Avtar Khalsa
Cornell University ECE 2008
166 Village Street
Millis, MA, 02054
774-270-1307
ask43@cornell.edu

T&AM 492
3 credits
Restructuring of Cornell Ranger Control Code
Cornell Biorobotics and Locomotion Lab
For Professor Andy Ruina
Spring 2008

Abstract:

During the course of the semester, my goal was to come up with new control system for operating the Cornell Ranger. It was to use modular coding concepts in order to achieve maximum possible scalability. While I was restricted to using C, I was also to take into consideration that we would be switching at some point soon to C++, and consequently would want to incorporate object

oriented programming concepts. Time was also a factor as our goal was the make the robot walk robustly within a few months. In the end we were successful in using the code to create a fairly robust walking algorithm, although we are now designing a larger scale more flexible version which will port well to C++ and be able to handle all the foreseeable future operations we are planning.

Introduction:

At the beginning of the semester, we started the very ambitious project of completely rewriting the code that effectively made the Cornell Ranger walk. In order to understand the initial state of the code, it is important to understand exactly how it came about. The code for the Ranger was not carefully planned from the beginning where everything was placed where it should be according to a master plan. It more came about through a kind of natural evolution. It started off as a very simple short block of code, and then increased in length as necessary to accommodate new controllers and new actions as necessary. This is very natural when code is not properly planned out from the beginning.

To illustrate a simple way that this can happen, consider the following scenario. Imagine a project where the goal is to model a ball bouncing off a floor. If this were all one were concerned about, it would be very simple to write one short block of code which handles all the decisions. It would compute where a ball should be at any given time and in the case of impact with a floor, remove some energy and just reverse the direction the ball was travelling in. However, if the project then evolved to take into account an impact with any surface angled

in any direction relative to the ball, it would be difficult to use the old code. The new code would need to be structured very differently to take all these new parameters into account.

For instance, it might be useful to write a function which handled impacts and took into account the direction, and speed of the ball, along with the angle of the wall, and returned the new direction and velocity of the ball. The wrapper code would just use the state information about the ball to update its position. While this restructuring sounds very simple, in the case of longer code it can be much more challenging. The result tends to be leaving the code structure intact and trying to make it fit the new goal. This is a lot like the analogy of trying to fit a square peg in a round hole. This leads to nested if statements throughout the code to try to handle all possible scenarios, which in turn makes the code very difficult to read.

The obvious solution to this is of course to plan the code extensively before anything is written. There are several components to this planning process. To begin with, it is important to take an extensive look at exactly what the code is expected to do. For instance, if the programmer in the above example had known from the start that the code was eventually going to need to handle ball impacts against walls in different directions, it would have made the initial solution of just using one block of code seem extremely impractical and he most likely would have known to use try the second solution from the start. However,

all the planning cannot take into account every possible use for the code. It's very likely that somewhere along the line, it will be important for the code to do something it wasn't initially structured for. Sometimes, this is just an unworkable scenario and the programmer has no choice but either fit a square peg in a round hole, or start over. However, there are numerous steps that can be taken in order to reduce the probability of this happening. These steps fall into a general set of principles called modular programming.

The premise of modular programming is that code is more flexible the more it is reduced into simple independent parts which take specific inputs and generate specific outputs irrespective of the surrounding code that they might be associated with. To illustrate this simply, consider the following pseudo code examples. Both functions calculate baseball statistics, one does so modularly while the other does not. In the first example shown below, the information about the players is held in global arrays, which are then accessed by both the main function and the return BattingAvg function.

Baseball.c

```
int teamAtBats[9];
int teamHits[9];


int main (void)
{

        //--------------------------------------------------------------------------------
        //populate arrays with each player's number of at bats and number of hits

        //--------------------------------------------------------------------------------

        //print out the scores using the function
```

```
    for (int i = 0; i<9; i++)
    {
        printf("player ".i." has a batting average of: ".returnBattingAvg(i));
    }

}

double returnBattingAvg(int player)
{
    return double(teamAtBats[player]/teamHits[player]);
}
```

This code might have a sound structure and be well planned; however its weakness is due to its lack of modularity. Consider a second implementation of this code Baseball2.c:

Baseball2.c

```
int main (void)
{
    int teamAtBats[9];
    int teamHits[9];
    //-------------------------------------------------------------------------------
    //populate arrays with each player's number of at bats and number of hits

    //-------------------------------------------------------------------------------

    //print out the scores using the function
    for (int i = 0; i<9; i++)
    {
        printf("player ");
        printf(i);
        printf(" has a batting average of: ");
        printf(returnBattingAvg(teamAtBats[i], teamHits[i]);

    }

}

double returnBattingAvg(int AtBats, int hits)
{
    return double(hits/AtBats);
}
```

In this scenario, it is hard to see how one of these programs is much better than the other. They both accomplish the same thing, and realistically, Baseball.c actually needs less overheard than baseball2.c. However it is important to note that in the case of Baseball.c, the returnBattingAvg makes several implicit assumptions about the rest of the code. Namely, that all the information it needs will be stored in the global variables teamAtBats and teamHits with each player's information associated with a certain number in both arrays. Consider the realistic scenario where it becomes beneficial to store this information a bit differently. Namely, imagine if it became easier to store this information in a single struct, and use an array of these structs. Consider the program Baseball3.c shown below.

Baseball3.c

```
struct baseballPlayer
{
        String Name;
        int AtBats;
        int Hits;
}


int main (void)
{
        baseballPlayer Players[9];
        //--------------------------------------------------------------------------------
        //populate array with each player's number of at bats and number of hits

        //--------------------------------------------------------------------------------

        //print out the scores using the function
        for (int i = 0; i<9; i++)
        {
                printf("player ");
                printf(Players[i].Name);
                printf(" has a batting average of: ");
                printf(returnBattingAvg(Players[i].AtBats, Players[i].Hits);
        }

}
```

```
double returnBattingAvg(int AtBats, int hits)
{
        return double(hits/AtBats);
}
```

There are numerous reasons to make such a change. A simple one is that structs often allow for easier information passing and storage. This will be a discussed in more detail later on. Baseball3.c is a very simple extension from Baseball2.c. It is very significant to note that there were no changes required to the function returnBattingAvg between Baseball2.c and Baseball3.c. This is because the returnBattingAvg function did not make any assumptions about the code it was contained within. On the other hand, upgrading Baseball.c to use the struct from Baseball3.c would require changing the returnBattingAvg function to match the new code. This may seem insignificant from this small example, however, when the size of the program scales, the amount of work that goes into making a seemingly small change like this goes up dramatically. This can lead to many problems. For instance, if a programmer attempts to make a change like this in code that isn't modular, it may become unreliable. In the case of robot control code, there may be a function that no longer works properly, but is rarely called, which could result in a bug that is very difficult to find. Perhaps even more disturbingly, the programmer may attempt to add functionality to the code without changing the structure at all. In this case, that might mean not adding the struct and just trying to keep track of all the information by hand. This can, and has in the past with the Ranger code led to "spaghetti". It is a clear example of trying to fit a square peg into a round hole because making the peg round becomes too

tedious. This problem tends to compound itself as this new version of the code is even more difficult to modify and may eventually need to be scrapped and started over. This is a short example of modular code and how, in combination with proper planning, it can dramatically decrease future workload.

Of course, the question still remains of how this applies to the actual Ranger code. Initially, all the code was built into one long file called mainV2A.c, which was about 5000 lines long. While this code originally was written to be fairly modular, it had evolved over time into “spaghetti”. This was primarily due to undisciplined programming practice during the evolution of the code. The specific weakness in the code that certainly contributed to this was the lack of protection. While the original code had started with a very discrete layering separating high level instructions and low level ones, the line had since been blurred.

In many ways this is a weakness inherent in using C instead of C++. C++ provides built in protection in the form of private variables and functions. This enforces modular coding practices and helps to prevent programmers from blurring the lines of high level and low level code. On the other hand, C does not have such a feature, and hence it is very tempting for a programmer to solve problems using hacks and “spaghetti” code which may be easier in the short run, but also make the code much more difficult to understand.

This is exactly what had happened to the previous ranger code, such that by the beginning of the semester it was very difficult to read and even more difficult to change. It quickly became apparent that the majority of the code would need to be scrapped and started over in order to bring it into line with the concepts of modularization. This was the control team's goal for the semester.

Methods and Results:

Splitting the code up

With the basic goals outlined, the control team set about implementing these changes. This began with a series of many meetings to decide exactly how to change the code. The team quickly found an initial goal of breaking the giant code into smaller pieces that could be changed independently. Actually doing this proved to be extremely time consuming, if not particularly difficult. During this stage, it should be stressed that there were no changes made to the code itself, it was all just moved around. For a complete listing of each block of code and its associated function, please see appendix A.

First Attempt at Modularization

Once this was completed, we began to look at what changes needed to be made to the actual code. The first target for restructuring was the actual walk controller. This was all located in the walkcontroller.c file and was a very clear

case of code that had turned into spaghetti. The code itself was about 1000 lines long and ran once per millisecond. Initially it would check the mode variable. This variable determined whether or not the walkcontroller was being run for the first time since entering walk mode or not. If it was being run for the first time, it would run all initialization commands and then end. Otherwise it would run the actual walk controller code. The real problems began once the robot started walking. The entire structure of the code was a large series of nested if statements checking to see if certain conditions were met under certain circumstances and then reacting based on them. The code used a loosely defined state machine. Each state in fact had numerous sub states which were checked using nested conditionals. These nested conditionals that broke up the states were exactly what led to the “spaghetti”.

Additionally, the state machine was not linear, which further complicated things. This is to say that the sequence of states that the robot entered was not always the same. During our meetings at the beginning of the semester, we had decided that always going through the same sequence of states was important to the robustness of the robot. It makes debugging significantly easier as any missed states inherently mean something has gone wrong. On the other hand, if the sequence is not the same, a missed state probably won’t mean anything. Thus, when something does go wrong, it is much harder to pin down the problem quickly.

Initially, we thought the code for the walk controller could be salvaged and just converted into a more usable and flexible version. Andrey Turovsky and I set about trying to reorganize the walk controller into more modular sections that did not rely on the other sections of the code and could be easily read and changed. After about 20 hours of work though, we were forced to accept that the code needed to be done over.

On the surface it might seem like this work was wasted, but in reality it was actually very instructive. By spending so much time using and reorganizing the old code, we had a much deeper understanding of exactly what it was doing, and more importantly were able to come up with a structure that could handle the fairly complex decision making process while keeping it quite modular.

The Structure of the New Code

The structure we came up with relied on breaking the walk controller into three main sections: an actions section, a set section, and a transitions section. Furthermore, the code would make decisions using three independent state machines: one for the hip, one for the inner feet, and one for the outer feet. The actual structure of the walk controller was then fairly simple. The portion of the code that does the bulk of the work can be seen below in walkcontroller.c.

WalkController.c

```
if (decisions(Walk) != 0)
{
	set(&Walk);
}
actions(&Walk);
```

This is of course not all of the code in the file, just the relevant part. Every time walkcontroller.c runs (once per millisecond), it first calls the decisions function. It passes it the walk struct which will be discussed in more detail shortly. The decisions function examines the present state of each state machine, and also examines all of the other state variables associated with the robot. If any of the state machines need to change states, the decisions function will return a non-zero value. If no state machine transitions, it will return a zero. The value that is returned from the decisions function is then used to determine whether or not the set function needs to be run at all. In the event that decisions returns a zero (no state transitions), set will not run. If decisions returns a non-zero value, set will run. The set function itself is used to set all variables required for walking within any given state. This is similar to setting initial conditions for each state. Finally, the actions function actually looked at the present states of all three state machines and decided exactly what to do with each motor associated with each state machine.

New Information Passing Method

While this structure was very helpful for making the code more readable, it was only a portion of what made it more flexible. The other significant change that increased the modularity of the code was adding a struct which contained all the information that needed to be passed back and forth between the actions, set and decisions functions. A struct is just a bundle of variables which can be easily passed around. This was the basic approach we used to solve the issue of

interdependent data. It allows the data to be passed in a simple, concise manner between the functions. The real benefit is that it dramatically reduces the number of global variables, and makes the entire structure of the walk controller less interdependent. Previously, if a programmer wanted to add a new variable to be passed between the three main functions, they would need to add an extern line at the top of every function that needed to be used and just as importantly, they would need to make sure that the variable name they picked was not used locally in any of the other modules associated with the entire robot code. This could become an enormous problem very quickly because, while the program will still compile, there will be variables all over it that don't necessarily contain the data the programmer expects because they have been changed elsewhere. This becomes an even larger problem as more people work on the code. Any identically-named variables would have immediately become a liability that would generate difficult-to-locate errors. By keeping as much information local as possible, it prevents this from becoming an issue and decreases the levels of interconnectedness across the entire structure of the program.

With the struct system of passing information that is effectively local, any changes a programmer needs to make to information being passed only need to be made once and pose no risk of accidentally conflicting with other variables. There are some small drawbacks associated with using it. To begin with, it requires a deeper understanding of programming than just using a global variable. This is true both at the obvious level of needing to understand how to

use a struct in general, and at a somewhat deeper level. To use a struct like this requires that the user have some understanding of the difference between passing by value and passing by reference. Ordinarily when a function receives an argument, it is actually only receiving a copy of the value. Any changes made to this new variable will not actually affect the original variable. This is called passing by value. On the other hand, passing by reference actually passes the variable itself to the function. Consequently, any changes made to the variable within the function are reflected everywhere in the code. To actually understand the implementation, the programmer must understand the premise of pointers which are a fairly challenging subject. Fortunately, it is not necessary to understand pointers in order to actually use this method. All that is necessary is following the correct syntax depending on if the programmer wants to pass by reference or pass by value. The differences in syntax are fairly simple and discussed in appendix B.

Results

The results seem to speak for themselves. When the robot was setting the record, it was using this new structure of the code. While this is clearly an indicator that the new code was successful, it only tells a part of the story. Before the final walk that broke the record, there was of course fairly extensive testing. During this time, it was extremely beneficial to have all of the state machines explicitly defined. This allowed us to print them on the screen and figure out exactly what states were causing the robot to fall, or more commonly, what states were accidentally being skipped. In the previous version of the code this would

have been very difficult. Since the sequence of states wasn't always the same, a skipped state did not automatically tell the debugger where the error was. Secondly, once the problem state was identified, the debugger also had to identify which sub state was causing trouble. Both of these processes required walking through all of the code and keeping track of all the variables at all times, which is of course a difficult and tedious process. While it is possible to debug like this, it is much easier to when you can immediately narrow the problem down to three or four lines. Much credit here needs to be given to Stephane Constantin whose matlab data display program was invaluable in the debugging process. This program enabled us to make full use of the new structure in the debugging process.

Between the new structure and the new debugging tools at our disposal, it was actually a fairly straightforward process to determine exactly what happened any time the Ranger fell. This enabled us to quickly eliminate any logic errors in the code and identify if the code was at fault or not when the robot fell.

After the robot broke the record, there were numerous attempts to make it do various other "behaviors". These included walking backwards, and coming to a stop. In order to quickly facilitate this, I reworked the decisions, set, and actions functions in a few weeks in order to allow a selector to choose a set of actions, decisions, and set functions to match each behavior. This allowed a programmer to make a new behavior and use it by pressing a different button on the back of

the robot. Unfortunately this is not a long term solution since it still does not allow the robot to switch in the middle of one behavior to another, nor does it allow the robot to have a list of behaviors that it executes in order. Both of these are eventual goals and will require a lot of work in order to implement in a way that still fits the general premises of modular code.

Discussion:

Since the robot successfully broke the distance walking record, the control team has been hard at work planning the next generation of the code. The goal has been to come up with a new structure which will hopefully allow for as much flexibility as possible. We started by trying to define a language for talking about all the elements of the code. We found that it was actually quite difficult to talk about different approaches to the structure without making sure everyone knew exactly what an action was as opposed to a behavior. This often led to miscommunication and was in general an annoyance. We cleared that up fairly quickly by spending some time discussing exactly what each term meant. The actual names are not important so long as everyone working on the code knows exactly what each one means before trying to exchange ideas about them.

Once this was done, we set about planning a new structure for the code. One that would hopefully take advantage of the eventual move to C++, and would also allow for all the flexibility we would need. These new design goals we have been planning for include allowing the robot to switch seamlessly from one

control structure to another. This is critical if we want the robot to be able to do things like walk forward, come to a stop and start walking backwards all by itself.

Another goal we are working on it to have all behaviors control all the motors by just using a single library of possible actions. This library of actions would include things like swinging the leg forward, allowing it to swing freely etc. Each state should then have single action it calls from this library for any given motor. Hopefully this will increase the amount of the code that gets reused and in doing so, will increase the reliability of the robot.

While nothing is final presently, we do have a basic structure for the future implementation of the code that hopefully accommodates all of these constraints. We define a behavior as a coordinated sequence of motions from the robot. Behaviors include walking forward, walking backwards, running etc. These individual behaviors will run using the same type of state machines currently employed on the robot. These state machines will rely on a library of actions. Actions are similar to behaviors, but on a smaller scale. Actions will include such motions as swinging the leg forward with certain target speeds, positions etc. Finally, these actions will all operate using a series of micropolicies. These micropolicies are just simple linear controls dedicated to each motor. They will receive simple inputs like torque, or PWMs for their motor and convert this to an actual output to the motor. A graphical representation of the top two levels of the state machine is shown below in Figure 1.

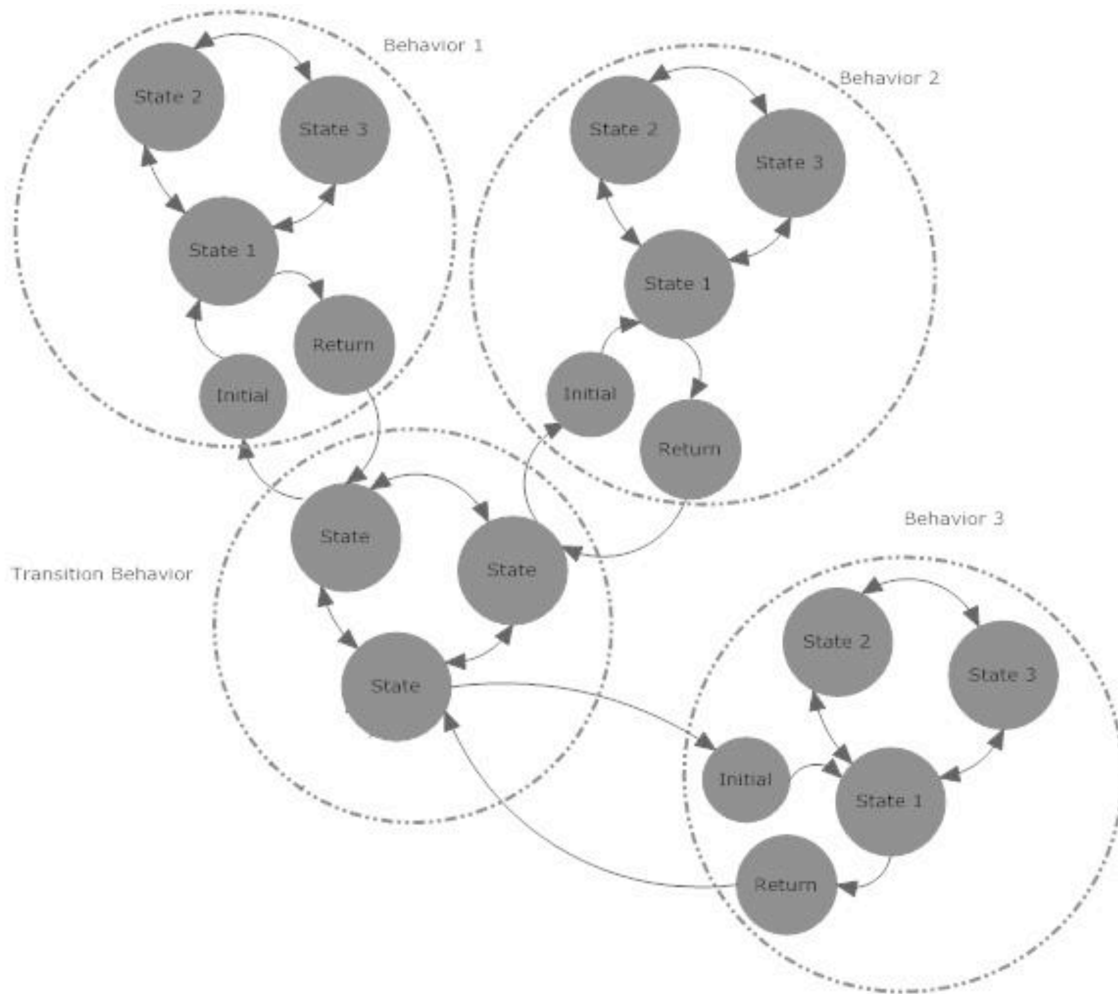

Figure 1

In Figure 1, the large dashed circles represent the top level of the state machine, while the gray circles represent the state machines within each behavior. It also demonstrates how we could presumably move from one behavior to another seamlessly. There will be a transition behavior (possibly

several), and each behavior will be responsible for both getting from the transition behavior and going back to the transition behavior. Each behavior then uses its own state machine to select from a library of actions, which then set the appropriate micropolicies for each motor.

Conclusion:

While there was clearly a lot of progress made this semester, it was still limited due to certain constraints. The biggest constraint we had to deal with was the use of C instead of C++. In general, object orientation naturally lends itself to encapsulation. It is possible to use good programming practice to simulate the natural encapsulation that comes from using objects, but it can quickly become tedious. There were numerous occasions while we were writing this code that we thought of easy ways to implement something in C++ that turned out to be quite challenging in C.

The basic conclusion from this semester is that as the robots get more complicated with more sensors and more information to process, it is vital that the code be very well planned. If the code is properly planned, it will be as flexible as possible and hopefully will evolve in a way that does not become unwieldy and unchangeable.

At the same time though, after a whole semester of hard work, the code has made undeniable improvements. It is easier to both debug and evolve. Perhaps more importantly, we have developed a frame work for both short term evolution and the long term changes that we eventually plan on making. This will hopefully go a long way towards preventing the code from turning into "spaghetti" as it changes in the future.

Acknowledgment:

It is important to acknowledge those who contributed to this project. Bram Hendriksen was extremely helpful in coming up with the heart of the new controller. Andrey Turovsky was also very helpful, especially with the initial design of the new structure, and even more so in the attempts at using the old code. Greg Stiesberg also was instrumental in the development of the most recent plan for the future code. Finally, all of this would have of course been impossible without help from both Jason Cortell and Andy Ruina who were both vital to the entire process.

Appendix:

It should also be noted that I spent a solid two weeks working on just debugging the robot in preparation for breaking the record. During this time I helped Bram with debugging code, and debugging electronics in order to ensure that the robot successfully walked before he left.

Appendix A:

Button.c: Handles the button presses from the back of the robot
CAN.c : Handles CAN bus to satellite processors
Encoder.c :Handles all the encoders for the hip

Foot.c: Handles low level control of the feet
LCD.c: Controls the LCD display screen
PWM.c: Low Level motor control
Sensor.c: Low Level Sensor control
TMRC0_Main_Loop_interrupt.c: The main loop, runs once per millisecond to control almost all robot operations
WalkController.c: Wrapper for the walk controller
Actions.c: Makes motor decisions based on present state of state machines
Set.c: Set initial values on transition
Decisions.c: Determines when state machine needs to transition

Appendix B:

The pointer syntax is fairly straightforward. In the function declaration, the user will need to include the pointer symbol. For instance, in the case of the set function, the declaration would look like this: `void set(struct WalkVars*);` In this case, the struct we have defined is called the WalkVars struct. The only difference between the pointer version of the declaration and the regular declaration is the asterisk.

In the function call, the syntax is again a bit different for pointers than it is for regular calls. The call to the set function would look like this: `set(&Walk);` The difference between using the pointer and not being the amperstand.

In the function definition itself, there are two more syntax changes to keep in mind. The first is in the first line of the definition. In the case of the set function, the first line looks like this: `void set(struct WalkVars *Walk)` , with the only difference being the asterisk. Finally, any time the programmer wishes to access an

element from the Walk struct, it must be accessed using an arrow like this:

Walk->I.

By using these simple changes in syntax, the programmer can pass by reference instead of by value. This enables him or her to change the actual values within the original struct, rather than changing the values of a copied version of it.

Appendix C:

WalkController.h

```
//checks to make sure this is only defined one time
#ifndef ALREADY_WALK_CONTROLLER_H
#define ALREADY_WALK_CONTROLLER_H
	// Hip states
	#define MC_HIP_SWING_INNER		ONEFF
	#define MC_HIP_FREESWING_INNER	TWOFF
	#define MC_HIP_EMERGENCY_INNER	THREEFF
	#define MC_HIP_SWING_OUTER			FOURFF
	#define MC_HIP_FREESWING_OUTER	FIVEFF
	#define MC_HIP_EMERGENCY_OUTER	SIXFF


	// foot states Inner
	#define MC_FEET_FLIP_UP		ONEFF
	#define MC_FEET_LANDING		TWOFF
	#define MC_FEET_STANCE		THREEFF
	#define MC_FEET_PUSHOFF		FOURFF
	#define MC_FEET_STANCELAND	FIVEFF

	struct WalkVars
	{
		ffloat HipSwingPWMLevel;
		unsigned long targettime;
		ffloat I;
		ffloat angleland;
		ffloat LandingAngle;
		ffloat FeetStanceAngle;
		ffloat Pushoff;
		ffloat PushOffStartAngle;
```

```
		ffloat DeltaPushOff;
		ffloat DecreaseRate;
		ffloat KpFeetStance;
		ffloat KpFeetLand;

	};
	//Walk Controller Function Prototypes
	void actions(struct WalkVars*);
	int decisions(struct WalkVars);
	void reset(struct WalkVars*);

#endif //ALREADY_WALK_CONTROLLER_H
```

WalkController.c

```
#include "headers.h"

extern ffloat GBL_Data[];
bool impact;
struct WalkVars Walk;
ffloat Prev_Stance_Leg; //= ONEFF;
ffloat Error, P,I,D,ForcePID;


void WalkController(int mode)
{
	GBL_Data[TestOutput7]=mode;
	if(mode==1)	//=====START UP MODE======
	{
		// Start up mode runs each time the walk mode is started
		// This mode sets all static variables to the correct values

		GBL_Data[StanceLeg]	= MC_OUTERLEG;
		Prev_Stance_Leg			= GBL_Data[StanceLeg];
		GBL_Data[WalkHipState]		= MC_HIP_FREESWING_INNER;
		GBL_Data[FeetStateInner] = MC_FEET_LANDING;
		GBL_Data[FeetStateOuter] = MC_FEET_PUSHOFF;
		GBL_Data[TotalWalkedDistance] = ZEROFF;
		GBL_Data[NumberSteps] = ZEROFF;
		LCDWrite('5',0);
		LCDWrite('3',1);
		Walk.targettime = 0;
		Walk.Pushoff = TWOFF;
		Walk.angleland = 0;
		Walk.LandingAngle = 1.6;
		I = ZEROFF;
	}
	else
	{
		if (decisions(Walk) != 0)
		{

			set(&Walk);

		}
```

```
		actions(&Walk);
	}

}
```

Decisions.c

```
#include "headers.h"

extern ffloat GBL_Data[];
extern ffloat GBL_Param[];
extern ffloat Prev_Stance_Leg;
extern unsigned long GBL_Elapsed_mS;
extern bool impact;
int Transition;
extern int counts;

int decisions(struct WalkVars Walk)
{
	Transition = 0;

	//CHECK FOR IMPACT
	if(GBL_Data[StanceLeg] != Prev_Stance_Leg)
	{
		impact = 1;
		Prev_Stance_Leg = GBL_Data[StanceLeg];
	}
	else
	{
		impact = 0;
	}

	//Detect errors
	DetectErrors();



	switch(GBL_Data[WalkHipState])
	{
		//handle the HIP state machine
		case MC_HIP_SWING_INNER:
			if(GBL_Elapsed_mS > Walk.targettime)
			{
				GBL_Data[WalkHipState] = MC_HIP_FREESWING_INNER;
				//Transition to FREESWING_INNER once elapsed time passes target
time
				Transition |= 1;
			}
			LCDWriteWord("SwingI");

		break;

		case MC_HIP_FREESWING_INNER:
```

```
			if ((FFlt(GBL_Data[AngleJointHip],GBL_Param[FixAngle]))  &
(FFltz(GBL_Data[AngleRateEncoderHip])) & (FFgtz(GBL_Data[AbsAngleInnerLeg])))
			{
				GBL_Data[WalkHipState] = MC_HIP_EMERGENCY_INNER;
				//Switch state to Emergency Inner
				Transition |= 1;
			}

			if (impact)
			{
				GBL_Data[WalkHipState] = MC_HIP_SWING_OUTER;
				//transition to swing outer in the event that we're in freeswing_inner
and there

				//is an impact
				Transition |= 1;
			}
			LCDWriteWord("FreeI");
		break;

		case MC_HIP_EMERGENCY_INNER:
			if (impact)
			{
				GBL_Data[WalkHipState] = MC_HIP_SWING_OUTER;
				Transition |= 1;
				//transition to swing_outer in the case that there is impact
				//and we were in emergency inner
			}
			LCDWriteWord("EMERI");
			BeepFreq(50,200);
			BlinkLEDColor(255,255,0,50,200);
			StoreError(21);

		break;

		case MC_HIP_SWING_OUTER:
			if(GBL_Elapsed_mS > Walk.targettime)
			{
				GBL_Data[WalkHipState] = MC_HIP_FREESWING_OUTER;
				//Transition to FREESWING_OUTER once elapsed time passes target
time

				Transition |= 1;
			}
			LCDWriteWord("SwingO");
		break;

		case MC_HIP_FREESWING_OUTER:
			if ((FFgt(GBL_Data[AngleJointHip],FFneg(GBL_Param[FixAngle])))  &
(FFgtz(GBL_Data[AngleRateEncoderHip]))& (FFgtz(GBL_Data[AbsAngleOuterLeg])))
			{
				GBL_Data[WalkHipState] = MC_HIP_EMERGENCY_OUTER;
				//Switch state to Emergency Inner
				Transition |= 1;
			}

			if (impact)
			{
```

```
					GBL_Data[WalkHipState] = MC_HIP_SWING_INNER;
					//transition to swing inter in the event that we're in freeswing_outer and
there
					//is an impact
					Transition |= 1;
				}
					LCDWriteWord("FreeO");
			break;

			case MC_HIP_EMERGENCY_OUTER:
				if (impact)
				{
					GBL_Data[WalkHipState] = MC_HIP_SWING_INNER;
					//transition to swing_inner in the case that there is impact
					//and we were in emergency outer
					Transition |= 1;
				}
				LCDWriteWord("EMERO");
				BeepFreq(50,200);
				BlinkLEDColor(255,255,0,50,200);
				StoreError(22);
			break;
		}


		switch(GBL_Data[FeetStateInner])
		{
			//handle the inner foot state machine
			case MC_FEET_FLIP_UP:
				if ((FFgt(GBL_Data[InnerFootHight], GBL_Param[FlipDownClearance]) &
FFgt(GBL_Data[AngleJointHip], GBL_Param[FlipUpStart])))
				{
					GBL_Data[FeetStateInner] = MC_FEET_LANDING;
				}

				if (impact)
				{
				GBL_Data[FeetStateInner] = MC_FEET_STANCELAND;
				BeepFreq(250,4000);
				BlinkLEDColor(0,0, 255,50,2000);
				StoreError(23);
				}
			break;

			case MC_FEET_LANDING:
				if (impact)
				{
					GBL_Data[FeetStateInner] = MC_FEET_STANCELAND;

				}
			break;

			case MC_FEET_STANCELAND:
				if (FFlt(Walk.angleland, Walk.FeetStanceAngle))
				{
					GBL_Data[FeetStateInner] = MC_FEET_STANCE;
```

```
				}
				if (impact)
				{
					GBL_Data[FeetStateInner] = MC_FEET_FLIP_UP;

					BeepFreq(250,4000);
					BlinkLEDColor(0,255, 0,50,2000);
					StoreError(24);
				}
			break;

			case MC_FEET_STANCE:
				if (FFlt(GBL_Data[AbsAngleInnerLeg],Walk.PushOffStartAngle))
				{
					GBL_Data[FeetStateInner] = MC_FEET_PUSHOFF;
					Transition |= 2;
				}

				if (impact)
				{
					GBL_Data[FeetStateInner] = MC_FEET_FLIP_UP;
					BeepFreq(250,4000);
					BlinkLEDColor(255,0, 0,50,2000);
					StoreError(25);
				}
			break;

			case MC_FEET_PUSHOFF:
				if (impact)
				{
					GBL_Data[FeetStateInner] = MC_FEET_FLIP_UP;
				}

			break;
		}
		switch(GBL_Data[FeetStateOuter])
		{
			//handle the outer foot state machine
			case MC_FEET_FLIP_UP:
				if ((FFgt(GBL_Data[OuterFootHight], GBL_Param[FlipDownClearance]) &
FFlt(GBL_Data[AngleJointHip], FFneg(GBL_Param[FlipUpStart]))))
				{
					GBL_Data[FeetStateOuter] = MC_FEET_LANDING;
				}

				if (impact)
				{
					GBL_Data[FeetStateOuter] = MC_FEET_STANCELAND;
					BeepFreq(250,4000);
					BlinkLEDColor(255,255, 255,50,2000);
					StoreError(26);
				}
			break;

			case MC_FEET_LANDING:
```

```
			if (impact)
			{
				GBL_Data[FeetStateOuter] = MC_FEET_STANCELAND;

			}
		break;

		case MC_FEET_STANCELAND:
			if (FFlt(Walk.angleland, Walk.FeetStanceAngle))
			{
				GBL_Data[FeetStateOuter] = MC_FEET_STANCE;

			}
			if (impact)
			{
				GBL_Data[FeetStateOuter] = MC_FEET_FLIP_UP;
				BeepFreq(250,4000);
				BlinkLEDColor(255,0, 255,50,2000);
				StoreError(27);
			}
		break;


		case MC_FEET_STANCE:
			if (FFlt(GBL_Data[AbsAngleOuterLeg],Walk.PushOffStartAngle))
			{
				GBL_Data[FeetStateOuter] = MC_FEET_PUSHOFF;
				Transition |= 4;
			}

			if (impact)
			{
				GBL_Data[FeetStateOuter] = MC_FEET_FLIP_UP;
				BeepFreq(250,4000);
				BlinkLEDColor(255,255, 0,50,2000);
				StoreError(28);
			}

		break;

		case MC_FEET_PUSHOFF:
			if (impact)
			{
				GBL_Data[FeetStateOuter] = MC_FEET_FLIP_UP;
			}
		break;
	}
	return Transition;
}
```

Set.c

```
#include "headers.h"
```

```
extern int counts;
extern ffloat GBL_Data[];
extern ffloat GBL_Param[];
extern unsigned long GBL_Elapsed_mS;
extern int Transition;


void set(struct WalkVars *Walk)
{
   int temptransition=0;
            if ((Transition&1) != 0)//run on hip state machine transition
            {
            ffloat temp2;
            temp2 = FFadd(0x000B7080,FFmult(FFsub(GBL_Param[GoalAngle], 0xFFFF4000),
0x000D4E20));

            if (GBL_Data[WalkHipState]==MC_HIP_SWING_OUTER)
            {
                        Walk->HipSwingPWMLevel= FFsub(FFmult(temp2,
FFsub(GBL_Data[AbsAngleOuterLeg],0x000A805)), 0x00065D11);
                        Walk->targettime= GBL_Elapsed_mS+325;
                        Walk->I=ZEROFF;
            }

            if (GBL_Data[WalkHipState]==MC_HIP_SWING_INNER)
            {
                        Walk->HipSwingPWMLevel= FFsub(FFmult(temp2,
FFsub(GBL_Data[AbsAngleInnerLeg],0x000A805)), 0x00065D11);
                        Walk->targettime= GBL_Elapsed_mS+325;
                        Walk->I=ZEROFF;
            }
            counts = 0;

            }


            if ((Transition&2) != 0)//run on inner state machine transition
            {
                        if (GBL_Data[FeetStateInner]==MC_FEET_PUSHOFF)
                        {
                                    temptransition = Transition;
                                    Walk -> Pushoff = FFadd(GBL_Data[AngleJointInner], Walk -> DeltaPushOff);
                        }

                        if (GBL_Data[FeetStateInner]==MC_FEET_STANCELAND)
                        {
                                    Walk -> angleland = Walk -> LandingAngle;
                        }
            }


            if ((Transition&4) != 0)//run on outer state machine transition
            {
                        if (GBL_Data[FeetStateOuter]==MC_FEET_PUSHOFF)
                        {
```

```
				Walk -> Pushoff = FFadd(GBL_Data[AngleJointOuter], Walk ->
DeltaPushOff);
			}


			if (GBL_Data[FeetStateOuter]==MC_FEET_STANCELAND)
			{
				Walk -> angleland = Walk -> LandingAngle;
			}
		}


		//Run every Reset
Walk -> LandingAngle	=	FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[LandingAngleNormal])),(FFmult(GBL_Data[Ksteering],
GBL_Param[LandingAngleSteering])));
Walk -> FeetStanceAngle =	FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[FeetStanceAngleNormal])),(FFmult(GBL_Data[Ksteering],
GBL_Param[FeetStanceAngleSteering])));
Walk -> DeltaPushOff	=	FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[PushOffNormal])),(FFmult(GBL_Data[Ksteering], GBL_Param[PushOffSteering])));
Walk -> PushOffStartAngle = FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[PushOffStartAngleNormal])),(FFmult(GBL_Data[Ksteering],
GBL_Param[PushOffStartAngleSteering])));
Walk -> DecreaseRate	=	FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[DecreaseRateNormal])),(FFmult(GBL_Data[Ksteering], TWOFF)));
Walk -> KpFeetLand		=	FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[KpFeetLandNormal])),(FFmult(GBL_Data[Ksteering],GBL_Param[KpFeetLandSteering])));
Walk -> KpFeetStance	=	FFadd((FFmult(GBL_Data[Knormal],
GBL_Param[KpFeetStanceNormal])),(FFmult(GBL_Data[Ksteering],GBL_Param[KpFeetStanceSteering])
));
GBL_Data[TargetTime] = U32int2FFloat(Walk->targettime);
		Transition = 0;


}
```

Actions.c

```
#include "headers.h"

extern ffloat GBL_Data[];
extern ffloat GBL_Param[];
extern ffloat Prev_Stance_Leg;
extern bool impact;
int counts;
extern ffloat Error, P,I,D,ForcePID;



void actions(struct WalkVars *Walk)
{
```

```
	/////// HIP state machine
	switch(GBL_Data[WalkHipState])
	{
		case MC_HIP_SWING_INNER:
			if (counts < 200) //ramp for 200 ms
			{
				counts++;
			}
			GBL_Data[PWMDesiredHip] =  (FFmult(FFdiv(Walk-
>HipSwingPWMLevel,0x00086400),S16int2FFloat(counts)));
		break;

		case MC_HIP_FREESWING_INNER:
			GBL_Data[PWMDesiredHip] = ZEROFF;
		break;

		case MC_HIP_EMERGENCY_INNER:
			Error = FFsub((GBL_Param[FixAngle]),GBL_Data[AngleJointHip]);
			//PID control of Hip
			P= FFmult(GBL_Param[KEmergency],Error);
			Walk -> I=FFadd(Walk ->I,FFmult(FFmult(0x000106270,LOOPTIME),Error));
			D= FFmult(0x000A4EC0,(GBL_Data[AngleRateEncoderHip]));
			ForcePID = FFadd(P,FFsub(I,D));  //difference between Fsubscribed and
Fmeasured times a PID action
			GBL_Data[PWMDesiredHip] = ForcePID ;
		break;

		case MC_HIP_SWING_OUTER:
			if (counts < 200) //ramp for 200 ms
			{
			counts++;
			}
			GBL_Data[PWMDesiredHip] =  FFneg(FFmult(FFdiv(Walk-
>HipSwingPWMLevel,0x00086400),S16int2FFloat(counts)));
		break;

		case MC_HIP_FREESWING_OUTER:
			GBL_Data[PWMDesiredHip] = ZEROFF;
		break;

		case MC_HIP_EMERGENCY_OUTER:
			Error = FFsub(FFneg(GBL_Param[FixAngle]),GBL_Data[AngleJointHip]);

			//PID control of Hip
			P= FFmult(GBL_Param[KEmergency],Error);
			Walk ->I=FFadd(Walk->I,FFmult(FFmult(0x000106270,LOOPTIME),Error));
			D= FFmult(0x000A4EC0,(GBL_Data[AngleRateEncoderHip]));
			ForcePID = FFadd(P,FFsub(I,D));  //difference between Fsubscribed and
Fmeasured times a PID action
			GBL_Data[PWMDesiredHip] = ForcePID ;
		break;
	}

	switch(GBL_Data[FeetStateInner])
	{
		case MC_FEET_FLIP_UP:
```

```
				Error = FFsub(0xFFFE6666,GBL_Data[AngleJointInner]);
				GBL_Data[PWMDesiredInner] = 
FFsub(FFmult(GBL_Param[KpFeetFlipUp],Error), FFmult(GBL_Param[KdFeetFlipUp], 
GBL_Data[AngleRateEncoderInner]));
			break;

			case MC_FEET_LANDING:
				Error = FFadd(Walk -> LandingAngle,GBL_Data[AbsAngleInnerFeet]);
				GBL_Data[PWMDesiredInner] = FFmult(Walk->KpFeetLand,Error);
				//GBL_Data[TestOutput6] = Walk->KpFeetLand;
			break;

			case MC_FEET_STANCELAND:
				Walk -> angleland = FFsub(Walk->angleland, Walk->DecreaseRate);
				Error = FFadd(Walk->angleland,GBL_Data[AbsAngleInnerFeet]);

				GBL_Data[PWMDesiredInner] = FFmult(Walk->KpFeetStance,Error);
			break;

			case MC_FEET_STANCE:
				Error = FFadd(Walk -> FeetStanceAngle,GBL_Data[AbsAngleInnerFeet]);

				GBL_Data[PWMDesiredInner] = FFmult(Walk->KpFeetStance,Error);
			break;

			case MC_FEET_PUSHOFF:
				Error = FFsub(Walk->Pushoff,GBL_Data[AngleJointInner]);

				GBL_Data[PWMDesiredInner] = FFmult(GBL_Param[KpFeetPushOff],Error);
				//GBL_Data[TestOutput6] = (*Walk).Pushoff;
			break;
		}


		switch(GBL_Data[FeetStateOuter])
		{
			case MC_FEET_FLIP_UP:
				Error = FFsub(0xFFFE6666,GBL_Data[AngleJointOuter]);
				GBL_Data[PWMDesiredOuter] = 
FFsub(FFmult(GBL_Param[KpFeetFlipUp],Error),FFmult(GBL_Param[KdFeetFlipUp], 
GBL_Data[AngleRateEncoderOuter]));
			break;

			case MC_FEET_LANDING:
				Error = FFadd(Walk ->LandingAngle ,GBL_Data[AbsAngleOuterFeet]);
				GBL_Data[PWMDesiredOuter] = FFmult(Walk->KpFeetLand,Error);
			//	GBL_Data[TestOutput6] = Walk->KpFeetLand;
			break;

			case MC_FEET_STANCELAND:
				Walk -> angleland = FFsub(Walk->angleland,Walk->DecreaseRate);
				Error = FFadd(Walk->angleland,GBL_Data[AbsAngleOuterFeet]);

				GBL_Data[PWMDesiredOuter] = FFmult(Walk->KpFeetStance,Error);
			break;
```

```
			case MC_FEET_STANCE:
				Error = FFadd(Walk -> FeetStanceAngle ,GBL_Data[AbsAngleOuterFeet]);

				GBL_Data[PWMDesiredOuter] = FFmult(Walk->KpFeetStance,Error);
			break;

			case MC_FEET_PUSHOFF:
				Error = FFsub(Walk->Pushoff,GBL_Data[AngleJointOuter]);

				GBL_Data[PWMDesiredOuter] = FFmult(GBL_Param[KpFeetPushOff],Error);
				//GBL_Data[TestOutput6] = Walk->Pushoff;
			break;
		}


	}
```