains definitions of all the data structures and
// functions used to manipulate them

#ifndef __backend_h
#define __backend_h
#include "os.h"

#define TICKS_PER_MS 10   // 10kHz
#define IDLE_LEVEL 4

typedef enum {READY, RUNNING, YIELD, DEAD} ProcessStateEnum;

// typedefs for Process Descriptor struct
struct ProcessDescriptor
{  
    ProcessStateEnum state; // process state  
    unsigned long sp; // stack pointer  
    unsigned char* wksp; // process workspace  
    Pid id; // id of the process  
    unsigned int level; // the type of process  
    int arg; // argument for the process  
    unsigned int n; // the extra argument for the process  
    unsigned int deltatime; // used by device process delta queue  
  }

// next process descriptor in list
// used by device and sporadic process lists
struct ProcessDescriptor* nextPD;

typedef struct ProcessDescriptor ProcessDescriptor;
typedef struct ProcessDescriptor* ProcessDescriptorPTR;

// Initializes the back end
void InitBackEnd(void);

// Returns the current device process, or NULL if none
ProcessDescriptorPTR GetCurrentDeviceProcess(void);

// Returns the current periodic process, idle if none, or NULL if PPPLen = 0
ProcessDescriptorPTR GetCurrentPeriodicProcess(void);

// Returns the current sporadic process, or idle if none
ProcessDescriptorPTR GetCurrentSporadicProcess(void);

// Returns the time in ticks until next device should be run
unsigned short GetNextDeviceTime(void);

// Returns the time in ticks until the periodic process must change
unsigned short GetNextPeriodicTime(void);

// Completes the period and updates the current periodic process
void CurrentPeriodCompleted(void);

// Adds a device process using function f, argument arg, rate, and start rate
// is the rate that will be used for the first delta
// rate and startrate are in Ticks
// It inserts the process into the delta queue
// Returns INVALIDPID if it couldn't create it
Pid AddDeviceProcess(void (*f)(void), int arg, unsigned int rate,
                      unsigned int startRate);

// Removes the current device process from the delta queue and updates the
deltas
// Subtracting offset (Ticks) from them
void RemoveCurrentDeviceProcess(unsigned int offset);

// Manually changes the delta of the current device process to d (ticks)
void UpdateCurrentDeviceDelta(unsigned int d);

// Updates the current device process and stops the current one.
// Updates the deltas in the delta queue by subtracting offset (ticks) from them
void YieldCurrentDeviceProcess(unsigned short offset);

// Adds a periodic process for the PPP to run. f is the function that will be
// used, arg is the argument, and name is the name of the periodic
returns INVALIDPID if it couldn't create it
Pid AddPeriodicProcess(void (*f)(void), int arg, unsigned int name);

removes the current periodic process and stops PPP from running it
void RemoveCurrentPeriodicProcess(void);

updates the state of the current periodic process
void YieldCurrentPeriodicProcess(void);

updates the state of the current periodic process
void InterruptCurrentPeriodicProcess(void);

adds a sporadic process to the sporadic queue. f is the function that is
used, and arg is the argument
returns INVALIDPID if it couldn't create it
Pid AddSporadicProcess(void (*f)(void), int arg);

removes the current sporadic process from the sporadic queue
void RemoveCurrentSporadicProcess(void);

moves the current sporadic process to the end of the sporadic queue
void YieldCurrentSporadicProcess(void);

updates the state of the current sporadic process
void InterruptCurrentSporadicProcess(void);

creates a new FIFO
returns INVALIDFIFO if it couldn't create one
FIFO InitFifo(void);

writes a value to the specified FIFO
void WriteFifo(FIFO f, int val);

reads a value from the specified FIFO
returns true if there is anything in the FIFO else false
bool ReadFifo(FIFO f, int* val);

#include "backend.h"
#include "newsystem.h"
#include <string.h> //for NULL

MAXPROCESS needs to be less than MEM_BLOCK_NUM - 1
WORKSPACE needs to be less than MEM_BLOCK_SIZE

externs of scheduling info
extern int PPPLen;
extern int PPP[];
extern int PPPMax[];

Idle Process
struct ProcessDescriptor IdleProcess;

typedef for FIFODescriptor
typedef struct
{
    int ReadIndex;
}
int WriteIndex;
icnt Data[FIFOSIZE];
icnt CurrentSize;
} FIFODescriptor, *FIFODescriptorPTR;

//Processes
static ProcessDescriptor Processes[MAXPROCESS];

//FIFO's
static FIFODescriptor FIFOs[MAXFIFO];

//Pointers for Device process list
static ProcessDescriptorPTR CurrentDevice;
static ProcessDescriptorPTR LastDevice;

//map for converting a periodic name into a process pointer
static ProcessDescriptorPTR PeriodicMap[MAXPROCESS];

//the current periodic process that is being run
static ProcessDescriptorPTR CurrentPeriodic;

//pointers for sporadic process list
static ProcessDescriptorPTR CurrentSporadic;
static ProcessDescriptorPTR LastSporadic;

//next pid for creating processes
static unsigned int NextPid;

//current place OS is in the PPP
static unsigned int CurrentPPPIndex;

//Number of live processes
static unsigned int NumProcesses;

//forward declarations
static void InitProcessDescriptor(ProcessDescriptorPTR pd, void (*f)(void),
icnt arg, unsigned int level, unsigned int n);
static int FindEmptyProcessDescriptor(void);

//function use for the Idle process
//returns when OS should finish
static void IdleFunction(void)
{
do
{
//lcd_putchar('P');
//OS_Yield();
//do something useful
//calculate pi or something...
//get the OS to quit if nothing is running
}while(NumProcesses > 0);
while(1);

//Initializes the back end
void InitBackend(void)
{
int x;

//setup the idle process
IdleProcess.wksp = MemBlock[MAXPROCESS];
InitProcessDescriptor(&IdleProcess, IdleFunction, 0, IDLE_LEVEL, 0);
IdleProcess.n = IDLE;

//set the whole PPPMap to NULL
for(x = 0; x < MAXPROCESS; x++)
    PeriodicMap[x] = NULL;

//set all of the processes to DEAD and give them there workspace
for(x = 0; x < MAXPROCESS && x < MEM_BLOCK_NUM; x++)
{
    Processes[x].state = DEAD;
    Processes[x].wksp = MemBlock[x];
}

//set all of the FIFOs to available
for(x = 0; x < MAXFIFO; x++)
{
    FIFOs[x].ReadIndex = -1;
    FIFOs[x].WriteIndex = -1;
    FIFOs[x].CurrentSize = 0;
}

//setup other data
NumProcesses = 0;
CurrentDevice = NULL;
CurrentSporadic = NULL;
CurrentPeriodic = NULL;
LastDevice = NULL;
LastSporadic = NULL;
NextPid = INVALIDPID + 1;
CurrentPPPIndex = 0;

//returns the current device process, or NULL if none
ProcessDescriptorPTR GetCurrentDeviceProcess(void)
{
    return CurrentDevice;
}

//returns the current periodic process, idle if none, or NULL if PPPLen = 0
ProcessDescriptorPTR GetCurrentPeriodicProcess(void)
{
    if(CurrentPeriodic != NULL && CurrentPeriodic->state == YIELD)
        return &IdleProcess;
    else
        return CurrentPeriodic;
}

//returns the current sporadic process, or idle if none
ProcessDescriptorPTR GetCurrentSporadicProcess(void)
{
    if(CurrentSporadic != NULL)
        return CurrentSporadic;
    else
        return &IdleProcess;
}

//returns the time in ticks until next device should be run
unsigned short GetNextDeviceTime(void)
{
    if(CurrentDevice != NULL)
        return (unsigned short)CurrentDevice->deltatime;

else
    return 0;
}
//returns the time in ticks until the periodic process must change
unsigned short GetNextPeriodicTime(void)
{
    if(PPPLen > 0)
        return TICKS_PER_MS * (unsigned short)PPPMax[CurrentPPPIndex];
    else
        return 0;
}

//completes the period and updates the current periodic process
void CurrentPeriodCompleted(void)
{
    int name;
    if(PPPLen > 0)
    {
        name = PPP[CurrentPPPIndex];
        if(name != IDLE && PeriodicMap[name] != NULL)
            PeriodicMap[name] = READY;
        CurrentPPPIndex = (CurrentPPPIndex + 1) % PPPLen;
        name = PPP[CurrentPPPIndex];
        if(name != IDLE && PeriodicMap[name] != NULL)
            CurrentPeriodic = PeriodicMap[name];
        else
            CurrentPeriodic = &IdleProcess;
    }
}

//adds a device process using function f, argument arg, rate, and start rate
//rate and startrate are in Ticks
//it inserts the process into the delta queue
//returns INVALIDPID if it couldn't create it
Pid AddDeviceProcess(void (*f)(void), int arg, unsigned int rate,
    unsigned int startrate)
{
    ProcessDescriptorPTR pdPTR, prev, next;
    int index = FindEmptyProcessDescriptor();
    if(index != -1)
    {
        pdPTR = &Processes[index];
        InitProcessDescriptor(pdPTR, f, arg, DEVICE, rate);
        pdPTR->deltatime = startrate;
        NumProcesses++;
        if(CurrentDevice != NULL)
        {
            prev = NULL; //set the previous node to NULL
            //set the next node to the current node
            next = CurrentDevice;
            //while there is another node and the time is still before us
            while(next != NULL && next->deltatime <= pdPTR->deltatime)
            {
                //subtract the time of the device process coming before us
                pdPTR->deltatime -= next->deltatime;
                //update the previous and current nodes
                prev = next;
            }
            //now that all the nodes have been updated
            //we need to insert this new node
            //we do this by setting the next node to the new node
            //and moving the previous node to the new previous node
            next = pdPTR;
            next->prev = prev;
            //set the previous node to the current node
            prev = NULL;
            //set the next node to the current node
            next = CurrentDevice;
            //while there is another node and the time is still before us
            while(next != NULL && next->deltatime <= pdPTR->deltatime)
            {
                //subtract the time of the device process coming before us
                pdPTR->deltatime -= next->deltatime;
                //update the previous and current nodes
                prev = next;
            }
        }
    }
}
next = next->nextPD;
}

//if there is a previous node, insert
if(prev != NULL)
    prev->nextPD = pdPTR;
else
    CurrentDevice = pdPTR;

//update the next ptr for the inserted node
pdPTR->nextPD = next;

//update the delta time if a node comes after us
if(next != NULL)
    next->deltatime -= pdPTR->deltatime;
else
    LastDevice = pdPTR;
else
{

CurrentDevice = pdPTR;
LastDevice = pdPTR;
}
return pdPTR->id;
}
else
return INVALIDPID;
}

//removes the current device process from the delta queue and updates the
//deltas subtracting offset (Ticks) from them
void RemoveCurrentDeviceProcess(unsigned int offset)
{
    int delta;
    ProcessDescriptorPTR pd = CurrentDevice;
    CurrentDevice->state = DEAD;
    NumProcesses--;

    //if this is the last device process
    if(LastDevice == CurrentDevice)
        LastDevice = NULL;

    delta = CurrentDevice->deltatime + offset;

    //go through the list updating the delta times
    while(pd->nextPD != NULL && delta > 0)
    {
        pd = pd->nextPD;
        if(pd->deltatime < delta)
        {
            delta -= pd->deltatime;
            pd->deltatime = 0;
        }
        else
        {
            pd->deltatime -= delta;
            delta = 0;
        }
    }

    CurrentDevice = CurrentDevice->nextPD;
}
//manually changes the delta of the current device process to d (ticks)
void UpdateCurrentDeviceDelta(unsigned int d)
{
    if(CurrentDevice != NULL)
        CurrentDevice->deltatime = d;
}

//Updates the current device process and stops the current one.
//updates the deltas in the delta queue by subtracting offset (ticks) from them
void YieldCurrentDeviceProcess(unsigned short offset)
{
    ProcessDescriptorPTR pd, prev, next;

    CurrentDevice->state = READY;
    //set the deltatime to the rate
    CurrentDevice->deltatime = CurrentDevice->n;

    if(CurrentDevice != LastDevice)
    {
        pd = CurrentDevice;
        prev = NULL;//set the previous node to NULL
        //set the next node to the current node
        next = CurrentDevice->nextPD;

        if(next != NULL)
            CurrentDevice = next;

        //while there is another node and the time is still before us
        while(next != NULL && next->deltatime <= pd->deltatime)
        {
            //subtract the time of the device process coming before us
            pd->deltatime -= next->deltatime;

            //update the previous and current nodes
            prev = next;
            next = next->nextPD;
        }

        //if there is a previous node, insert
        if(prev != NULL)
            prev->nextPD = pd;
        else       //else this is the front of the list
            CurrentDevice = pd;

        //update the next ptr for the inserted node
        pd->nextPD = next;

        //update the delta time if a node comes after us
        if(next != NULL)
            next->deltatime -= pd->deltatime;
        else
            LastDevice = pd;
    }

    pd = CurrentDevice;

    //go through the list updating the delta times
    while(pd != NULL && offset > 0)
    {
        if(pd->deltatime < offset)
            offset -= pd->deltatime;
        pd->deltatime = 0;
    }
else
{
    pd->deltatime -= offset;
    offset = 0;
}
}
pd = pd->nextPD;
}

//adds a periodic process for the PPP to run. f is the function that will be
//used, arg is the argument, and name is the name of the periodic
//returns INVALIDPID if it couldn't create it
Pid AddPeriodicProcess(void (*f)(void), int arg, unsigned int name)
{
    int index;

    //is the name in the PPP?
    for(index = 0; index < PPPLen; index++)
    {
        if(PPP[index] == name)
            break;
    }

    //name not found...
    if(index == PPPLen)
        return INVALIDPID;

    //make sure we have room and the periodic doesn't already exist
    if(PPPLen > 0 && name < MAXPROCESS && PeriodicMap[name] == NULL)
    {
        //make sure it's in the PPP
        for(index = 0; index < PPPLen; index++)
        {
            if(PPP[index] == name)
            {
                index = -1;
                break;
            }
        }

        if(index == -1)
        {
            ProcessDescriptorPTR pdPTR;

            //find a place for it
            index = FindEmptyProcessDescriptor();
            if(index != -1)
            {
                pdPTR = &Processes[index];
                InitProcessDescriptor(pdPTR, f, arg, PERIODIC, name);
                NumProcesses++;
                PeriodicMap[name] = pdPTR;
                index = PPP[CurrentPPPIndex];
                if(index != IDLE)
                    CurrentPeriodic = PeriodicMap[index];
                else
                    CurrentPeriodic = &IdleProcess;
                pdPTR->id;
            }
            else
                return INVALIDPID;
        }
    }

    return pdPTR->id;
}
else
    return INVALIDPID;
}

//removes the current periodic process and stops PPP from running it
void RemoveCurrentPeriodicProcess(void)
{
    CurrentPeriodic->state = DEAD;
    NumProcesses--; 
    PeriodicMap[CurrentPeriodic->n] = NULL;
    CurrentPeriodic = &IdleProcess;
}

//updates the state of the current periodic process
void YieldCurrentPeriodicProcess(void)
{
    CurrentPeriodic->state = YIELD;
}

//updates the state of the current periodic process
void InterruptCurrentPeriodicProcess(void)
{
    CurrentPeriodic->state = READY;
}

//adds a sporadic process to the sporadic queue. f is the function that is
//used, and arg is the argument
//returns INVALIDPID if it couldn't create it
Pid AddSporadicProcess(void (*f)(void), int arg)
{
    int index;
    ProcessDescriptorPTR pdPTR;

    index = FindEmptyProcessDescriptor();
    if(index != -1)
    {
        pdPTR = &Processes[index];
        InitProcessDescriptor(pdPTR, f, arg, SPORADIC, 0);
        NumProcesses++;
        if(CurrentSporadic != NULL)
        {
            LastSporadic->nextPD = pdPTR;
            LastSporadic = pdPTR;
        }
        else
        {
            CurrentSporadic = pdPTR;
            LastSporadic = pdPTR;
        }
        return pdPTR->id;
    }
    else
        return INVALIDPID;
}

//removes the current sporadic process from the sporadic queue
void RemoveCurrentSporadicProcess(void)
{
    CurrentSporadic->state = DEAD;
}
NumProcesses--;  //if this is the last process update the last index
if(LastSporadic == CurrentSporadic)
    LastSporadic = NULL;
CurrentSporadic = CurrentSporadic->nextPD;

//moves the current sporadic process to the end of the sporadic queue
void YieldCurrentSporadicProcess(void)
{
    ProcessDescriptorPTR temp;
    CurrentSporadic->state = READY;
    //if it's not the only one in the list put it to the end
    if(CurrentSporadic->nextPD != NULL)
    {
        //move it to the end of the list
        LastSporadic->nextPD = CurrentSporadic;
        LastSporadic = CurrentSporadic;

        //set it's next index to NULL
        temp = CurrentSporadic->nextPD;
        CurrentSporadic->nextPD = NULL;

        //update the first index
        CurrentSporadic = temp;
    }
}

//updates the state of the current sporadic process
void InterruptCurrentSporadicProcess(void)
{
    CurrentSporadic->state = READY;
}

//creates a new FIFO
//returns INVALIDFIFO if it couldn't create one
FIFO InitFifo(void)
{
    int fifo = INVALIDFIFO;
    int x = 0;

    while(x < MAXFIFO)
    {
        if(FIFOs[x].ReadIndex == -1)
        {
            fifo = x;
            FIFOs[x].ReadIndex = 0;
            FIFOs[x].WriteIndex = 0;
            break;
        }
        else
        {
            x++;
        }
    }

    return fifo;
}

//writes a value to the specified FIFO
void WriteFifo(FIFO f, int val)
FIFODescriptorPTR fifoPTR = &FIFOs[f];
fifoPTR->Data[fifoPTR->WriteIndex] = val;
fifoPTR->WriteIndex = (fifoPTR->WriteIndex + 1) % FIFOSIZE;
if(fifoPTR->CurrentSize == FIFOSIZE)
    fifoPTR->ReadIndex = (fifoPTR->ReadIndex + 1) % FIFOSIZE;
else
    fifoPTR->CurrentSize++;
}

//reads a value from the specified FIFO
//returns true if there is anything in the FIFO else false
bool ReadFifo(FIFO f, int* val)
{
    FIFODescriptorPTR fifoPTR = &FIFOs[f];

    if(fifoPTR->CurrentSize == 0)
        return FALSE;
    else
    {
        *val = fifoPTR->Data[fifoPTR->ReadIndex];
        fifoPTR->ReadIndex = (fifoPTR->ReadIndex + 1) % FIFOSIZE;
        fifoPTR->CurrentSize--;
        return TRUE;
    }
}

//initializes all of the members of the process descriptor
//except nextPDindex and wksp and deltatime
static void InitProcessDescriptor(ProcessDescriptorPTR pd, void (*f)(void),
    int arg, unsigned int level, unsigned int n)
{
    pd->state = READY;
    pd->sp = (unsigned long) &(pd->wksp[WORKSPACE]);

    // exit point
    pd->sp -= 4;
    *((unsigned long*) pd->sp) = (unsigned long) OS_Terminate;

    // PC
    pd->sp -= 4;
    *((unsigned long*) pd->sp) = (unsigned long)f;

    // SR register
    pd->sp -= 2;
    *((unsigned short*) pd->sp) = 0x2000;

    // general purpose registers
    pd->sp -= 56;
    memset((void*) pd->sp, 0, 56);
    pd->id = NextPid++;
    pd->level = level;
    pd->arg = arg;
    pd->n = n;
    pd->nextPD = NULL;
}

//returns the index of the first empty process descriptor
static int FindEmptyProcessDescriptor(void)
{

int x = MAXPROCESS;
while(x >= 0)
{
    if(Processes[x].state == DEAD)
        break;
    x--;
}
return x;

A.2 68HC11 Code

/////////////////////////////////////////////////////////////////////
// rproj5.c
// Initializes the OS and creates the first process.
// Contains scheduling definitions for the periodic processes.

#include "hc11.h"
#include "os.h"
#include "rcoord.h"
#include "sched.h"

#define RESET *(void(**)())(0xFFFE)
extern void _start(void);

//create the scheduling info here
int PPPLen = NUM_PERIODICS + 1;
int PPP[] = {IDLE,TAPE_SEN_NUM,DRIVING_NUM,COIN_BUMPER_NUM,SERIAL_WRITE_NUM};
int PPPMax[] = {3,TAPE_SEN_PERIOD,DRIVING_PERIOD,COIN_BUMPER_PERIOD,SERIAL_WRITE_PERIOD};

int main(void)
{
    RESET = _start;

    OS_DI();
    OS_Init();

    if(OS_Create(Coordinator, 0, SPORADIC, 0) == INVALIDPID)
        return 0;

    OS_Start();
    return 1;
}

/////////////////////////////////////////////////////////////////////
// sched.h
// defines the name and max CPU time of each periodic process

#define NUM_PERIODICS  4
#define DRIVING_NUM   0
#define DRIVING_PERIOD  3

Autonomous Mobile Robot Design and Implementation  A-28  CENG 499
#define TAPE_SEN_NUM 1
#define TAPE_SEN_PERIOD 4

#define COIN_BUMPER_NUM 2
#define COIN_BUMPER_PERIOD 3

#define SERIAL_WRITE_NUM 3
#define SERIAL_WRITE_PERIOD 3

// serial reading is interrupt driven
// coordination process is sporadic
// motors and servos are interrupt driven

// rcomm.h

#ifndef __rcomm_h
#define __rcomm_h

#include "os.h"

bool setupComm(FIFO in, FIFO out);
#endif

// rcomm.c

#include "hc11.h"
#include "rcomm.h"
#include "os.h"
#include "sched.h"
#include <stdio.h>

void ReadSerial(void);
void WriteSerial(void);

static FIFO rRxFIFO;
static FIFO rTxFIFO;

bool setupComm(FIFO in, FIFO out)
{
    // setup SCI ISR to ReadSerial
    *(void (**)) (0xFFD6) = ReadSerial; // address of interrupt handler

    // 9600 BAUD
    BAUD = 0x30;

    // use default values of SCCR1
    // enable transmitter and receiver, enable receiver interrupts
    SCCR2 = 0x2C;

    rRxFIFO = in;
    rTxFIFO = out;

    if(OS_Create(WriteSerial, out, PERIODIC, SERIAL_WRITE_NUM) == INVALIDPID)
        return FALSE;

    return TRUE;
}
#pragma interrupt_handler ReadSerial()
void ReadSerial(void)
{
    char received;
    char RxStatus;

    //check NF and FE flags?? OR flag?
    RxStatus = SCSR;

    //assumes SCI interrupt was generated by RDRF flag
    received = SCDR;

    OS_Write(rRxFIFO, (int) received);
}

void WriteSerial(void)
{
    unsigned char LowByte;
    int write;

    while(1)
    {
        if(OS_Read(rTxFIFO, &write) == TRUE)
        {
            LowByte = (unsigned char) write;
            //SCDR = LowByte;
            putchar(LowByte);
        }
        OS_Yield();
    }
}

/comments----------------------------------
// rcoord.h
//
 ifndef __rcoord_h
#define __rcoord_h
#include "os.h"
#include "rcomm.h"
#include "driving.h"
#include "coinbump.h"
#include ".\messages.h"
#include "buzzer.h"
/*static*/ FIFO CommIn = INVALIDFIFO;
/*static*/ FIFO CommOut = INVALIDFIFO;
/*static*/ FIFO DrivingIn = INVALIDFIFO;
/*static*/ FIFO DrivingOut = INVALIDFIFO;
FIFO CoinOut = INVALIDFIFO;
FIFO BumpOut = INVALIDFIFO;

void Coordinator(void)
{
    int commread, drivingread;
    int coinread, bumpread;
    bool handling_message, have_coin;

    CommIn = OS_InitFiFo();
    CommOut = OS_InitFiFo();
    if(CommIn == INVALIDFIFO || CommOut == INVALIDFIFO)
        return;
    if(setupComm(CommIn, CommOut) == FALSE)
        return;

    DrivingIn = OS_InitFiFo();
    DrivingOut = OS_InitFiFo();
    if(DrivingIn == INVALIDFIFO || DrivingOut == INVALIDFIFO)
        return;
    if(setupDriving(DrivingIn, DrivingOut) == FALSE)
        return;

    CoinOut = OS_InitFiFo();
    BumpOut = OS_InitFiFo();
    if(CoinOut == INVALIDFIFO || BumpOut == INVALIDFIFO)
        return;
    if(setupCoinBump(CoinOut, BumpOut) == FALSE)
        return;

    handling_message = FALSE;
    have_coin = FALSE;

    OS_Write(CommOut, R_P_RESET);

    while(1)
    {
        if(OS_Read(CommIn, &commread) == TRUE)
        {
            if(handling_message == TRUE)
            {
                if(commread == P_R_STOP_DRIVE)
                    OS_Write(DrivingIn, C_D_STOP);
            }
            else
            {
                switch(commread)
                {
                    case P_R_NO_MESSAGE:
                        break;

                    case P_R_GO_FORWARD_TO_INT:
                        handling_message = TRUE;
                        OS_Write(DrivingIn, C_D_GO_FORWARD_TO_INT);
                        break;
                }
            }
        }
    }
}
case P_R.BACKUP_FROM_INT:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_BACKUP_FROM_INT);
    break;

case P_R.BACKUP_FROM_MID:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_BACKUP_FROM_MID);
    break;

case P_R.TURN_LEFT:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_TURN_LEFT);
    break;

case P_R.TURN_RIGHT:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_TURN_RIGHT);
    break;

case P_R.RESUME:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_RESUME);
    break;

case P_R.STOP_DRIVE:
    OS_Write(CommOut, R_P_STOPPED);
    break;

case P_R.CRY:
    playBuzzer(1);
    OS_Write(CommOut, R_P_CRY_DONE);
    break;

case P_R.DROP_COIN_LEFT:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_DROP_COIN_LEFT);
    break;

case P_R.DROP_COIN_RIGHT:
    handling_message = TRUE;
    OS_Write(DrivingIn, C_D_DROP_COIN_RIGHT);
    break;

if(OS_Read(DrivingOut, &drivingread) == TRUE)
{
    switch(drivingread)
    {
        case D_C_NO_MESSAGE:
            break;

        case D_C_AT_INTERSECTION:
            handling_message = FALSE;
            if(have_coin == TRUE)
                OS_Write(CommOut, R_P_AT_INT_WITH_COIN);
            else
                OS_Write(CommOut, R_P_AT_INT_WITHOUT_COIN);
            break;
    }
}
case D_C_TURN_COMPLETE:
    handling_message = FALSE;
    OS_Write(CommOut, R_P_TURN_COMPLETE);
    break;

case D_C_STOPPED:
    handling_message = FALSE;
    OS_Write(CommOut, R_P_STOPPED);
    break;

case D_C_LOST:
    handling_message = FALSE;
    OS_Write(CommOut, R_P_LOST);
    break;
}
}

if(OS_Read(CoinOut, &coinread) == TRUE)
{
    switch(coinread)
    {
    case COIN:
        have_coin = TRUE;
        break;

    case NO_COIN:
        have_coin = FALSE;
        break;
    }
}

if(OS_Read(BumpOut, &bumpread) == TRUE)
{
    switch(bumpread)
    {
    case BUMP:
        OS_Write(CommOut, R_P_BUMPER_IN);
        break;

    case NO_BUMP:
        OS_Write(CommOut, R_P_BUMPER_OUT);
        break;
    }
}

OS_Yield();

}
#include "os.h"

//messages
#define C_D_NO_MESSAGE   0
#define C_D_GO_FORWARD_TO_INT 1
#define C_D_BACKUP_FROM_INT 2
#define C_D_BACKUP_FROM_MID 3
#define C_D_TURN_LEFT    4
#define C_D_TURN_RIGHT   5
#define C_D_TURN_COMPLETE 6
#define C_D_STOP        7
#define C_D_RESUME      8
#define C_D_STOPPED     9
#define C_D_DROP_COIN_LEFT   10
#define C_D_DROP_COIN_RIGHT 11

//replies
#define D_C_NO_MESSAGE   0
#define D_C_AT_INTERSECTION 1
#define D_C_TURN_COMPLETE 2
#define D_C_STOPPED     3
#define D_C_STOPPED     4

bool setupDriving(FIFO in, FIFO out);

.handleSubmit()

//////////////////////////////////////////////////////////////////////////
// driving.c
// Cointains the code for the driving process
// Whenever a new message is received, the process makes a function call
to execute the appropriate action

#include "driving.h"
#include "os.h"
#include "tapesen.h"
#include "sched.h"

FIFO Din = INVALIDFIFO;
FIFO Dout = INVALIDFIFO;
FIFO Tapein = INVALIDFIFO;
FIFO Tapeout = INVALIDFIFO;

void DrivingProcess(void);

bool goForwardToIntersection(bool* reset, int message, FIFO Coordout, FIFO Tapein, FIFO Tapeout);
bool turnLeft(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout);
bool turnRight(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout);
bool dropCoinLeft(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout);
bool dropCoinRight(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout);

bool setupDriving(FIFO in, FIFO out)
{
    Din = in;
    Dout = out;

    if(OS_Create(DrivingProcess, 0, PERIODIC, DRIVING_NUM) == INVALIDPID)
        return FALSE;

    Tapein = OS_InitFiFo();
    Tapeout = OS_InitFiFo();
    if(Tapein == INVALIDFIFO || Tapeout == INVALIDFIFO)
void DrivingProcess(void)
{
    int message = C_D_NO_MESSAGE;
    int state = C_D_NO_MESSAGE;
    bool reset = FALSE;
    bool stopped = FALSE;
    int stopped_state = C_D_NO_MESSAGE;

    initMotors();

    while(1)
    {
        //check for message and update behavior
        if(OS_Read(Din, &message) == TRUE)
        {
            if(stopped == TRUE && message == C_D_RESUME)
            {
                stopped = FALSE;
                state = stopped_state;
                stopped_state = C_D_NO_MESSAGE;
            }
            else if(message == C_D_STOP)
            {
                stopped_state = state;
                stopped = TRUE;
            }
            else if(state == C_D_NO_MESSAGE)
            {
                state = message;
                message = C_D_NO_MESSAGE;
                reset = TRUE;
                stopped = FALSE;
                stopped_state = C_D_NO_MESSAGE;
            }
        }
    }

    switch(state)
    {
    case C_D_NO_MESSAGE:
        //if(message == C_D_STOP)
        //    OS_Write(Dout, D_C_STOPPED);
        break;
    case C_D_GO_FORWARD_TO_INT:
        if(goForwardToIntersection(&reset, message, Dout, Tapein, Tapeout) == TRUE)
            state = C_D_NO_MESSAGE;
        break;
    case C_D_TURN_LEFT:
        if(turnLeft(&reset, message, Dout, Tapein, Tapeout) == TRUE)
state = C_D_NO_MESSAGE;
break;

case C_D_TURN_RIGHT:
    if(turnRight(&reset, message, Dout, Tapein, Tapeout) == TRUE)
        state = C_D_NO_MESSAGE;
    break;

case C_D_DROP_COIN_LEFT:
    if(dropCoinLeft(&reset, message, Dout, Tapein, Tapeout) == TRUE)
        state = C_D_NO_MESSAGE;
    break;

case C_D_DROP_COIN_RIGHT:
    if(dropCoinRight(&reset, message, Dout, Tapein, Tapeout) == TRUE)
        state = C_D_NO_MESSAGE;
    break;
}
message = C_D_NO_MESSAGE;
OS_Yield();
}

////////////////////////////////////////////////////////////////////
// gofwd.c
// Contains the code to drive the robot forward along the tape track
// Requests readings from the tapesen via FIFO to determine its
// current location.  Tries to smoothly curve back on when it
// starts to go off the tape

#include "os.h"
#include "motors.h"
#include "tapesen.h"
#include "driving.h"

#define ARC_RATIO  4
#define ARC_INCREASE 3

static unsigned int collision_time[2];
static int left_tape, right_tape;
static unsigned int state = -1;
static unsigned int arc_count;

#define LEFT 0 #define RIGHT 1
unsigned int min(unsigned int x, unsigned int y)
{
    if(x < y)
        return x;
    else
        return y;
}

bool goForwardToIntersection(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout)
{
    unsigned int currentTime;
if(*reset == TRUE)
{
    state = -1;
    *reset = FALSE;
}

//handle a stop message
if(message == C_D_STOP)
{
    stopMotors();

    OS_Write(Cout, D_C_STOPPED);
    return TRUE;
}

//handle going forward
else if(message == C_D_NO_MESSAGE || message == C_D_RESUME)
{
    if(message == C_D_RESUME)
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        state--;
    }

    switch(state)
    {
    //start driving forward
    case -1:
        state = 0;
        collision_time[LEFT] = collision_time[RIGHT] =
        getTicks();
        OS_Write(Tout, D_T_REQUEST_READ);
        return FALSE;
        break;

    //drive straight
    case 0:
        if(OS_Read(Tout, &left_tape) == FALSE ||
        OS_Read(Tout, &right_tape) == FALSE)
        {
            OS_Write(Tin, D_T_REQUEST_READ);
            return FALSE;
        }
        OS_Write(Tin, D_T_REQUEST_READ);

        arc_count = 0;
        if(left_tape == BLACK && right_tape == BLACK)
        {
            state = 3;
            forwardMotors(SLOW_SPEED);
        }

        //update motor speed for left collision
        else if(left_tape == BLACK)
        {

        }
arcForwardMotors(MED_SPEED / ARC_RATIO, MED_SPEED);

if the left side always hits
    currentTime = getTicks();
    if(currentTime >= collision_time[LEFT])
        updateLeftForwardMotor(currentTime - collision_time[LEFT], 32);
    else
        updateLeftForwardMotor(currentTime + (0xFFFF - collision_time[LEFT]), 32);
    collision_time[LEFT] = currentTime;
    state = 1;

//update motor speed for right collision
else if(right_tape == BLACK)
    {
        arcForwardMotors(MED_SPEED, MED_SPEED / ARC_RATIO);
        //this will (ONE DAY) try and compensate
        currentTime = getTicks();
        if(currentTime >= collision_time[RIGHT])
            updateRightForwardMotor(currentTime - collision_time[RIGHT], 32);
        else
            updateRightForwardMotor(currentTime + (0xFFFF - collision_time[RIGHT]), 32);
        collision_time[RIGHT] = currentTime;
        state = 2;
    }

else
    forwardMotors(MAX_SPEED);

return FALSE;
break;

//turn left onto the track

case 1:
    {
        if(OS_Read(Tout, &left_tape) == FALSE || OS_Read(Tout, &right_tape) == FALSE)
            {
                OS_Write(Tin, D_T_REQUEST_READ);
                return FALSE;
            }
        OS_Write(Tin, D_T_REQUEST_READ);
        if(left_tape == BLACK && right_tape == BLACK)
            {
                state = 3;
                forwardMotors(SLOW_SPEED);
            }
        else if(left_tape == WHITE)
{ 
    state = 0;
    forwardMotors(MED_SPEED);
}

return FALSE;
break;

// turn right onto the track

case 2:
    if(OS_Read(Tout, &left_tape) == FALSE || OS_Read(Tout, &right_tape) == FALSE)
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        return FALSE;
    }

    OS_Write(Tin, D_T_REQUEST_READ);
    if(left_tape == BLACK && right_tape == BLACK)
    {
        state = 3;
        forwardMotors(SLOW_SPEED);
    }
    else if(right_tape == WHITE)
    {
        state = 0;
        forwardMotors(MED_SPEED);
    }

    return FALSE;
break;

// drive across the intersection

case 3:
    if(OS_Read(Tout, &left_tape) == FALSE || OS_Read(Tout, &right_tape) == FALSE)
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        return FALSE;
    }

    if(left_tape == WHITE || right_tape == WHITE)
    {
        stopMotors();
        state = -1;

        OS_Write(Cout, D_C_AT_INTERSECTION);
        return TRUE;
    }
    else
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        return FALSE;
    }

break;

// if you don't know your own state.... give up
default:
    state = -1;
    return TRUE;
    break;

/******************************
//recover from turning to far left
case 3:
{
    if(OS_Read(Tout, &left_tape) == FALSE ||
    OS_Read(Tout, &right_tape) == FALSE)
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        return FALSE;
    }
    OS_Write(Tin, D_T_REQUEST_READ);
    if(left_tape == BLACK && right_tape == BLACK)
    {
        state = 5;
        forwardMotors(SLOW_SPEED);
        //printf("going to state 5\n");
    }
    else if(right_tape == WHITE)
    {
        state = 0;
        forwardMotors(MED_SPEED);
    }
    else
    {
        arc_count++;
        arcForwardMotors(MED_SPEED,
        min(MED_SPEED, (MED_SPEED / ARC_RATIO) + (arc_count * ARC_INCREASE)));
        return FALSE;
    }
} break;

//recover from turning to far right
case 4:
   if(OS_Read(Tout, &left_tape) == FALSE ||
   OS_Read(Tout, &right_tape) == FALSE)
   {
       OS_Write(Tin, D_T_REQUEST_READ);
       return FALSE;
   }
   OS_Write(Tin, D_T_REQUEST_READ);
   if(left_tape == BLACK && right_tape == BLACK)
   {
       state = 5;
       forwardMotors(SLOW_SPEED);
       //printf("going to state 5\n");
   }
else if(left_tape == WHITE)
{
    state = 0;
    forwardMotors(MED_SPEED);
}
else
{
    arc_count++;
    arcForwardMotors(min(MED_SPEED, (MED_SPEED / ARC_RATIO) + (arc_count * ARC_INCREASE)), MED_SPEED);
}
return FALSE;
break;

}
state--; 
}
switch(state)
{
    case -1:
        state = 0;
pivotLeft(MAX_SPEED);
        OS_Write(Tout, D_T_REQUEST_READ);
        return FALSE;
        break;
//wait to read white on left sensor
    case 0:
        if(OS_Read(Tout, &left_tape) == FALSE ||
        OS_Read(Tout, &right_tape) == FALSE)
        {
            OS_Write(Tin, D_T_REQUEST_READ);
            return FALSE;
        }
    OS_Write(Tin, D_T_REQUEST_READ);
        if(left_tape == WHITE)
        {
            pivotLeft(SLOW_SPEED);
            state = 1;
        }
        return FALSE;
        break;
//wait to read black on left sensor
    case 1:
        if(OS_Read(Tout, &left_tape) == FALSE ||
        OS_Read(Tout, &right_tape) == FALSE)
        {
            OS_Write(Tin, D_T_REQUEST_READ);
            return FALSE;
        }
    OS_Write(Tin, D_T_REQUEST_READ);
        if(left_tape == BLACK)
        {
            state = -1;
            stopMotors();
            OS_Write(Cout, D_C_TURN_COMPLETE);
            return TRUE;
        }
        else
            return FALSE;
        break;
    default:
        state = -1;
        return TRUE;
        break;
}
} else
    //do something
    return FALSE;
// turnright.c
// Contains the code to execute a right turn.
// Called by driving.c when the new goal is a right turn.

#include "os.h"
#include "motors.h"
#include "tapesen.h"
#include "driving.h"
#include <stdio.h>

static int left_tape, right_tape;
static unsigned int state = -1;

bool turnRight(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout)
{
    if(*reset == TRUE)
    {
        state = -1;
        *reset = FALSE;
    }

    //handle a stop message
    if(message == C_D_STOP)
    {
        stopMotors();
        OS_Write(Cout, D_C_STOPPED);
        return TRUE;
    }

    //handle going forward
    else if(message == C_D_NO_MESSAGE || message == C_D_RESUME)
    {
        if(message == C_D_RESUME)
        {
            OS_Write(Tin, D_T_REQUEST_READ);
            state--;
        }

        switch(state)
        {
            case -1:
                state = 0;
                pivotRight(MAX_SPEED);
                OS_Write(Tout, D_T_REQUEST_READ);
                return FALSE;
                break;

            case 0:
                if(OS_Read(Tout, &left_tape) == FALSE ||
                    OS_Read(Tout, &right_tape) == FALSE)
                {
                    OS_Write(Tin, D_T_REQUEST_READ);
                    return FALSE;
                }

                OS_Write(Tin, D_T_REQUEST_READ);
                if(right_tape == WHITE)
                {
                    pivotRight(SLOW_SPEED);
                    state = 1;
                }
                return FALSE;
        }
    }

    return FALSE;
}
break;

//wait to read black on left sensor
case 1:
    if(OS_Read(Tout, &left_tape) == FALSE ||
       OS_Read(Tout, &right_tape) == FALSE)
    {
       OS_Write(Tin, D_T_REQUEST_READ);
       return FALSE;
    }

    OS_Write(Tin, D_T_REQUEST_READ);
    if(right_tape == BLACK)
    {
        state = -1;
        stopMotors();
        OS_Write(Cout, D_C_TURN_COMPLETE);
        return TRUE;
    }
    else
    {
        return FALSE;
    }
break;

default:
    state = -1;
    return TRUE;
break;
} }

///////////////dropleft.c
// Contains the code to execute dropping coins
// in a left turn

#include "os.h"
#include "motors.h"
#include "tapesen.h"
#include "driving.h"
#include "magnet.h"
#include "motors.h"

#include <stdio.h>

#define DELAY_TIME 17

static int left_tape, right_tape;
static unsigned int state = -1;
static unsigned int delayStart = 0;

bool dropCoinLeft(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout)
{
    if(*reset == TRUE)
    {
        

state = -1;
*reset = FALSE;
}

//handle a stop message
if(message == C_D_STOP)
{
    stopMotors();
    OS_Write(Cout, D_C_STOPPED);
    return TRUE;
}

//handle going forward
else if(message == C_D_NO_MESSAGE || message == C_D_RESUME)
{
    if(message == C_D_RESUME)
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        state--;
    }

    switch(state)
    {
    case -1:
        state = 0;
        initServo();
        pivotLeft(MAX_SPEED);
        OS_Write(Tout, D_T_REQUEST_READ);
        return FALSE;
        break;

    case 0:
        if(OS_Read(Tout, &left_tape) == FALSE ||
            OS_Read(Tout, &right_tape) == FALSE)
        {
            OS_Write(Tin, D_T_REQUEST_READ);
            return FALSE;
        }

        OS_Write(Tin, D_T_REQUEST_READ);
        if(left_tape == WHITE)
        {
            pivotLeft(SLOW_SPEED);
            raiseServo();
            state = 1;
        }

        return FALSE;
        break;

    case 1:
        if(OS_Read(Tout, &left_tape) == FALSE ||
            OS_Read(Tout, &right_tape) == FALSE)
        {
            //wait to read black on left sensor
OS_Write(Tin, D_T_REQUEST_READ);
return FALSE;
}

OS_Write(Tin, D_T_REQUEST_READ);

if (left_tape == BLACK)
{
    state = -1;
    stopMotors();
    lowerServo();
    delayStart = getTicks();

    while ((getTicks() - delayStart) < DELAY_TIME);

    OS_Write(Cout, D_C_TURN_COMPLETE);
    killServo();
    return TRUE;
}

else
return FALSE;

break;

default:
    state = -1;
    return TRUE;
break;
}

else
{
    // message was not a recognized command
    return FALSE;
}

}

///////////////////////////////////////////
// dropright.c
// Contains the code to execute the goal of
// turning right and dropping any coins

#include "os.h"
#include "motors.h"
#include "tapesen.h"
#include "driving.h"
#include "magnet.h"
#include "motors.h"

#include <stdio.h>

#define DELAY_TIME 17

static int left_tape, right_tape;
static unsigned int state = -1;
static unsigned int delayStart = 0;

bool dropCoinRight(bool* reset, int message, FIFO Cout, FIFO Tin, FIFO Tout)
{
    if(*reset == TRUE)
    {
        state = -1;
        *reset = FALSE;
    }

    //handle a stop message
    if(message == C_D_STOP)
    {
        stopMotors();
        OS_Write(Cout, D_C_STOPPED);
        return TRUE;
    }

    //handle going forward
    else if(message == C_D_NO_MESSAGE || message == C_D_RESUME)
    {
        if(message == C_D_RESUME)
        {
            OS_Write(Tin, D_T_REQUEST_READ);
            state--;
        }

        switch(state)
        {
        case -1:
            state = 0;
            initServo();
            pivotRight(MAX_SPEED);
            OS_Write(Tout, D_T_REQUEST_READ);
            return FALSE;
            break;

        case 0:
            if(OS_Read(Tout, &left_tape) == FALSE ||
               OS_Read(Tout, &right_tape) == FALSE)
            {
                OS_Write(Tin, D_T_REQUEST_READ);
                return FALSE;
            }

            OS_Write(Tin, D_T_REQUEST_READ);
            if(right_tape == WHITE)
            {
                pivotRight(SLOW_SPEED);
                raiseServo();
                state = 1;
            }

            return FALSE;
        }
    }
break;

//wait to read black on left sensor
case 1:
    if(OS_Read(Tout, &left_tape) == FALSE ||
       OS_Read(Tout, &right_tape) == FALSE)
    {
        OS_Write(Tin, D_T_REQUEST_READ);
        return FALSE;
    }

    OS_Write(Tin, D_T_REQUEST_READ);
    if(right_tape == BLACK)
    {
        state = -1;
        stopMotors();
        lowerServo();
        delayStart = getTicks();

        //give the servo time to lower before disconnecting it
        while((getTicks()-delayStart) < DELAY_TIME);

        OS_Write(Cout, D_C_TURN_COMPLETE);
        killServo();
        return TRUE;
    }
    else
    {
        return FALSE;
        break;
    }
    default:
    {
        state = -1;
        return TRUE;
        break;
    }
}
else  //do something
return FALSE;


/***************************************************************************
 // tapesen.h
 // Declares function used to create tapesensor process
 // and messages sent to the tape sensor.

#ifndef __tapesen_h
#define __tapesen_h

#include "os.h"

#define WHITE 0
#define BLACK 1
#define D_T_REQUEST_READ 1

#endif
bool setupTapeSensors(FIFO in, FIFO out);

#define WINDOW_SIZE 5
static unsigned char LeftWindow[WINDOW_SIZE];
static unsigned char RightWindow[WINDOW_SIZE];
static unsigned int current_index;

/*static*/ FIFO Tin = INVALIDFIFO;
/*static*/ FIFO Tout = INVALIDFIFO;

void TapeSensorProcess(void);

bool setupTapeSensors(FIFO in, FIFO out)
{
    int x;
    for(x = 0; x < WINDOW_SIZE; x++)
    {
        LeftWindow[x] = WHITE;
        RightWindow[x] = WHITE;
    }
    current_index = 0;

    Tin = in;
    Tout = out;

    if(OS_Create(TapeSensorProcess, 0, PERIODIC, TAPE_SEN_NUM) == INVALIDPID)
        return FALSE;
    return TRUE;
}

void TapeSensorProcess(void)
{
    int read, x;
    unsigned char rightsum, leftsum;

    while(1)
    {
        if(OS_Read(Tin, &read) == TRUE && read == D_T_REQUEST_READ)
        {
            flushFiFo(Tout);
        }
    }
}
for(x = 0; x < WINDOW_SIZE; x++)
{
    LeftWindow[current_index] = !(PORTA & 0x01);
    RightWindow[current_index] = !(PORTA & 0x80);
    current_index = (current_index+1)%WINDOW_SIZE;
}

leftsum = 0;
rightsum = 0;
for (x = 0; x < WINDOW_SIZE; x++)
{
    rightsum += RightWindow[x];
    leftsum += LeftWindow[x];
}

OS_Write(Tout, (leftsum >= (WINDOW_SIZE / 2 + 1)));
OS_Write(Tout, (rightsum >= (WINDOW_SIZE / 2 + 1)));
    }
    OS_Yield();
}

///////////////////////////////////////////////////////////////////////////
// coinbump.h
// header file for coin and bumper process

#ifndef _coinbump_h_
#define _coinbump_h_
#include "os.h"
#define COIN 'z'
#define NO_COIN 'y'
#define BUMP 'x'
#define NO_BUMP 'w'

bool setupCoinBump(FIFO CoinOut, FIFO BumpOut);
#endif

///////////////////////////////////////////////////////////////////////////
// coinbump.c
// Cointains the code for setup and execution of the
// processes using the A/D converter.

#include "hc11.h"
#include "os.h"
#include "sched.h"
#include "coinbump.h"

#define COIN_THRESHOLD 50
#define BUMP_THRESHOLD 50

#define COIN_WINDOW 3

void coinBump();
unsigned int readCoin();
unsigned int readBump();

//global FIFOs to communicate sensor status to coordinator
FIFO Cout;
FIFO Bout;

bool setupCoinBump(FIFO coin, FIFO bump)
{
    OPTION |= 0x80;
    if(OS_Create(coinBump, 0, PERIODIC, COIN_BUMPER_NUM) == FALSE)
        return FALSE;

    Cout = coin;
    Bout = bump;
    return TRUE;
}

void killAD()
{
    OPTION &= ~0x80;
    //kill process also?
}

//coinbump process code
void coinBump()
{
    unsigned int result;
    int coinDetect[3];
    int coinIndex = 0;
    int bumpDetect[3];
    int bumpIndex = 0;
    int i, sum;
    bool hasCoin = FALSE;
    bool bumped = FALSE;

    while(1)
    {
        result = readCoin();
        if(result < COIN_THRESHOLD)
            coinDetect[coinIndex] = 1;
        else
            coinDetect[coinIndex] = 0;

        if(coinIndex == 2)
        {
            sum = 0;
            for(i = 0; i < COIN_WINDOW; i++)
                sum += coinDetect[i];
            if(sum >= (COIN_WINDOW/2 + 1))
                { if(!hasCoin)

        }
hasCoin = TRUE;
OS_Write(Cout, COIN);
}
else
{
    if(hasCoin)
    {
        hasCoin = FALSE;
        OS_Write(Cout, NO_COIN);
    }
}
coinIndex = (++coinIndex) % COIN_WINDOW;

if(readBump() > BUMP_THRESHOLD)
{
    if(!bumped)
    {
        bumped = TRUE;
        OS_Write(Bout, BUMP);
    }
} else
{
    if(bumped)
    {
        bumped = FALSE;
        OS_Write(Bout, NO_BUMP);
    }
}
OS_Yield();

unsigned int readCoin()
{
    //initialize A/D approximation sequence
    ADCTL = 0x06;
    //check CCF flag
    while((ADCTL & 0x80) == 0);
    return ADR3;

    /*printf("[%d %d %d %d]", ADR1, ADR2, ADR3, ADR4);
    //average the four approximations
    result = ADR1 + ADR2 + ADR3 + ADR4;
    return (result >> 2); //divide by four */
}

unsigned int readBump()
{
    ADCTL = 0x03;
    while((ADCTL & 0x80) == 0);
    return ADR4;
}
// magnet.h
// Header file containing prototypes of
// methods used to manipulate servo magnet

#ifndef _MAGNET_H_
#define _MAGNET_H_

#define PERIOD 40000
#define MIN_PW 1350
#define MAX_PW 4300

// may need tweaking for each robot
#define UP 2250
#define DOWN 4000

void initServo();
void killServo();
void raiseServo();
void lowerServo();

#endif

#include "hc11.h"
#include "os.h"
#include "magnet.h"

static unsigned int servo_position;

#pragma interrupt_handler Servo_ISR()

void Servo_ISR()
{
    if (PORTA & 0x10) /* SIGNAL is currently HIGH */
    {
        // pull it LOW on next compare for "cur_position" count
        TOC4 += servo_position;
    }
    else /* SIGNAL is currently LOW */
    {
        // pull it HIGH on next compare for the remaining period
        TOC4 += (PERIOD - servo_position);
    }
    TFLG1 = 0x10;
}

void initServo()
{
    *(void (**))0xFFE2 = Servo_ISR;

    // toggle PA4 on successful OC
    TCTL1 |= 0x04;
    // start PA4 high
    PORTA |= 0x10;
    // set time of next OC
}
servo_position = DOWN;
TOC4 = TCNT + servo_position;
//clear flag
TFLG1 = 0x10;
//enable OC4 interrupt
TMSK1 |= 0x10;
}

//disable servo and free OC4
void killServo()
{
    //take PA4 low
    PORTA &= ~0x10;
    TCTL1 &= ~0x0C;
    TMSK1 &= ~0x10;
}

void raiseServo()
{
    servo_position = UP;
}

void lowerServo()
{
    servo_position = DOWN;
}

/***********************************************
// motors.h
// Header file to declare all wheel control functions
// and speed constants
#ifndef __motors_h
#define __motors_h
#define MAX_SPEED 90
#define MED_SPEED 55
#define SLOW_SPEED 40

void initMotors(void);
void killMotors(void);

void stopMotors(void);
void forwardMotors(unsigned int speed);
void updateRightForwardMotor(unsigned int period, unsigned int rate);
void updateLeftForwardMotor(unsigned int period, unsigned int rate);

void pivotForwardRightMotor(unsigned int speed);
void pivotForwardLeftMotor(unsigned int speed);

void arcForwardMotors(unsigned int leftspeed, unsigned int rightspeed);
void pivotLeft(unsigned int speed);
void pivotRight(unsigned int speed);

unsigned int getTicks();
#endif
 motors.c
Contains the drivers for the motors.
Controls the PWM signals which determine the power
transferred to the motor

#include "motors.h"
#include "hc11.h"
#include <stdio.h>

static unsigned int ticks;
#define OC1F 0x80

/* Constants */
#define SPEED 650  // 1/100th of maximum speed, must <= 650
#define L_DIR 0x10  // PD4
#define R_DIR 0x20  // PD5
#define L_FORWARD 0x10  // PD4 = 1
#define R_FORWARD 0x20  // PD5 = 1
#define L_BACKWARD ~0x10  // PD4 = 0
#define R_BACKWARD ~0x20  // PD5 = 0
#define FORWARD_MAX 100
#define BACKWARD_MAX 100
#define fwdScale 10

static unsigned int left_forward_max = FORWARD_MAX;
static unsigned int right_forward_max = FORWARD_MAX;
static unsigned int left_backward_max = BACKWARD_MAX;
static unsigned int right_backward_max = BACKWARD_MAX;

#pragma interrupt_handler timerISR()
void timerISR(void)
{
    ticks++;  // update global 'tick' count
    TFLG1 = OC1F;
}

void initMotors(void)
{
    DDRD |= 0x30;  // Set PD4 and PD5 to be output.
    PORTD |= (L_FORWARD | R_FORWARD);  // Set the direction bits to forward.
    OC1M = 0x60;  // OC1 affects OC2 and OC3, i.e., PA6 and PA5.
    OC1D = 0x60;  // Successful OC1 turns on OC2 and OC3
    TCTL1 = (TCTL1 & ~0xF0) | 0xA0;  // OC2 turn off PA6, OC3 turn off PA5
    TOC1 = 0;
    TOC2 = 1;  // control PA6, which is the left motor
    TOC3 = 1;  // control PA5, which is the right motor

    // setup OC1 ISR to timerISR
    *(void (**)) (0xFFE8) = timerISR;  // address of interrupt handler

    ticks = 0;
    TFLG1 = OC1F;
void killMotors(void)
{
    DDRD &= ~0x30;
    OC1M &= ~0x60;
    OC1D &= ~0x60;
    TCTL1 = (TCTL1 & ~0xF0) | 0x00;
}

void stopMotors(void)
{
    OC1M = 0x00; //disconnect OC1 from PA6, PA5
}

void forwardMotors(unsigned int speed)
{
    if(speed > FORWARD_MAX)
        speed = FORWARD_MAX;

    PORTD |= (L_FORWARD | R_FORWARD);
    TOC2 = SPEED * speed;
    TOC3 = SPEED * speed;

    OC1M = 0x60; //successful OC1 takes PA6, PA5 high
}

void updateRightForwardMotor(unsigned int period, unsigned int rate)
{
    unsigned int scaleAmount = fwdScale / period * rate;

    if((left_forward_max - FORWARD_MAX) >= scaleAmount)
    {
        left_forward_max += fwdScale/period*rate;
        TOC2 = SPEED * left_forward_max;
    }
    else
    {
        right_forward_max -= fwdScale/period*rate;
        TOC3 = SPEED * right_forward_max;
    }
}

void updateLeftForwardMotor(unsigned int period, unsigned int rate)
{
    unsigned int scaleAmount = fwdScale/period*rate;

    if((right_forward_max - FORWARD_MAX) >= scaleAmount)
    {
        right_forward_max += fwdScale/period*rate;
        TOC3 = SPEED * right_forward_max;
    }
    else
    {
        left_forward_max -= fwdScale/period*rate;
        TOC2 = SPEED * left_forward_max;
    }
void pivotForwardRightMotor(unsigned int speed)
{
    if(speed > FORWARD_MAX)
        speed = FORWARD_MAX;

    PORTD |= (L_FORWARD | R_FORWARD);
    TOC2 = 1;
    TOC3 = SPEED * speed;
    OC1M = 0x60;  // successful OC1 takes PA6, PA5 high
}

void pivotForwardLeftMotor(unsigned int speed)
{
    if(speed > FORWARD_MAX)
        speed = FORWARD_MAX;

    PORTD |= (L_FORWARD | R_FORWARD);
    TOC2 = SPEED * speed;
    TOC3 = 1;
    OC1M = 0x60;  // successful OC1 takes PA6, PA5 high
}

void arcForwardMotors(unsigned int leftspeed, unsigned int rightspeed)
{
    if(leftspeed > FORWARD_MAX)
        leftspeed = FORWARD_MAX;
    if(rightspeed > FORWARD_MAX)
        rightspeed = FORWARD_MAX;

    PORTD |= (L_FORWARD | R_FORWARD);
    TOC2 = SPEED * leftspeed;
    TOC3 = SPEED * rightspeed;
    OC1M = 0x60;  // successful OC1 takes PA6, PA5 high
}

void pivotLeft(unsigned int speed)
{
    if(speed > FORWARD_MAX)
        speed = FORWARD_MAX;

    PORTD |= R_FORWARD;
    PORTD &= L_BACKWARD;
    TOC2 = SPEED * speed;
    TOC3 = SPEED * speed;
    OC1M = 0x60;  // successful OC1 takes PA6, PA5 high
}

void pivotRight(unsigned int speed)
{
    if(speed > FORWARD_MAX)
        speed = FORWARD_MAX;

    PORTD |= L_FORWARD;
    PORTD &= R_BACKWARD;
    TOC2 = SPEED * speed;
    TOC3 = SPEED * speed;
    OC1M = 0x60;  // successful OC1 takes PA6, PA5 high
}
//public method to access current 'tick' count updated everytime an OCl
//interrupt occurs
unsigned int getTicks()
{
    return ticks;
}

#endif

// Support code to flush any data from a FIFO

int main(void)
{
    DI();
    OS_Init();
    OS_Create(Coordinator, 0, SPORADIC, 0);
    OS_Start();
    return 1;
}
### pcomm.h

```c
#ifndef __pcomm_h
#define __pcomm_h

#include "os.h"

bool setupComm(FIFO in, FIFO out);

#endif
```

### pcomm.c

```c
#include "uart.h"
#include "pcomm.h"
#include "os.h"
#include "sched.h"
#include <stdio.h>

void ReadSerial(void);
void WriteSerial(void);

static FIFO pRxFIFO;
static FIFO pTxFIFO;

bool setupComm(FIFO in, FIFO out)
{
    uart_init(BAUD_9600);
    pRxFIFO = in;
    pTxFIFO = out;
    if(OS_Create(ReadSerial, 0, PERIODIC, SERIAL_READ_NUM) == INVALIDPID)
        return FALSE;
    if(OS_Create(WriteSerial, 0, PERIODIC, SERIAL_WRITE_NUM) == INVALIDPID)
        return FALSE;
    return TRUE;
}

void ReadSerial(void)
{
    char received;

    while(1)
    {
        if(uart_char_pending())
        {
            received = uart_getchar();
            OS_Write(pRxFIFO, (int) received);
        }
        OS_Yield();
    }
}

void WriteSerial(void)
{
{ unsigned char LowByte;
  int write;
  while(1)
  {
    if(OS_Read(pTxFIFO, &write) == TRUE)
    {
      LowByte = (unsigned char) write;
      uart_putchar(LowByte)
    }
    OS_Yield();
  }
}

////////////////////////////////////////////////////////////////////////////////////////
//  pcoord.h
//
#define __pcoord_h
#define __pcoord_h
#include "os.h"
void Coordinator(void);
#endif

////////////////////////////////////////////////////////////////////////////////////////
//  pcoord.c
//
#include "pcomm.h"
#include "lcd.h"
#include "keys.h"
#include ".../messages.h"
#include "nav.h"
#include "system.h"
#define WAIT_STATE    -1
#define START_STATE    0
#define GOING_FORWARD_STATE  1
#define TURNING_LEFT_STATE   2
#define TURNING_RIGHT_STATE  3
#define DROPPING_COIN_LEFT_STATE  4
#define DROPPING_COIN_RIGHT_STATE  5

FIFO CommIn = INVALIDFIFO;
FIFO CommOut = INVALIDFIFO;

void Coordinator(void)
{
  int commread;
  int state;
  bool have_coin, done;
  unsigned int x_pos, y_pos, orientation, dest_x, dest_y, move;

  CommIn = OS_InitFiFo();
  CommOut = OS_InitFiFo();

  if(CommIn == INVALIDFIFO || CommOut == INVALIDFIFO)
    return;
if(setupComm(CommIn, CommOut) == FALSE)
    return;

//if(setupKeys() == FALSE)
//    return;

x_pos = 0;
y_pos = 0;
orientation = ORIENT_UP;
have_coin = FALSE;
state = WAIT_STATE;
resetIntersectionMarks();

while(1)
{
    if(OS_Read(CommIn, &commread) == FALSE)
        commread = R_P_NO_MESSAGE;

    if(commread == R_P_RESET)
    {
        state = WAIT_STATE;
        commread = R_P_NO_MESSAGE;
    }

    switch(state)
    {
    case WAIT_STATE:
        if(commread == R_P_BUMPER_OUT)
            state = START_STATE;
        break;

    case START_STATE:
        x_pos = 0;
y_pos = 0;
        orientation = ORIENT_UP;

        markIntersection(0, 0);
        done = FALSE;

        if(have_coin == TRUE)
        {
            switch(orientation)
            {
            case ORIENT_UP:
                dest_x = 0;
dest_y = 1;
            break;

            case ORIENT_DOWN:
                state = DROPPING_COIN_LEFT_STATE;
                OS_Write(CommOut, P_R_DROP_COIN_LEFT);

                done = TRUE;
            break;

            case ORIENT_LEFT:
                state = DROPPING_COIN_RIGHT_STATE;
                state =

                OS_Write(CommOut, P_R_DROP_COIN_RIGHT);

                done = TRUE;
            break;

            case ORIENT_RIGHT:
                state =

                OS_Write(CommOut, P_R_DROP_COIN_RIGHT);

                done = TRUE;
            break;

            default:
                break;
            }
        }
    break;
    }
}
case ORIENT_RIGHT:
    dest_x = 1;
    dest_y = 0;
    break;
}
}

else if(getNextIntersection(x_pos, y_pos, orientation, &dest_x, &dest_y) == FALSE)
{
    resetIntersectionMarks();
    markIntersection(x_pos, y_pos);
    if(getNextIntersection(x_pos, y_pos, orientation, &dest_x, &dest_y) == FALSE)
        EXIT();
}

if(done != TRUE)
{
    move = getMoveToDestination(x_pos, y_pos, orientation, dest_x, dest_y);
    switch(move)
    {
    case MOVE_NONE:
        //this should not happen
        //EXIT();
        break;
    case MOVE_FORWARD:
        state = GOING_FORWARD_STATE;
        OS_Write(CommOut, P_R_GO_FORWARD_TO_INT);
        break;
    case MOVE_LEFT:
        state = TURNING_LEFT_STATE;
        OS_Write(CommOut, P_R_TURN_LEFT);
        break;
    case MOVE_RIGHT:
        state = TURNING_RIGHT_STATE;
        OS_Write(CommOut, P_R_TURN_RIGHT);
        break;
    }
    break;
}

case GOING_FORWARD_STATE:
    switch(commread)
    {
    case R_P_AT_INT_WITH_COIN:
        updatePositionWithMove(orientation, &x_pos, &y_pos);
        markIntersection(x_pos, y_pos);
        dest_x = 0;
        dest_y = 0;
have_coin = TRUE;
move = getMoveToDestination(x_pos,
switch(move)
{
    case MOVE_NONE:
        if(orientation ==
        {
            state =
            OS_Write(CommOut,
        }
    else //orientation ==
        {
            state =
            OS_Write(CommOut,
        }
    break;
    case MOVE_FORWARD:
        state =
        OS_Write(CommOut,
    break;
    case MOVE_LEFT:
        state =
        OS_Write(CommOut,
    break;
    case MOVE_RIGHT:
        state =
        OS_Write(CommOut,
    break;
    case R_P_AT_INT_WITHOUT_COIN:
        updatePositionWithMove(orientation,
        markIntersection(x_pos, y_pos);
        have_coin = FALSE;
        if(getNextIntersection(x_pos, y_pos,
        orientation, &dest_x, &dest_y) == FALSE)
        {
            resetIntersectionMarks();
            markIntersection(x_pos, y_pos);
if(getNextIntersection(x_pos, y_pos, orientation, &dest_x, &dest_y) == FALSE)
    EXIT();

move = getMoveToDestination(x_pos, y_pos, orientation, dest_x, dest_y);
switch(move)
{
    case MOVE_NONE:
        //this should not happen
        //EXIT();
        break;
    case MOVE_FORWARD:
        state = GOING_FORWARD_STATE;
        OS_Write(CommOut, P_R_GO_FORWARD_TO_INT);
        break;
    case MOVE_LEFT:
        state = TURNING_LEFT_STATE;
        OS_Write(CommOut, P_R_TURN_LEFT);
        break;
    case MOVE_RIGHT:
        state = TURNING_RIGHT_STATE;
        OS_Write(CommOut, P_R_TURN_RIGHT);
        break;
    case R_P_LOST:
        break;
    case R_P_BUMPER_IN:
        break;
}
break;

case TURNING_LEFT_STATE:
    switch(commread)
    {
    case R_P_TURN_COMPLETE:
        updateOrientationWithMove(TURN_LEFT, &orientation);
        move = MOVE_FORWARD;
        switch(move)
        {
            case MOVE_NONE:
                //this should not happen
                //EXIT();
                break;
            case MOVE_FORWARD:
                break;
        }
    }
break;
state =

GOING_FORWARD_STATE;

P_R_GO_FORWARD_TO_INT);

OS_Write(CommOut,

break;

case MOVE_LEFT:
    //this should not happen
    //EXIT();
break;

case MOVE_RIGHT:
    //this should not happen
    //EXIT();
break;
}
break;

case R_P_LOST:
    break;

case R_P_BUMPER_IN:
    break;
}
break;

case TURNING_RIGHT_STATE:
    switch(commread)
    {
        case R_P_TURN_COMPLETE:
            updateOrientationWithMove(TURN_RIGHT,
                &orientation);
            move = MOVE_FORWARD;
            switch(move)
            {
                case MOVE_NONE:
                    //this should not happen
                    //EXIT();
break;

                case MOVE_FORWARD:
                    state =

GOING_FORWARD_STATE;

P_R_GO_FORWARD_TO_INT);

break;

case MOVE_LEFT:
    //this should not happen
    //EXIT();
break;

case MOVE_RIGHT:
    //this should not happen
    //EXIT();
break;
}
break;

case R_P_LOST:
    break;

case R_P_BUMPER_IN:


break;
break;

case DROPPING_COIN_LEFT_STATE:
    switch(commread)
    {
        case R_P_TURN_COMPLETE:
            updateOrientationWithMove(TURN_LEFT, &orientation);
            have_coin = FALSE;
            if(getNextIntersection(x_pos, y_pos, orientation, &dest_x, &dest_y) == FALSE)
            {
                resetIntersectionMarks();
                markIntersection(x_pos, y_pos);
                if(getNextIntersection(x_pos, y_pos, orientation, &dest_x, &dest_y) == FALSE)
                    EXIT();
            }
            move = getMoveToDestination(x_pos, y_pos, orientation, dest_x, dest_y);
            switch(move)
            {
                case MOVE_NONE:
                    //this should not happen
                    //EXIT();
                    break;
                case MOVE_FORWARD:
                    state = GOING_FORWARD_STATE;
                    OS_Write(CommOut, P_R_GO_FORWARD_TO_INT);
                    break;
                case MOVE_LEFT:
                    //this should not happen
                    //EXIT();
                    break;
                case MOVE_RIGHT:
                    //this should not happen
                    //EXIT();
                    break;
            }
            break;
        case R_P_LOST:
            break;
        case R_P_BUMPER_IN:
            break;
    }
    break;

case DROPPING_COIN_RIGHT_STATE:
    switch(commread)
    {
        case R_P_TURN_COMPLETE:
            break;
    }
&orientation);

  updateOrientationWithMove(TURN_RIGHT, have_coin = FALSE;

  orientation, &dest_x, &dest_y) == FALSE)
  {
    resetIntersectionMarks();
    markIntersection(x_pos, y_pos);

    if(getNextIntersection(x_pos, y_pos, orientation, &dest_x, &dest_y) == FALSE)
      EXIT();
  }

  move = getMoveToDestination(x_pos, y_pos, orientation, dest_x, dest_y);

  switch(move)
  {
    case MOVE_NONE:
      // this should not happen
      // EXIT();
      break;

    case MOVE_FORWARD:
      state = GOING_FORWARD_STATE;
      OS_Write(CommOut, P_R_GO_FORWARD_TO_INT);
      break;

    case MOVE_LEFT:
      // this should not happen
      // EXIT();
      break;

    case MOVE_RIGHT:
      // this should not happen
      // EXIT();
      break;
  }

  break;

  case R_P_LOST:
  break;

  case R_P_BUMPER_IN:
    break;
  }

  break;

  OS_Yield();
}


// nav.h
//

#ifndef __nav_h
#define __nav_h

Autonomous Mobile Robot Design and Implementation A-67 CENG 499
```c
#include "os.h"
#define MOVE_NONE 0
#define MOVE_FORWARD 1
#define MOVE_LEFT 2
#define MOVE_RIGHT 3
#define ORIENT_UP 0
#define ORIENT_DOWN 1
#define ORIENT_LEFT 2
#define ORIENT_RIGHT 3
#define TURN_LEFT 0
#define TURN_RIGHT 1
#define BOARD_X 4
#define BOARD_Y 4

void updatePositionWithMove(unsigned int orientation, unsigned int *x, unsigned int *y);
void updateOrientationWithMove(unsigned int turn, unsigned int *orientation);

void resetIntersectionMarks(void);
void markIntersection(unsigned int x, unsigned int y);

bool getNextIntersection(unsigned int cx, unsigned int cy, unsigned int orient, unsigned int *dx, unsigned int *dy);
unsigned int getMoveToDestination(unsigned int cx, unsigned int cy, unsigned int orient, unsigned int dx, unsigned int dy);

#endif
```

```c
#include "os.h"
#define MOVE_NONE 0
#define MOVE_FORWARD 1
#define MOVE_LEFT 2
#define MOVE_RIGHT 3
#define ORIENT_UP 0
#define ORIENT_DOWN 1
#define ORIENT_LEFT 2
#define ORIENT_RIGHT 3
#define TURN_LEFT 0
#define TURN_RIGHT 1
#define BOARD_X 4
#define BOARD_Y 4

#include "nav.h"
#include "lcd.h"
#include <stdio.h>
#define MARKED 1
#define UNMARKED 0

unsigned char board[BOARD_X][BOARD_Y];

void updatePositionWithMove(unsigned int orientation, unsigned int *x, unsigned int *y)
{
    //char string[20];
    switch(orientation)
    {
    case ORIENT_UP:
        (*y)++;
        //sprintf(string, "p x:%d y:%d up\n", *x, *y);
        break;

    case ORIENT_DOWN:
        (*y) --;
        //sprintf(string, "p x:%d y:%d down\n", *x, *y);
        break;

```

---

Autonomous Mobile Robot Design and Implementation  A-68  
CENG 499
case ORIENT_LEFT:
    (*x)--;
    //sprintf(string, "p x:%d y:%d left\n", *x, *y);
    break;

case ORIENT_RIGHT:
    (*x)++;
    //sprintf(string, "p x:%d y:%d right\n", *x, *y);
    break;
} //lcd_puts(string);
}

void updateOrientationWithMove(unsigned int turn, unsigned int *orientation)
{
    if(turn == TURN_LEFT)
    {
        switch(*orientation)
        {
            case ORIENT_UP:
                *orientation = ORIENT_LEFT;
                break;
            case ORIENT_DOWN:
                *orientation = ORIENT_RIGHT;
                break;
            case ORIENT_LEFT:
                *orientation = ORIENT_DOWN;
                break;
            case ORIENT_RIGHT:
                *orientation = ORIENT_UP;
                break;
        }
    }
    else
    {
        switch(*orientation)
        {
            case ORIENT_UP:
                *orientation = ORIENT_RIGHT;
                break;
            case ORIENT_DOWN:
                *orientation = ORIENT_LEFT;
                break;
            case ORIENT_LEFT:
                *orientation = ORIENT_UP;
                break;
            case ORIENT_RIGHT:
                *orientation = ORIENT_DOWN;
                break;
        }
    }

void resetIntersectionMarks(void)
{
int x, y;
for (x = 0; x < BOARD_X; x++)
{
    for (y = 0; y < BOARD_Y; y++)
        board[x][y] = UNMARKED;
}

void markIntersection(unsigned int x, unsigned int y)
{
    if (x < BOARD_X && y < BOARD_Y)
        board[x][y] = MARKED;
}

bool getNextIntersection(unsigned int cx, unsigned int cy, unsigned int orient,
unsigned int *dx, unsigned int *dy)
{
    //char string[20];
    int num_turns = 5;
    int distance = BOARD_X + BOARD_Y;
    int temp_distance, temp_num_turns;
    bool found = FALSE;

    int x, y;
    //if we can, move up
    if (cy + 1 < BOARD_Y && board[cx][cy + 1] == UNMARKED)
    {
        temp_distance = cx + cy + 1;
        if (orient == ORIENT_UP)
            temp_num_turns = 0;
        else if (orient == ORIENT_DOWN)
            temp_num_turns = 3;
        else
            temp_num_turns = 1;

        if (temp_num_turns == num_turns)
            if (temp_distance < distance)
                { found = TRUE;
                    distance = temp_distance;
                    num_turns = temp_num_turns;
                    *dx = cx;
                    *dy = cy + 1;
                }
        else if (temp_num_turns < num_turns)
            { found = TRUE;
                distance = temp_distance;
                num_turns = temp_num_turns;
                *dx = cx;
                *dy = cy + 1;
            }
    }

    if (cx + 1 < BOARD_X && board[cx + 1][cy] == UNMARKED)
    {
        temp_distance = cx + cy + 1;
        if (orient == ORIENT_RIGHT)
temp_num_turns = 0;
else if(orient == ORIENT_LEFT)
    temp_num_turns = 3;
else
    temp_num_turns = 1;

if(temp_num_turns == num_turns)
{
    if(temp_distance < distance)
    {
        found = TRUE;
        distance = temp_distance;
        num_turns = temp_num_turns;
        *dx = cx + 1;
        *dy = cy;
    }
}
else if(temp_num_turns < num_turns)
{
    found = TRUE;
    distance = temp_distance;
    num_turns = temp_num_turns;
    *dx = cx + 1;
    *dy = cy;
}

if(((int)cy) - 1 >= 0 && board[cx][cy - 1] == UNMARKED)
{
    temp_distance = cx + cy - 1;
    if(orient == ORIENT_UP)
        temp_num_turns = 3;
    else if(orient == ORIENT_DOWN)
        temp_num_turns = 0;
    else
        temp_num_turns = 1;

    if(temp_num_turns == num_turns)
    {
        if(temp_distance < distance)
        {
            found = TRUE;
            distance = temp_distance;
            num_turns = temp_num_turns;
            *dx = cx;
            *dy = cy - 1;
        }
    }
    else if(temp_num_turns < num_turns)
    {
        found = TRUE;
        distance = temp_distance;
        num_turns = temp_num_turns;
        *dx = cx;
        *dy = cy - 1;
    }
}

if(((int)cx) - 1 >= 0 && board[cx - 1][cy] == UNMARKED)
{
    temp_distance = cx + cy - 1;
    if(orient == ORIENT_RIGHT)
        temp_num_turns = 3;
else if(orient == ORIENT_LEFT)
    temp_num_turns = 0;
else
    temp_num_turns = 1;

if(temp_num_turns == num_turns)
{
    if(temp_distance < distance)
    {
        found = TRUE;
        distance = temp_distance;
        num_turns = temp_num_turns;
        *dx = cx - 1;
        *dy = cy;
    }
}
else if(temp_num_turns < num_turns)
{
    found = TRUE;
    distance = temp_distance;
    num_turns = temp_num_turns;
    *dx = cx - 1;
    *dy = cy;
}

if(found == FALSE)
{
    for(x = 0; x < BOARD_X; x++)
    {
        for(y = 0; y < BOARD_Y; y++)
        {
            if(board[x][y] == UNMARKED)
            {
                found = TRUE;
                *dx = x;
                *dy = y;
            }
        }
    }
}

//sprintf(string, "d x:%d y:%d\n", *dx, *dy);
//lcd_puts(string);
return found;

unsigned int getMoveToDestination(unsigned int cx, unsigned int cy, unsigned int orient, unsigned int dx, unsigned int dy)
{
    int move_x = ((int)dx) - ((int)cx);
    int move_y = ((int)dy) - ((int)cy);
    unsigned int move;

    switch(orient)
    {
        case ORIENT_UP:
            if(move_y > 0)
                move = MOVE_FORWARD;
            else if(move_x > 0)
                move = MOVE_RIGHT;
            else if(move_x < 0)
                move = MOVE_LEFT;
            else if(move_x == 0)
                move = MOVE_FORWARD;
            else if(move_y == 0)
                move = MOVE_UP;
            else
                move = MOVE_UP;
            break;
        case ORIENT_RIGHT:
            if(move_x > 0)
                move = MOVE_FORWARD;
            else if(move_x < 0)
                move = MOVE_LEFT;
            else if(move_y == 0)
                move = MOVE_RIGHT;
            else
                move = MOVE_RIGHT;
            break;
        case ORIENT_DOWN:
            if(move_y < 0)
                move = MOVE_FORWARD;
            else if(move_x > 0)
                move = MOVE_RIGHT;
            else if(move_x < 0)
                move = MOVE_LEFT;
            else if(move_x == 0)
                move = MOVE_DOWN;
            else
                move = MOVE_DOWN;
            break;
        case ORIENT_LEFT:
            if(move_x < 0)
                move = MOVE_FORWARD;
            else if(move_x > 0)
                move = MOVE_LEFT;
            else if(move_y == 0)
                move = MOVE_DOWN;
            else
                move = MOVE_DOWN;
            break;
    }
    return move;
}
move = MOVE_LEFT;
else if (move_x == 0 && move_y == 0)
    move = MOVE_NONE;
else //move_y < 0 && move_x == 0
{
    if (cx == 0)
        move = MOVE_RIGHT;
    else
        move = MOVE_LEFT;
}
break;

case ORIENT_DOWN:
    if (move_y < 0)
        move = MOVE_FORWARD;
    else if (move_x > 0)
        move = MOVE_LEFT;
    else if (move_x < 0)
        move = MOVE_RIGHT;
    else if (move_x == 0 && move_y == 0)
        move = MOVE_NONE;
    else //move_y > 0 && move_x == 0
    {
        if (cx == 0)
            move = MOVE_LEFT;
        else
            move = MOVE_RIGHT;
    }
break;

case ORIENT_LEFT:
    if (move_x < 0)
        move = MOVE_FORWARD;
    else if (move_y > 0)
        move = MOVE_RIGHT;
    else if (move_y < 0)
        move = MOVE_LEFT;
    else if (move_y == 0 && move_x == 0)
        move = MOVE_NONE;
    else //move_x > 0 && move_y == 0
    {
        if (cy == 0)
            move = MOVE_RIGHT;
        else
            move = MOVE_LEFT;
    }
break;

case ORIENT_RIGHT:
    if (move_x > 0)
        move = MOVE_FORWARD;
    else if (move_y > 0)
        move = MOVE_LEFT;
    else if (move_y < 0)
        move = MOVE_RIGHT;
    else if (move_y == 0 && move_x == 0)
        move = MOVE_NONE;
    else //move_x < 0 && move_y == 0
    {
        if (cy == 0)
            move = MOVE_LEFT;
        else
            move = MOVE_RIGHT;
break;

return move;
}

// sched.h
#endif
}

// keys.h
// [In the future will] enable keypad input from the Palm

bool setupKeys(void);
#endif

// keys.c
// Allows input from the Palm keys

static void **ramvec;

struct PortD
{
    unsigned char dir;
    unsigned char irqen;
    unsigned char iqeg;
    unsigned char kben;
    unsigned char sel;
    unsigned char data;
};

//port d backup
struct PortD portd;

void savePortD(void){
    //save port d registers
    portd.dir = PDDIR;
    portd.irqen = PDIRQEN;
    portd.iqeg = PDIQEG;
    portd.kben = PDKBEN;
void restorePortD(void){
    //save port d registers
    PDDIR = portd.dir;
    PDIRQEN = portd.irqen;
    PDIQEG = portd.iqeg;
    PDKBEN = portd.kben;
    PDSEL = portd.sel;
    PDDATA = portd.data;
}

INTERRUPT_HANDLER(keyboard_isr)
{
    IMR |= IMR_MKB;
    //restore port d registers
    restorePortD();
    //OS_Abort();
    EXIT();
}

bool setupKeys(void)
{
    ramvec[28] = keyboard_isr;
    //save the values currently in port d registers
    savePortD();
    PDDIR &= ~0x0F; //set port d as input
    PDIRQEN &= 0x00; //enable interrupts
    PDIQEG |= 0x0F; //set edge sensitive
    PDKBEN = 0xFF; //keyboard enable ints
    PDSEL |= 0xF0; //connect interrupt (not i/o sigs,0x00)
    PDDATA = 0x00; //reset data reg
    IMR &= ~IMR_MKB;

    return TRUE;
}
A.4 Shared Files

```c
#ifndef __messages_h
#define __messages_h

//messages
#define P_R_NO_MESSAGE   0
#define P_R_GO_FORWARD_TO_INT  1
#define P_R_BACKUP_FROM_INT  2
#define P_R_BACKUP_FROM_MID  3
#define P_R_TURN_LEFT   4
#define P_R_TURN_RIGHT  5
#define P_R_RESUME   6
#define P_R_STOP_DRIVE  7
#define P_R_CRY   8
#define P_R_DROP_COIN_LEFT  9
#define P_R_DROP_COIN_RIGHT 10

//replies
#define R_P_NO_MESSAGE  0
#define R_P_AT_INT_WITH_COIN 1
#define R_P_AT_INT_WITHOUT_COIN 2
#define R_P_TURN_COMPLETE  3
#define R_P_STOPPED   4
#define R_P_LOST   5
#define R_P_BUMPER_IN  6
#define R_P_BUMPER_OUT  7
#define R_P_CRY_DONE   8
#define R_P_RESET   9

#endif
```