AN ARCHITECTURE FOR A MODULAR ROBOT

NSF Summer Undergraduate Fellowship in Sensor Technologies
Kapil Kedia, MEAM, University of Pennsylvania
Vincent Marshall, EE, University of Pennsylvania
Collaborators: Darnel Degand and Robert Breslawski
Advisor: Dr. James Ostrowski

ABSTRACT

This paper describes the design of an architecture to construct and control a modular robot. This architecture is to serve as a logical and electrical framework for a small mobile robot that has logical and functional separation between its modules. This architecture has a master/slave scheme, where a master "brain" module controls multiple function modules. The communication between the master and the slaves takes place over a serial bus on which the master initiates all communication in order to avoid bus access problems. The architecture was tested using BASIC Stamp II and PIC microprocessors. Testing with both actuator and sensor modules was successful when using this hardware.

1. INTRODUCTION

Traditionally, robots have been designed monolithically - as a single piece to perform a certain task. A good example would be factory manipulator arms. Historically, they have been designed to complete specific tasks, such as welding or painting. A prominent example of a mobile monolithic robot is the Mars Pathfinder. It was also designed as a single entity to perform a single task; exploring the surface of Mars. Building a robot with a modular architecture could lend certain advantages over monolithic robots.

A modular robot is built from physically separate sub-units, each contributing functionally to the operation of the robot. A modular robot would be much more versatile because a single robot could be used to complete multiple tasks with the selection of the most appropriate module for the task. This could be especially useful in factory manipulator arms because a single arm could be used with a modular tool at the end. This could lead to reduced costs, because a single modular robot could be used in place of ten task specific monolithic robots. Another advantage of a modular robot is that is would be more robust than a traditional robot because it could continue to operate more easily with a broken part, and a broken part could be more easily replaced. This advantage would be very beneficial in space applications like the Mars Pathfinder; because it would very difficult to fix a robot that is millions of miles away. Also, a modular robot could facilitate prototyping and testing because a robot could be easily assembled and operated using preexisting modules. Finally, modular robot could also serve as a testbed for new technologies.

There are many different ways for a robot to be considered modular. Modules can have functional and logical separation. Functional separation of modules refers to a physical modularity that could be mechanical and/or electrical. Logical separation refers to the division of control between the modules. Compared to many research topics, relatively little work has been done the field of modular robotics. Some other work includes Mark Yim's polypod project at Xerox Parc[1]. Each module of this robot is completely functionally and logically separate and identical so that the robots could reconfigured into many different locomotory behaviors. Another type of modular robot is the Hexplorer project being done at the University of Waterloo in Canada[2]. This robot is not completely functionally modular, as it cannot be physically reconfigured, but it is organized into subunits so that it is logically modular.

The ultimate goal of this project to build a small mobile robot that is both functionally and logically modular. The scope of this research project is to design an architecture that serves as an electrical and logical framework for this robot.

2. MODULARITY

2.1 Master/Slave Division

In this architecture, the modules are configured using a master/slave scheme. A single master, or "brain", module controls multiple "slave" function modules as shown in figure one. The brain of the robot orchestrates the behavior of the slave function modules and will also be referred to as the "master" throughout this paper. The slave function modules are designed to perform specific tasks, such as actuation and sensing, and will also be referred to as "slaves" or "modules" for the remainder of the paper.

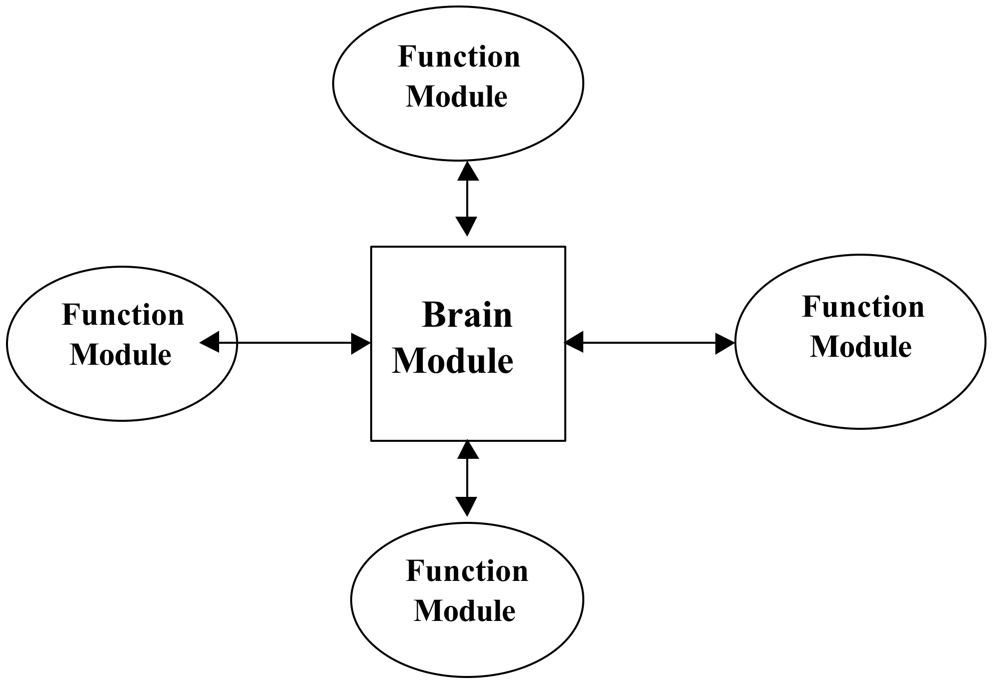

Figure 1: Brain/ Function Module

Control of the robot can be logically broken down into the concepts of high-level and low-level control. High-level control refers to coordination of the available complex behaviors of the robot to produce some overall behavior. Low-level control refers to the coordination of simple behaviors, all of which are associated with a specific module on the robot in our case, to produce complex behaviors. In our architecture, slaves perform the low-level control and therefore exist to produce complex behaviors for high-level control from their repertoire of simple functions. A sonar distance-sensing module, for example, has certain simple behaviors available to it: the ability to initiate a sonar ping, the ability to recognize the return of a sonar ping, the ability to measure times, and the ability to communicate with a master module. The task of this module is to respond with the distance from the robot to some object in the environment when queried by the master module for this information. In our architecture, the brain performs the high-level control. Drawing on the example above; some of the tasks of a brain module, depending on the current configuration of the robot, might include retrieving distance readings from a sonar module and then issuing commands reflecting decisions based on this information to a locomotion module. A logically analogous concept to the division of control in our architecture comes from computer programming. The brain module is analogous to the main program and the slave modules are analogous to subroutines in the sense that the brain issues the overall commands and calls upon the subroutines to perform the more specific tasks. To extend the analogy, the slave modules can be reused in different modular robots, can be interchanged, and can be replaced with different modules that perform a similar task and offer the same interface to the outside world. Furthermore, the brain module need not be concerned with any of the specifics of the functioning of the slave modules - it relates to them only in terms of their outward behavior.

The division of high-level and low-level control forces us to make determinations, which can be on a per-module basis, of where the line between high and low level control will be drawn. Different levels of abstraction will obviously have different advantages and disadvantages and must be selected accordingly. To illustrate, consider the tasks involved in controlling a walking, legged robot. The overall path of the robot must be chosen, the gait must be controlled so that walking actually takes place, and the segments of the individual legs must be controlled to create the aforementioned gait. Several different levels of abstraction between high and low level control are possible in this case. Most possible solutions will begin with giving each leg module the ability to take a step. From that point, one solution is to both determine the path and control the gait with the brain module. In this situation, the brain commands each leg to perform a step at the appropriate time. This division of control has the advantage of requiring a small amount of hardware to implement, as the hardware of the brain is, in some sense, forced to do double duty. A corresponding disadvantage is that the brain is loaded more heavily with repetitive tasks that are likely to require execution fairly often. A second possible solution is to create an intermediary module that provides the gait control. This module receives commands from the brain as to the desired dynamics of movement (i.e. speed, direction, etc.) and responds by issuing the needed commands to each leg module to perform the requested movement. The obvious disadvantage of this solution is the extra hardware that is required. The corresponding advantage in this case, is that of reduced load on the brain module. Both solutions implement the desired behavior, although they have different sets of advantages and disadvantages resulting from the choice of the level of abstraction seen by the brain module. The solution most appropriate for any given situation will be chosen as the circumstances merit. The key point of our master/slave division however, is that there is a division between high-level and low-level control; the level at which that division is placed does not change the overall conception.

2.2 Proposed Implementation

2.2.1 Physical

The master is seated on a main body and is attached to a set of busses, communications and power (described later), that run the length of the body. Electrically, the modules attach to both busses on the robot, and mechanically they attach in a manner appropriate to their function. By way of these connections, a modular robot that is composed of physically and logically separate modules is able to function as a single robot.

2.2.2 Logical

Just as each slave module will have some form of predetermined physical connector so that it can be added into the robot physically, each slave module will also have a set of software routines that make interaction with the brain module possible. Through this interaction with the brain module, the slave modules are added logically into the overall behavior of the robot. The software libraries which make this brain/slave interaction and communication possible will reside in or be added dynamically to the brain's program code so that they are available when it is necessary to communicate with any given slave module. These libraries provide interface functions that are represented in terms of concepts the brain has been programmed to understand, such as distance or locomotion. Through these interface routines, the brain can communicate with the slave modules in terms of predetermined concepts that can be used in the brain's control software. This is directly analogous to the way that physical connectors on the robot's body can provide support as needed without specific knowledge during their design of the type of slave module that is to be connected at the time of operation.

3. COMMUNICATION

In order for the robot to function, the master and the slaves need to communicate with each other. The following section covers the software protocol and the hardware designed to implement the communication is covered.

3.1 High Level Overview

Communication between the master and the modules is achieved by using a communications bus that is run throughout the robot and over which data is transmitted serially. All modules connected to the system have a network address and are contacted over the communications bus by the use of that address. The software in the master maintains a table connecting representations of the physical port location, the network address, and the type of module connected at that point. In this way, the brain knows what type of module is connected to each port. By using the software components of the module, which are part of the master's software, the brain also knows what the capabilities of that module are.

3.1.1 Communication Bus Access Control

All communications over the bus are initiated by the master and take the general form of a message from the master followed by a response from the module. Each communication over the bus contains several fields that specify the module with which the brain is communicating, the type of communication taking place, and, if necessary, some data. Using the communication system, the brain can maintain contact with the modules and perform such tasks as collecting processed or partially processed data from the sensors and controlling motors. As the modules become more advanced, the level of abstraction of the master becomes greater, but the communication remains the same (only the content of the communication changes). For a module that needs to request immediate attention from the master, there is an interrupt line in the communication bus. The module requesting attention will assert that line, and the master will poll all modules to service the one that triggered the interrupt. In this way, time-sensitive messages can be sent to the master in what amounts to a module-initiated transmission without the difficulties involved in having a multiple access protocol on the communication bus signal lines.

3.1.2 Terminology

The following terminology is used in describing the system:

Master - The microcontroller that is the "brain" or coordinator of the robot.

Slave or Module - The self-contained units that are attached to the robot to allow it to perform functions and that communicate with and are controlled by the master.

Communications Bus - The group of wires, or bus, that runs throughout the robot and makes communications between the brain and the modules possible.

Slave Send - The name of the line in the communications bus on which the modules send data and the master receives it.

Slave Receive - The name of the line in the communications bus on which the modules receive data and the master sends it.

Port - The physical connector on the robot body to which modules are connected.

The bus can be visualized as shown in Figure 2.

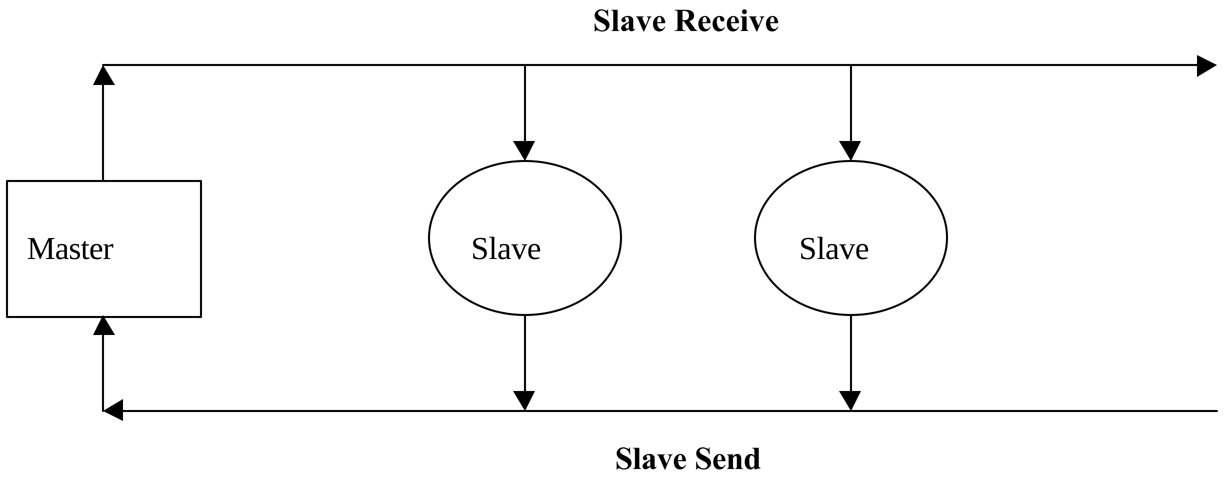

Figure 2: Master/Slave

3.2 Physical Description of the Bus

The communication bus contains five wires:

VDD - Power line to the module microcontrollers.

GND - Signal Ground.

SS - Slave Send.

SR - Slave Receive.

IRQ – Interrupt Request Line.

A power bus runs separately from the communications bus in order to prevent interference if a module that draws a large or rapidly changing amount of current. Two lines running different voltages will be run to account for the different requirements of different modules. The wires are as follows:

V1 - Supply at a certain voltage.

V2 - Supply at a different voltage.

GND - Power Ground.

3.2.1 Single Module Abstraction

The Slave Send line is pulled high through pull-up resistors, and all of the devices connected to this bus have open drain outputs. This means that a logic high output will result in a high-Z state at its output and a logic low output will result in a logic low state at its output. In this manner, each module is able to treat the communications bus essentially as if it were a dedicated serial line. The initiation of communication by the master handles the access control, so that the dedicated serial line abstraction does not have to be modified. The open drain configuration allows the modules to act as if they are disconnected from the output lines of the communication bus unless they are currently using them. In the case of the IRQ line, assertion by multiple modules will not cause any problems because there will merely be multiple outputs pulling the line low. At each port, these lines will be passed through to the module that is to be connected at that port. The net effect, therefore, is that each module is connected to a common communication bus that acts as a serial line, allowing communication between the master and any one module as if that module were the only device connected to the brain over that serial line. In addition, the IRQ line provides the ability for modules to initiate transmissions (and will be discussed in more detail later).

3.3 Addressing

Each module is addressable through the communication bus and therefore is given a network address. This address allows the master to specify to whom a communication is targeted, and it allows the modules to discern for whom a communication is intended. An address on this network is 8 bits long, which provides 256 possible addresses in this scheme. Therefore, the master can communicate with any one of 256 modules on the robot as if it were the only module connected to the brain.

3.3.1 Module mapping

Each physical port on the robot is associated with a network address, which the module will assume when it is connected to that port. In this way, the brain is able to associate a network address with a physical location, and communications with the modules can then take physical position into account as well. Our initial design will have dip switches on the module body through which we can configure the module's network address, or else the addresses will be programmed directly into the module memory. This is merely to make prototyping go as quickly and easily as possible. In the more advanced stages of this project, some circuitry with the port on the robot body will configure the module to have the proper network address when it is connected to the body through that port.

3.4 Protocol

3.4.1 Low Level

At the low level, the network operates as a shared serial bus, with transmit, receive, and ground lines. Our use of this serial bus approximates the way a serial communications line would be used if only two devices were connected to it. Since the idle state of a transmission is high, the SR and SS lines are pulled high with pull-up resistors and therefore they appear as normal serial lines to a receiver. The outputs connected to this serial bus are open drain, and therefore a logic high output is actually put on the line as a high-Z output. The logic high outputs from the transmitters, which is the idle state of a serial transmitter, will have no effect on the line because of this arrangement. Only when a transmitter takes the line low (for the start bit and all subsequent logic 0's to be transmitted) will the output have any effect on the serial bus. Furthermore, the high-level communications protocol will insure that no more than one device will be transmitting on a serial line at any one time.

In summary, in terms of the low level protocol, each module can behave as if it is the only module that is connected to the master via the serial bus. Low-level communication, therefore, is simply implemented as serial communication for the modules and the master.

3.4.2 High Level

The high-level protocol must insure that the serial communication can function properly under the simplification that each module behaves as if it is the only module on the serial bus. The high-level protocol does this by allowing the master to select a module with which it wants to communicate and allowing the master and module to control the serial line while two frames are transmitted – one to the module and one in response back from the module.

3.4.2.1 Frame Contents

All transmissions over this serial bus consist of frames, which are simply groups of bytes. The bytes making up a frame are as follows:

1. Module Address - The address of the module with which the brain is communicating in this particular exchange of frames.

2. Data Type – The type of data the current frame is carrying. This explains to the receiver how many bytes of data to read in and how to interpret that data.

3. Data – The content of the transmission.

3.4.2.2 Frame Types

As stated above, the Data Type field denotes the type of frame that is being transmitted. Four types of frames currently are defined:

1. ID Query – The master must scan through all possible ports (which correspond to network addresses) on which a module could be connected so that it can form a table in memory connecting network addresses and physical locations to module types. When the master is building this table (presumably when it first starts up, although it could conceivably send ID Queries at any time), it will send an ID Query to every possible network address. The ports that have modules connected will send a return frame containing some predetermined representation of the module's type. From these responses, the master's code will build a table in memory. Software libraries associated with each module will also be a part of the master's code, and so the representation of the module's type will actually allow the master to get representations of the capabilities and requirements of the module. In this way, the master can make connections between the physical location on the robot's body and the capabilities of the module connected to that point.

2. Polling for Data – The master may have the ability to request certain predefined types of data from a module. For instance, a sensor module may be able to report back to the brain certain types of data it has retrieved. The master uses this type of frame to request a response containing certain data from a module.

3. Control Message – Some modules may be able to execute commands from the master. For instance, a servo controller will take data representing servo positions from the master and will act upon that data. The master uses this type of frame to send commands to a module for execution.

4. Scanning for Interrupts – The communications line contains an IRQ line that allows the modules to trigger an interrupt. The master responds to an interrupt request by scanning all of the modules, in order of priority, until it finds the first module that has triggered an interrupt. When acknowledging an interrupt request, a module will return to the master the type of information that is urgent and required the interrupt. The master will then poll the module for that data and act upon it accordingly. This method allows us to have interrupts for time-critical messages on the system, without destroying the system by which the master controls access to the communications bus. The system by which we determine the module that interrupted may not be quick enough to work in practice, but it will at least work, and it can be modified later, if need be.

3.4.2.3 Frame Data

The data section of the Frame depends first upon which type of frame is being sent and then on whether it is the original message from the master or the response from the module.

1. ID Query - From brain: Empty (0 bytes). From the module: A representation of the module's type (1 byte).

2. Polling for Data - From brain: Type of data requested (1 byte). From the module: The requested data (size varies with the type of data requested).

3. Control Message - From brain: Command data (size varies with the type of module being commanded). From the module: At least a confirmation of the command's proper reception and execution, and possibly some feedback information (size varies with the type of module).

4. Scanning for Interrupts - From the brain: Empty (0 bytes). From the module: Positive or negative acknowledgment and, if positive, the type of data for which the master should poll the module (2 bytes).

3.5 Network Robustness

3.5.1 Timeout

If one of the modules becomes temporarily unable to respond to the master, then a timeout condition will result. Every message sent from the master requires a result from the module to which it was sent. If the master does not receive a response within a certain time period, then it will set some sort of flags indicating that the module had a timeout, and will then either wait for a short time and retry the transmission, or else go on about its business. The response will be determined by the software on the master.

3.5.2 Module Inoperative

If the master has more than some predetermined number of timeouts in a row for a given module, then it will consider that module to be effectively off the network. The master software will then reconfigure itself as if that module had not been on the robot when it first started and determined its configuration.

3.6 Hardware

In order to implement the communication as described in the preceding sections, certain hardware is needed. To implement the bus certain cables must transfer data and power. Appropriate choices must be made observing current limits and other factors such as shielding. Also, the master and modules both need intelligence to communicate with each other over the bus. For this, microcontrollers are used. Microcontrollers are often called single-chip microcomputers because they are microprocessors with I/O capabilities as well as other features such as a clock. Hardware choices made are discussed in the next section.

4. EXPERIMENTATION/TESTING AND RESULTS

4.1 Hardware

4.1.1 BASIC Stamp II

The BASIC Stamp II (BS2) microcontroller from Parallax Inc. is used for control of some of the modules as well as a temporary master. The BASIC Stamp is a 24 pin IC that has 15 I/O pins. It contains 2kb of EEPROM memory and 32 bytes of RAM. The programming for the Stamp is done in Parallax's proprietary PBASIC language using the BASIC Stamp II editor. The 2kb EEPROM is for program and data storage and the 32 bytes of RAM are for runtime variables and I/O pin access. This microcontroller was chosen because of its ease of use, its history, and its reliable serial interface. The Stamp is relatively easy to program, and to debug. The Stamp also has a history of being used faithfully in many robot projects at Penn as well as around the country. Also, it's serial communication functions work quite well. These factors as well as the Stamp's capabilities made it a good choice for the module controllers. Its main drawback is that it is a bit slow; but it has proven to be fast enough so far to perform its function. It has even been used as a temporary master until a more suitable microcontroller is chosen. More information on the BS2 can be found on the Parallax Inc. website [3].

4.1.2 PIC

A PIC microcontroller – the SX28AC50, which is made by Scenix – was chosen to control some of the modules. It operates at a high speed (50 MHz) and is affordable; about $7 per chip. The PIC provides one on-chip clock, the "Real Time Clock/Counter" or "RTCC", which is incremented with each clock cycle (in the mode of operation that we chose to use). The model we used has three input/output ports, two with 8 pins and one with 4. More detailed specifications can be found on the data sheet for the PIC, which is available on the scenix website [4].

The PIC lacks many of the additional functions that something like a Motorola 68HC11 has on chip, but it provides the basic functionality required from a microcontroller and, especially given its speed, is quite suitable to act as the controller of a module in our modular robotics architecture. The single RTCC is a detriment in control and communications applications, so we implemented eight virtual clocks in software that are based on the hardware RTCC. A variety of clocks are provided by this code, which give us 16- and 8-bit clocks, with and without prescalars to slow the clock. The interrupt, which is triggered by a rollover of the RTCC, is used to update these clocks at regular intervals.

4.1.3 Bus

Initially, both the power and data buses were tested using standard 22AWG hookup wire with success. Later, 28 AWG ribbon cable was chosen for the data bus. So far, none of the modules being used draws a great deal of current, so interference has not been a large problem and a separate line for the power bus has not yet been chosen.

4.2 Modules

Robots operate by input and output. Input comes from sensors and output is realized through actuators. Alongside the development of the architecture, colleagues have been developing the mechanical aspects of the robot, the major development of which is a leg module for a walking modular robot. We have been developing sensor modules as well as the control electronics for the leg module.

4.2.1 Actuators

The actuator module that has been developed by a colleague is a three degree of freedom leg. The fact that it has degrees of freedom means that a single leg has three motors on it. The control electronics for this module consist of a BS2 microcontroller, and a FerretTronics FT639 servo motor controller as shown in the circuit diagram. The RC servo motors that are used in the leg are position controlled motors that go to certain position when that corresponds to the signal sent to them. The signal is a certain pulse width that must be continually sent to the motor. Because the Stamp cannot do this and communicate at the same time, the FT639 continually sends the pulse train just by receiving positional information from the Stamp. Located in figure 4 is a picture of the leg module and the control electronics next to it.

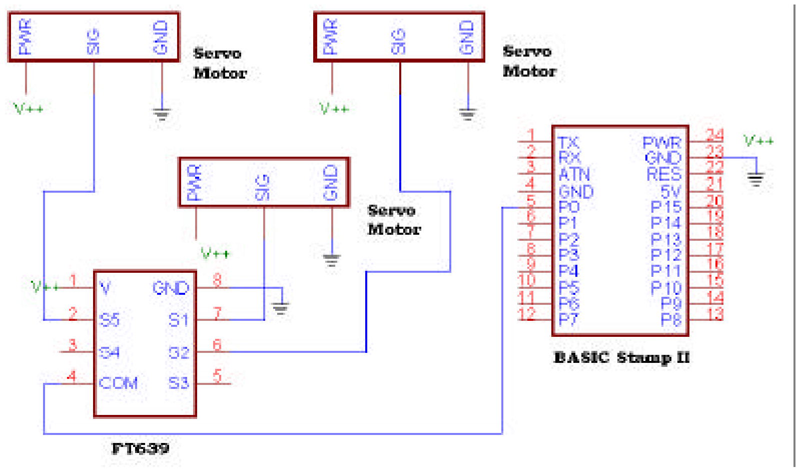

Figure 3: Circuit Schematic of the Leg Module

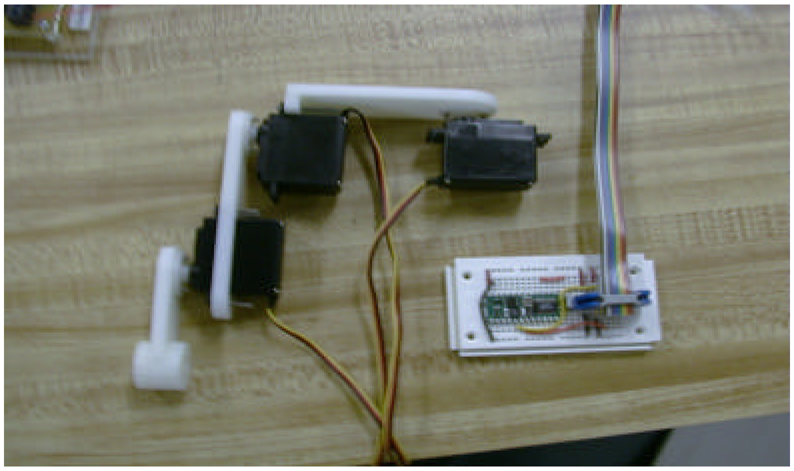

Figure 4: Picture of leg module and control electronics

4.2.2 Sensors

There are a few modules presently in development for used as modules that would contribute to the robot. The only module that is fully constructed and that has been tested in the architecture is an infrared distance measuring device called a PSD (Position Sensitive Detector). The PSD used is the Sharp GP2D12 [5]. This device has an analog voltage output. An 8 bit Analog/Digital (National Semiconductor ADC0831) converter is used to convert the PSD's analog output to a digital output. Figure 5 shows the circuit schematic of the PSD module.The other sensor modules in development include near infrared sensors for obstacle detection and sonar for distance measurement. A picture of the PSD module is included in Figure 6.

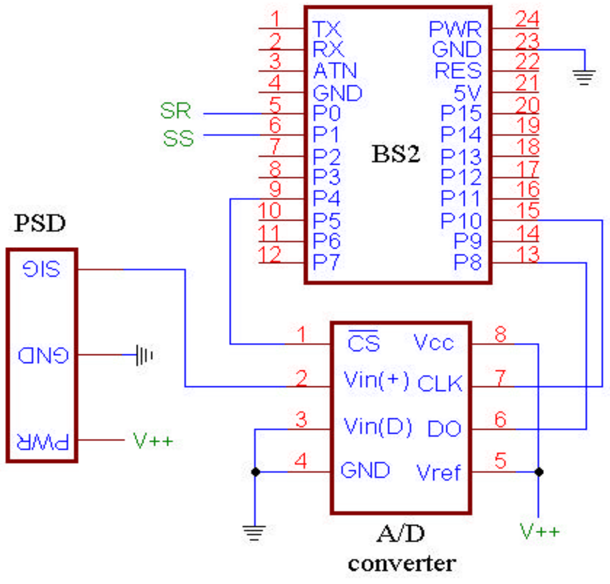

Figure 5: Circuit Schematic of the PSD module

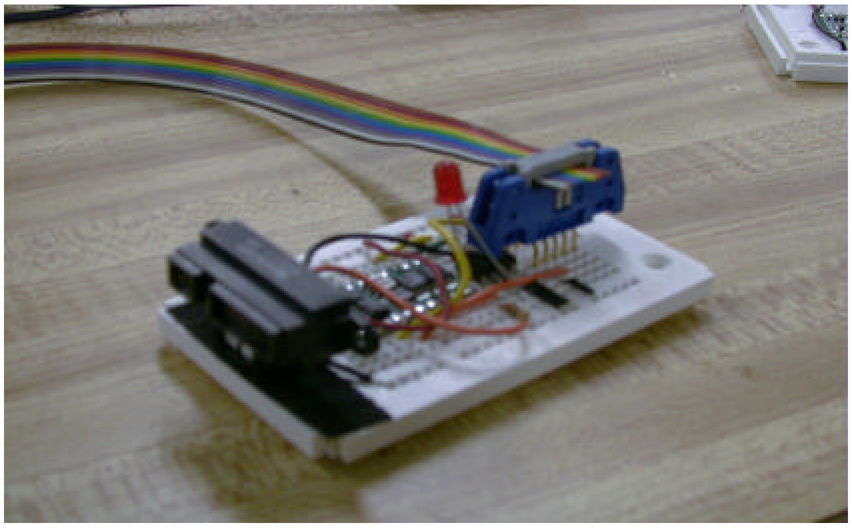

Figure 6: Picture of PSD module.

4.3 Implementation

4.3.1 BASIC Stamp II

The work with the BS2 entailed serial communication in the master/slave scheme and then implementation with the BS2 controlling modules and as temporary master. The first test of networking any microcontrollers was with three BS2's with one acting as master and two acting as slaves. This worked very well. The BS2 has serial communication functions written for it that worked very well. Further implementation followed with the BS2's controlling individual modules. Both the leg and the PSD had to made to function individually before they could be attempted to be networked. Other work done with the Stamp was analysis of its serial transmission so that the architecture could be made open – that is, any microcontroller, such as the PIC, could control a module.

4.3.2 PIC

The work done during the summer dealt primarily with serial transmission and reception by the PIC microcontrollers. This serial communication was necessary for the modules controlled by PICs to communicate with the rest of the robot as defined by our communication protocol. The software clocks on the PIC, as described in the section detailing the hardware of the PIC, made the implementation of serial code much easier. All of the code for the PIC, except that in the interrupt service routine, is grouped into blocks within a main loop. Flags set by some part of the program or other characteristics pertaining to the PIC's current state of operation are used to determine if a given block of code will be executed when the chain of execution arrives at the start of a new code block. Both serial transmission and reception are implemented in such a way. The serial code works by checking one of the software clocks for the expiration of one bit time each time that block of code comes up for execution. When one bit time has expired, a data bit is either shifted in or out, depending on the direction of serial communication. In addition, several code blocks also exist to perform the other necessary functions to make serial reception and transmission possible. The serial communication takes a very small fraction of the available processing time on the microcontroller because the code blocks pertaining to serial communication are only executed once per bit time. The time for one bit at 9600 baud is about 104 uSec and the time to execute one instruction on a 50 MHz processor is about 0.02 uSec. The orders or magnitude between these two times allows for the PIC to perform an enormous amount of useful computation and control even with the requirements placed upon it by serial communication. At this point, the PIC has the code for serial communication via the communications bus and virtual timers in software. This leaves the PIC ready to be used in the control of modules to be used with a modular robot.

4.3.3 Overall

The slaves have open drain outputs on the slave send line and the line is pulled up with a 4.7 kΩ resistor. So far, all communication on the bus with the BS2 is being done at 2400 baud. Currently, the PIC has not yet been tested in the architecture - all tests have been completed using the BS2.

The first test conducted was the networking of three BS2's in a master/slave configuration [6]. The purpose of this test was just to establish that the communication would work; the end effect was of the slaves lighting LEDs. But the addressed slaves successfully received data, took the appropriate action, and returned data to the master. The master also successfully received this data.

After this test was successfully completed, work was done with the individual modules to make them function individually and then individually on the network. After the modules were made to function individually, master code was written using the protocol that we outlined. Using the BS2's, the PSD and the leg module were both successfully controlled when they were, physically, the only modules attached to network. The control of the PSD module, consists of polling that module for data. The PSD itself is always transmitting and analog voltage to the A/D converter which in turn sends the digital conversion to the BS2. The master can get that value from the Stamp. The leg module is being controlled by the master sending a single joint angle at a time. The master sends a joint angle, receives a response, sends the next joint angle, receives a response, and so on. The code for both the PSD and leg module can be found in Appendices B and C respectively.

Finally, three modules were attached to the network, two PSDs and a leg module. Here, a Stamp acted as the master and Stamps were also controlling each of the slaves. For now, the addresses are hard coded on slaves. The master's code had subroutines for each of the slaves and the main body of the code consisted of giving directions. For the PSD, the master polled for distance data. For the leg, the master sends three positional commands to move all three motors to certain positions because the kinematics for the leg have not yet been determined. After many trials, the architecture did work taking data from the PSD sensors and sending control output to the leg. The code for the master controlling this configuration can be found in Appendix A.

5. CONCLUSIONS

Currently, only preliminary tests have been conducted on the architecture. As described in the implementation section, the first tests consisted of networking BS2 microcontrollers. These tests did not use the outlined protocol, but they demonstrated the feasibility of communication between microcontrollers, and of using the BASIC Stamp II microcontroller as both master and slave.

After successfully networking the Stamps and developing the individual modules and final step of networking the modules was taken. Attaching multiple modules to the serial bus and controlling them worked well. Presently, only three modules have been tested with the architecture and the outlined protocol. Most of the problems encountered were either wiring or code errors. There very were virtually no data errors encountered – the master's control of bus access appears to be effective. As the robot becomes more complex, and more modules are added, the likelihood of errors will increase. Error correction is a concern and will be developed further in the future. Another factor is the speed of the network. The current speed, 2400 baud, worked very quickly with three modules. Determining a communication speed that will be able to control the entire robot effectively will have to be determined after the robot is constructed.

Overall, the designed architecture and protocol work very well as they have been tested. But, much more extensive testing needs to be done. The interrupt system devised for the protocol has to be tested yet. The testing so far has only been done with BS2 microcontrollers. Testing needs to be done with different types, such as the PIC, to demonstrate the modular system is open. And, of course, the system needs to tested with more modules of the same and different types.

But, the architecture shows great promise for the control and construction of a modular robot. A robot is currently being constructed that will use the architecture and modules developed for this project. Although only the distance sensors and leg modules are currently under construction, the possibilities for a modular robot are endless. Using this architecture a single robot walk, roll, or even swim using different combinations of sensor and actuator modules.

6. ACKNOWLEDGEMENTS

We would like to thank our advisor, Dr. Jim Ostrowski for all of his support and guidance. We would also like to thank our collaborators, Darnel Degand and Robert Breslawski. The help of Terry Kientz and Sid Deliwala has also been invaluable in completing this project. Finally, I would like to thank Dr. Jan Van der Spiegel, and the National Science Foundation for supporting the REU grant that made this project possible through the SUNFEST program.

7. REFERENCES

1. M. Yim, “Modular Reconfigurable Robots,” February 1999. [http://www.parc.xerox.com/spl/members/yim/modrobots.html]
2. M. Peasegood, J. Pineau, J. Wylie, and J. McPhee, “Hexplorer Automated Walking Vehicle,” October 1998. [http://real.uwaterloo.ca:80/~robot/]
3. Parallax Inc, 1999. [http://www.parallaxinc.com]
4. Scenix Semiconductor Inc, 1999. [http://www.scenix.com/products/datasheets/sx_datasheet.PDF].
5. Sharp Microelectronics of the Americas, 1999. [http://www.sharp.co.jp/ecg/NewProducts/sensor/gp2d12.pdf]
6. C. Kühnel and K Zahnert, *BASIC Stamp,* Newnes, Boston, 1997, pp. 126-131.

APPENDICES

The following appendices contain the source code for the experiments. It is all written for the BASIC Stamp II microcontroller in PBASIC. An apostrophe denotes a comment.

Appendix A: Master Code

```
' --- Kapil Kedia ---

'----Title----> Master

' Master.BS2
' Program for the Master in a BS2 network.
' Master controls one leg module and two PSD modules

'---Constants----

REC con 0       ' Pin 0 is the master recieve line
SND con 1       ' Pin 1 is the master send line

baud con 396 ' 2400, 8 bit no parity, true data, driven

'----Variables---

servo var byte          ' variable to store servo motor number
servoPos var byte               ' variable to store servo motor position

i var byte
address var byte                ' variable to store module address
j var byte

'-----Main----

' Poll PSD #1 for data

        address="1"
        gosub PSD
        pause 1000

' Control leg

        address=$FF
        gosub homer                     ' appropriate movement subroutine
        debug "Done",cr, cr
        pause 50

 Poll PSD #2 for data

        address="2"
        gosub PSD
        pause 1000

stop
'*****************************
'**------- Subroutines -----***
'*****************************

'PSD
'*****Take data from Psd subroutine
'This subroutine sends a request to the PSD modules for data and then recieves it

PSD:

'--- constants ----

datatype con "P"
distance con "d"

'--- variables -----

dist var byte
type var byte

'----Main--------

serout SND, baud, [address, datatype, distance]           ' send data request

serin REC, baud,[WAIT(address), type, dist]         ' recieve requested
                                                     ' data
'debug hex? address                                   ' debug the data
'debug ? type                                         ' recieved by the
'debug DEC? dist,cr,">"                               ' master


if type="s" then record              ' Checks to see if the data type is correct

serout SND, baud, [address, datatype, distance]       ' If so, the program
' proceeds, if not it will
serin REC, baud,[WAIT(address), type, dist]' attempt to poll the module
                                           ' again. If this is not
```

```
                    ' successful, the program will                                                              '
continue
  'debug hex? address
  'debug ? type
  'debug DEC? dist,cr,">"

record:
return


'Leg
'*****3 DOF Leg control subroutine

'This subroutine sends control data to a 3 degree of freedom leg one joint angle at a time

Leg:


'----- constants -----
dat con "C"

'-----variables -----

typ var byte
number var byte
position var byte

'----Main--------


'Send servo # and position to leg module
 serout SND, baud, [address, dat, servo, servoPos]

'Recieve feedback information
 serin REC, baud, [WAIT(address), typ, number, position]

debug hex? address              'debug information recieved by master   debug ASC? typ
debug ? number
debug ? position,cr,">"

pause 100

IF typ="f" THEN go              'Check if data type is correct
                                         'If so, the program will proceed
serout SND, baud, [address, dat, servo, servoPos]
```

```
'If not, the program will send control data again. If this is not successful, the program will
proceed.
serin REC, baud, [WAIT(address), typ, number, position]

debug hex? Address
debug ASC? typ
debug ? number
debug ? position,cr,">"

go:
return


'**********************************
''''''''''''''''''''''''''
''' Movement subroutines  '''
''''''''''''''''''''''''''
' These are several movement subroutine designed to test the movement of a single 3 dof leg



'''''''''''''
'  kick  '
'''''''''''''
kick:

v var byte

gosub homer
pause 200

'leg back
servo=2
servoPos=130
gosub Leg
pause 100

servo=1
servoPos=75
gosub Leg
pause 100
servo=0
servoPos=40
gosub Leg
pause 100
```

```
'leg forward
servo=2
servoPos=152
gosub Leg
pause 100
servo=1
for i=80 to 200 step 20
        servoPos=i
        gosub Leg
        pause 200
next
pause 100
servo=0
for i=50 to 230 step 20
        servoPos=i
        gosub Leg
        pause 200
next
pause 1000

goto kick

return

'----------------
''''''''''
' stepping '
''''''''''

stepping:

gosub homer
pause 400

for i=0 to 255
        servo=0
        servoPos=i
        gosub Leg
        pause 400
next
pause 200
servo=0
servoPos=152
gosub Leg
pause 200
for i=0 to 255
```

```
        servo=1
        servoPos=i
        gosub Leg
next
pause 200
servo=1
servoPos=152
gosub Leg
pause 200

for i=75 to 220
        servo=2
        servoPos=i
        gosub Leg
next
pause 200
gosub homer

return

'----------------------
''''''''''''''''
''''dance''''''
''''''''''''''''

dance:

gosub homer

pause 1000
servo=0
servoPos=255
gosub Leg
pause 1000

servo=1
for i=155 to 255 step 50
        servoPos=i
        gosub Leg
        pause 100
next
servo=2
for j=152 to 182 step 10
        servoPos=j
        gosub Leg
        pause 100
```

```
next

for i=1 to 3

servo=0
servoPos=152
gosub Leg
pause 100
servo=2
servoPos=152
gosub Leg
pause 400
servoPos=122
gosub Leg
pause 40
servo=0
servoPos=255
gosub Leg
pause 40
servo=1
servoPos=0
gosub Leg
pause 300

servo=2
servoPos=182
gosub Leg
pause 200
servo=0
servoPos=0
gosub Leg
pause 40
servo=1
servoPos=255
gosub Leg
pause 300

next

gosub homer

return


'------------------------------
''''''''''''''''''''''''''''''''''
```

```
'side -- side to side"
''''''''''''''''''''
side:

gone:
servo=0
servoPos = 152
gosub Leg
servo=1
servoPos = 152
gosub Leg

servo=2
servoPos=122
gosub Leg
pause 1000

servo=2
servoPos=152
gosub Leg
pause 100
servo=2
servoPos=182
gosub Leg
pause 1000

goto gone

return


'------------------------------------------------------------
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'sweep -- A leg sweep back and forth -- just motors 1 and 2
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

sweep:
servo=2
servoPos = 152
gosub Leg


repeat:

servo=1
servoPos=152
```

```
gosub Leg
servo=0
servoPos=152
gosub Leg

pause 100

servo=1
servoPos=225
gosub Leg
servo=0
servoPos=75
gosub Leg

pause 1000

servo=1
servoPos=152
gosub Leg
servo=0
servoPos=152
gosub Leg

pause 100

servo=1
servoPos=75
gosub Leg
servo=0
servoPos=215
gosub Leg

pause 1000

goto repeat

return


'**The home position -- all servos in the neutral position

homer:
for i=0 to 2
            servo=i
            servoPos = 152
            gosub Leg
```

```
          pause 100
          next
return
```

Appendix B: PSD Module Code

```
'--------Kapil Kedia--------

'---------Title--------> PSD Slave

'Inerfacing the Sharp PSD with the National Semiconductor
'ADC0831 Analog/Digital converter to the stamp
'and using it as a slave module in a BS2 network

'---------Constants---------'

CS con 4     ' Chip select is pin 8.
AData con 8    ' ADC data output is pin 9.
CLK con 10    ' Clock is pin 10.

address con "1"           ' Address of module
error con 100   ' Error constant
sensor con "s" ' dataype - sensor
REC con 0                 ' recieve line P0
SND con 1                 ' send line P1

baud_in con 396                    ' 2400, 8 bit no parity, true data, driven
baud_out con 396+$8000  ' 2400, 8 bit no parity, true data, open drain




'--------Variables---------'

ADres var byte                    ' A-to-D result: one byte.
datatype var byte
request var byte
distace var byte

high CS          ' Deselect ADC to start.

'-----Main----

start:

'recieve data request from master
 serin REC, baud_in, [WAIT(address), datatype, request]
'debug "Frame:"                              'debug recieved data
'debug hex? address
'debug ASC? datatype
```

```
'debug ASC? request,cr,">"

if datatype = "P" and request="d" then poll   'If the datatype is
serout SND, baud_out, [address, error]           'correct, the slave will
                                                 'poll the PSD,else send an
                                          'error condition back to
                                                        'the master
poll:

  low CS                 ' Activate the ADC0831.
        shiftin AData,CLK,msbpost,[ADres\9]        ' Shift in the data.
        high CS              ' Deactivate '0831.
        debug ? ADres        ' Show us the conversion result.

serout SND, baud_out, [address, sensor, ADres]      ' Send distance data
                                                    ' to master

  pause 1000                  ' Wait a second.


goto start                                'Go to beginning and wait for data
                                                 'request from the master
```

Appendix C: Leg Module Code

```
' --- Kapil Kedia --- '

'----Title----> Leg Slave

'LegSlave.BS2

' Slave control code for direct control of a 3 DOF leg module
' Program for a Slave in a BS2 network.


''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Start of Declarations for the FT639 Controller
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' These are the constants for the FT639 servo controller
SACTIVE     con 117
SSETUP     con 122
SSHORT     con 85
SLONG      con 90
SHEADER     con 96

' These are the variables for the FT639 servo controller
' The Subroutines used require 3 temporary variables
' These variables can be change to a specific byte
' All the bytes used for these variables may be reused
'  in other part of the program.
Servo       var   byte 'Can be changed to a specific byte
ServoPos    var   byte 'Can be changed to a specific byte
ServoTmp    var   byte 'Can be changed to a specific byte
ServoHeader   var   ServoPos 'This is the same byte as ServoPos

' This is the pin that the FT639 is connected to. The output pin
'  0 may be changed to any of the other output pins.
FT639Pin    con   10

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' End of Declarations for the FT639 Controller
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''



'---Constants----

REC con 0                               'Receive line set to P0
SND con 1                               'Send line set to P1
```

```
baud_in con 396                    ' 2400, 8 bit no parity, true data, driven
baud_out con 396+$8000      ' 2400, 8 bit no parity, true data, open drain

address con $FF                    ' address of leg module

error con 101               ' error condition constant

feedback con "f"                   ' datatype

'----Variables---

datatype var byte
position var byte
number var byte
x var byte

'----Initialiazation----

'----FT639 initialization----

gosub  servoSetup      'This puts the FT639 in setup mode

gosub  setLong        'This sets the pulse length to long

'servoHeader =2        'This sets the header to 2. Need to reset
'gosub  setHeader       ' after changing pulse length.

gosub  servoActive      'This puts the FT639 in active mode


'-----Main----

start:

   'recieve control data from master
          serin REC, baud_in, [WAIT(address), datatype, number, position]

   'debug "Frame:"                          'debug data recieved from master
   'debug hex? address
   'debug ASC? datatype
   'debug ? number
   'debug ? position,cr,">"
```

```
        IF datatype = "C" THEN move                'If the datatype is correct, the  serout
SND, baud_out, [address, error]        'program proceeds else
                                                              'it sends the error condition
                                                     'back to the master

move:                          ' The module uses the servo number and
                               ' position informaiton to send a certain
Servo=number                   ' motor on the leg to a certain joint
  ServoPos=position ' angle through the FT639 subroutine moveServo
  gosub moveServo

 'send feedback information to the master
  serout SND, baud_out, [address, feedback, number, position]
  pause 100

goto start                     ' go to the beginning of the program and
                               ' wait for more control instructions from
                               ' master



''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' The Start of the Subroutines for the FT639
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

' This will set the header value on the FT639
' The header value must be loaded into servoHeader variable before being
'  called.
setHeader:   servoTmp = SHEADER + servoHeader
      serout FT639Pin,16780,[servoTmp]
      return

' This will set the Pulse length to short
setShort:   serout FT639Pin,16780,[SSHORT]
      return

' This will set the Pulse length to long
setLong:   serout FT639Pin,16780,[SLONG]
      return

' This will set the FT639 to setup mode
servoSetup:   serout FT639Pin,16780,[SSETUP]
      return

' This will set the FT639 to active mode
servoActive:  serout FT639Pin,16780,[SACTIVE]
```

```
        return

' This will move a servo to a position
'
' The following variables must be set before calling subroutine
'    servoPos - This is the position of the servo 0-255
'    servo  - This is the servo to control. Start at 0 so
'         servo 1 = 0, ... , servo 5 = 4.

moveServo:   ServoTmp = (ServoPos & %00001111) | (Servo << 4)
        serout FT639Pin,16780,[ServoTmp]

        ServoTmp = ((Servo << 4) | %10000000) | (ServoPos >> 4)
        serout FT639Pin,16780,[ServoTmp]
        return
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' The End of the Subroutines for the FT639
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
```