Encouraging Innovation by Engineering the Learning Curve

by

Christopher Thomas Lyon

Bachelor of Science, Electrical Science and Engineering
Massachusetts Institute of Technology
Cambridge, MA
June, 2002

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in
Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 21, 2003

Author _____

Christopher T. Lyon
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by _____

Bakhtiar J. Mikhak
Thesis Supervisor

Accepted by _____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Encouraging Innovation by Engineering the Learning Curve

by
Christopher Thomas Lyon

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in
Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 9, 2003

Abstract

Never before has it been easier to make powerful, computationally enhanced tools for creative expression and technical innovation available to the masses. We set out to investigate the ways in which a system designed to promote rapid electronics design and prototyping could enable people of diverse technical and social backgrounds to explore powerful ideas through meaningful, hands-on design experiences. When the tools themselves are transparent and easily extensible by anyone, a strong user community naturally begins to develop. Users are encouraged to dig deeper into the layers of complexity that they encounter while following the learning trajectories we have carefully built into the system itself. This thesis focuses on the design, development, and deployment of the Tower system, a powerful, flexible, and extensible electronics design environment, as well as the exploration of its applications in the fields of development, education, and industrial prototyping.

Thesis Supervisor: Bakhtiar J. Mikhak
Title: Research Scientist, MIT Media Laboratory

Table of Contents

Acknowledgements

This thesis represent the culmination of thousands upon thousands of hours of work by myself and others. The work discussed herein would not have been possible without the dedicated efforts of a number of individuals.

I would like to thank...

- Bakhtiar Mikhak, for his dedication and guidance thoughout my five years at MIT. He has served as an amazing source of inspiration and encouragement, and I consider myself fortunate to have had the priviledge of working with him.
- Tim Gorton, for implementing key portions of the Tower system design, and working tirelessly to help build our global userbase.
- Margarita Dekoli, Andrew Sempere, Sara Cinnamon, and Daniel Bryan, for providing some of the first and most extensive applications for the system, and for never losing faith in the Tower throughout its development process.
- Ali Mashtizadeh, Benjamin Walker, Larisa Egloff, Jeremy Walker, Austin McNurlen, Glenn Tournier, Vanessa Hsu, Lele Yu, Martin Kang, and Amir Mikhak, for helping to continue the development of the system.
- Brian Silverman, Robbie Berg, and Fred Martin, for providing an exceptional body of work upon which much of my own has been built.
- Mitchel Resnick, for his overall support throughout my entire time at the MIT Media Lab.
- The educators, students, developers, and designers around the world who are actively using the Tower system, and are taking upon themselves the task of its continued extension and development.
- Dawn Wendell, for her patience during the entire thesis-writing process, and for helping to turn my words into something more closely resembling the English language.
- My parents, for their continued support and always trying their best to understand what I'm talking about whenever I try to explain this stuff to them.
- Scooby Doo and the gang, because I couldn't have gotten away with it if it weren't for those meddling kids and that darn dog.

List of Figures

List of Tables

Reading this Thesis

This document is divided into seven chapters:

Chapter 1 - Overview

This chapter presents an introduction to our research, and outlines the guiding principles which have directed our work towards encouraging innovation in developmental, educational, and industrial settings.

Chapter 2 - Motivations

This chapters details our motivations in designing a modular computational construction kit, and discusses the design of the Tower system to meet the needs of an extensive user community.

Chapter 3 - Scenario of Use

This chapter tells a story of how the Tower system could be used in a rural educational setting, to provide students with valuable experiences in electronics, design, and prototyping.

Chapter 4 - Suite of Applications

This chapter examines a broad cross section of projects created using the Tower system, and explores the ways in which the system meets the needs outlined previously in this document.

Chapter 5 - Case Studies

This chapter discusses the use of the Tower system within a global user community, the ways in which the system has met the needs of those using, and how its inherent extensibility has enabled continued development by those using it.

Chapter 6 - Technical Detail

This chapter explains the technical details of the Tower system architecture on hardware, firmware, and software levels, and explains the process by which the Tower system has evolved to its current state.

Chapter 7 - Reflections

This chapter reflects upon the lessons learned and goals achieved through the design, development, and deployment of the Tower system to users around the world, and outlines paths for continued development and extension.

There are twenty-six appendices, which fall into three main categories:

Appendices A → D

These appendices discuss the basic set up and operating of the Tower system, and provide detailed language references for PIC Logo, Rabbit Logo, and PIC assembly.

Appendices E → T

These appendices present detailed technical information and examples of use for every module currently supported in the official release of the Tower system

Appendices U → Z

These appendices contain the application code developed for the six projects outlined in Chapter 4 of this document.

The layout of this document is designed to provide uninterrupted text flow whenever possible, by locating footnotes and most of the images to the side of the main text. Symbols are used to correlate notes on a page with those in the margins, and when noted, references can be located by their number in the section immediately following Chapter 7. This entire document is meant to be printed double-sided, and read as a book would be.

"If we all did the things we are capable of doing,
we would literally astound ourselves."

Thomas Edison

A young child exploring one of the many tools available for use on his project at a workshop in Dublin, Ireland.

Chapter 1 - Overview

Never before has it been easier to make powerful, computationally enhanced tools for creative expression and technical innovation available to the masses. With the new tools we are developing, anyone now has the ability to create truly amazing things.

Building on our previous work in creating computational construction kits[◦], we set out to explore the ways in which new tools for design and prototyping could enable people of diverse technical and social backgrounds to explore powerful ideas through meaningful, hands-on design experiences, when the tools themselves are transparent and easily extensible by anyone.

While systems to aid in electronics design do exist[□], none of the ones currently available possess the level of end-user transparency needed to meet our goals. In addition to creating a system that is useful for many applications, we wanted to make a system that could be explored and extended by users, providing unique insight into the underlying ideas of system design and problem solving, thereby promoting an environment for encouraging innovation.

◦ The members of the core Tower development team, comprised of Bakhtiar Mikhak, Tim Gorton, and myself, were all active participants in the development of the MIT Media Lab's Cricket system. See reference [29], Metacricket paper, and reference [8], Cricket System website.

□ A survey of existing development platforms can be found in section 2.1.1 of this document.

The goal was simple: to allow those with design ideas to realize them in the form of functional prototypes that could be demonstrated to others. The power of rapid prototyping is that it enables faster iterative design cycles, removing the focus from the technical aspects of implementation and placing it on the ideas themselves. In order to turn a difficult engineering problem into a simple prototyping one, the challenging technical problems need to be taken off of the table until the user is ready for them.

To achieve this goal, the system itself must embody a cleverly engineered architecture conducive to learning. By breaking the system into distinct functional building blocks, users are able to explore its different parts at their own paces, using only what they need at a given time for their application. To make sure that novice and expert users alike will not be discouraged as they work, there must be varying plateaus for success, allowing users to enter the system on their own technical level, yet not feel constrained by the depth of levels remaining ahead of them.

A modular, end-user transparent system architecture ensures that experts don't have to reegineer the wheel for each application, while at the same time, novices are less likely to be intimidated by something overly challenging.

For those new to technology development, the system makes it exceptionally easy to create functional prototypes. When exploring the system on its highest level, novices do not have to use it for a long time before seeing results. Even a student using it for the first time could create a functioning project in just a matter of hours. A working model provides an excellent opportunity for them to iterate through designs, and by doing so familiarize themselves with the design process, figuring out the details of their initial ideas in a low-risk environment.

As applications continue to develop, users can begin peeling off layers of transparency in the system to uncover its true power and more closely adapt it to their specific applications. As things progress even further, motivation and interest grow, and users will experience renewed passion to playfully explore and reflect upon the process of design. Additionally, as new users familiarize themselves with basic building blocks of design, they gain an added understanding of the workings of more complicated applications that they might encounter. The key to encouraging this growth lies in solving specific problems by using tools that have low entry points and high ceilings. When the tools themselves can be easily reconfigured

to meet varying needs, users are encouraged to explore the ways in which a single tool can be used to solve a number of different problems.

Innovation itself is the effective recombination of existing ideas and functional elements. Our goal is to create not only a network of users, but a network of inventors who can build upon their collective knowledge by sharing what they learn with a global userbase, in order to further extend each other's skills and capabilities.

This thesis focuses on the design, development, and deployment of the Tower system, a powerful, flexible, and extensible electronics design environment, as well as the exploration of its applications in the fields of development, education, and industrial prototyping.

University students building electronics into their project at a workshop in Cartago, Costa Rica.

Chapter 2 - Motivations

When creating a computational construction kit, one must first decide what problem it is that needs to be solved. Our goal was to create a system that enables people to design, regardless of their technical background.

In designing the system, we set out to provide maximum benefit to users from a wide range of backgrounds with varying experiences and trajectories. By carefully structuring the system itself as well as the means and environment through which its use is taught, we hoped to create something that would prove useful to an extremely diverse community of users.

The system itself must have multiple and frequent plateaus for success. Ideally, beginners will be able to build functioning prototypes quickly without being intimidated by complex technical details. As their interests grow, the details can be explored in greater depth, revealing only as much complexity as needed at any given point. At the same time, experts need not be tied up in repetitive technical tasks when designing applications. With common system elements in place, there is no need to reinvent the wheel by engineering things like sensing and actuation hardware multiple times over.

More than just a tool, the system should be an example of toolbuilding and transparency. It must not only meet the expectations of those wishing to use it, but at the same time be capable of surpassing our own expectations for it by providing simple entry points for continued extension by a global userbase.

2.1. Need

To determine what is needed in a new system, we are required to observe existing systems and their shortcomings, and evaluate the basic scenarios of use and the ways in which users can benefit from a new system tailored specifically to their needs.

2.1.1. Existing Technologies

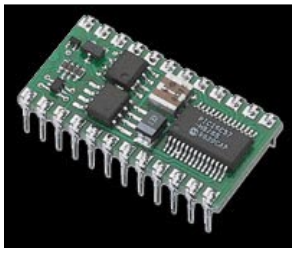


Figure 2.1 - The BASIC Stamp 2, a microcontroller module running a BASIC language interpreter.

In addressing the needs for a computational prototyping system, it is important to evaluate the spectrum of existing systems, and note their strengths and weaknesses.

There are devices available like BASIC Stamps *(Figure 2.1)* from Parallax Inc.[◦], which provide a self-contained computational unit in a small package, but they suffer from a lack of transparency and a closed development network. While they are easy to obtain, as are their add-on modules, users are limited to programming them in BASIC and are faced with steep obstacles if they wish to extend the core functionality of the system. For example, while a module such as a servo-motor driver board can be purchased, problems arise when one needs to interface to new hardware, and is left with nothing but pin level controls and limited ability due to the speed of the virtual machine. New hardware development requires the ability to seamlessly implement and integrate low-level protocols with the underlying system itself, which is lacking in this system.

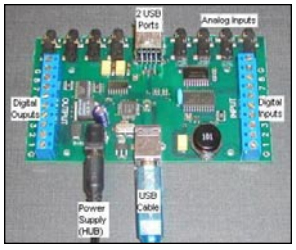


Figure 2.2 - The Phidget™ core interface module. The system itself is a collection of tethered USB I/O devices.

The Phidget™ system *(Figure 2.2)* from Phidgets Inc.[□] is an example of a system that has been widely praised for its seemingly endless possibilities when it comes to prototyping with hardware. However, it has the same transparency flaws as BASIC Stamp, but also falls into the much more serious trap of having no central processing power of its own. The core module and all of its associated devices will not function unless connected to a personal computer. While in some cases the required tether could be an acceptable option, in most it proves to be a significant shortcoming. Teaching about basic computational systems becomes a difficult topic

◦ See reference [35], Parallax corporate website.

□ See reference [18], Phidgets paper, and reference [36], Phidgets corporate website.

when you need to have a thousand-dollar-plus machine connected at one end to handle all of the "hard stuff." In actuality, the Phidget™ System is more of a "computational enhancement," than an actual prototyping system.

Another system, in many ways similar to Phidgets™, is the GoGo Board[◦] *(Figure 2.3)*, developed in the Future of Learning group at the MIT Media Lab. The GoGo Board is essentially a serial interface for providing input-out (I/O) functionality. With eight sensor and three motor ports, the GoGo Board is a useful alternative for those wishing to complement personal computer applications with some degree of real-world interaction, but again falls short in its inability to retain end-user code and operate independently of a host computer.

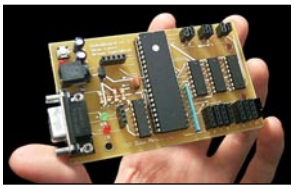


Figure 2.3 - The GoGo Board, a tethered serial I/O interface providing sensor and motor ports.

In the past, we worked extensively on the Cricket[□] system *(Figure 2.4)*, under the direction of Professor Mitchel Resnick and Bakhtiar Mikhak in the Lifelong Kindergarten group at the MIT Media Lab. The Cricket system predates Phidgets™, the GoGo Board, and much of BASIC Stamp development. A Cricket itself is a small, self-contained prototyping system. Its functionality can be extended by creating new "bus devices" that can be connected to the core module. While the system is ideal for many smaller projects, it lacks the processing power needed for higher-end applications. Also, creating new devices for the system requires a knowledge of low-level microcontroller programming and the proprietary protocols used in the system, which can prove quite daunting to individuals trying to simply realize a new application. Since the system is in a state where it successfully meets the needs of its userbase, recent efforts have been focused on the development of new software environments for communicating with it. Although we are not still developing hardware for the Cricket system, we continue to study its use and support the current user community.

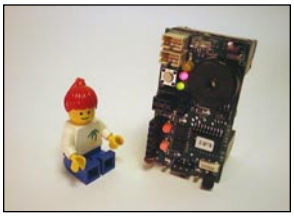


Figure 2.4 - The Classic Cricket, a tiny computer running a Logo interpreter and providing sensor and motor ports as well as a small beeper. Shown next to a LEGO™ figure for scale.

2.1.2. Developing World

For our applications in the developing world, there is an ever-present need to have a robust electronics toolkit that can be easily extended, even without the availability of advanced hardware development technologies and equipment that we often take for granted.

In more isolated communities, it is important for the users to be able to make what they do not have and be capable of extending the functionality of the system without the need for complex printed-circuit-board (PCB)

◦ See reference [15], GoGo Board project website.

□ The members of the core Tower development team, comprised of Bakhtiar Mikhak, Tim Gorton, and myself, were all active participants in the development of the MIT Media Lab's Cricket system. See reference [29], Metacricket paper, and reference [8], Cricket system website.

fabrication. Prototyping hardware must be in place that allows anyone to tap into the core of the system via available prefabricated circuit boards, allowing them to extend the system quickly and effectively.

Furthermore, since these communities will not be closely tied to the knowledge base of our core development circle, it is important to avoid the trap of just giving them technology that they would not be able to understand the inner workings of. When electronics are simply handed to users and not properly explained, there is very little motivation for them to understand and extend what they are given. Instead, we need to teach users how to actually build the technology they need. Selecting the proper layers of abstraction in the system design is an incredible challenge. By creating a system that is technologically transparent on many levels and then working with users to explore the functionality of those levels, powerful lessons will be learned. An open system architecture will encourage users to enhance the capabilities of the system as needed. In addition to encouraging them to modify and extend the tools, a deep understanding of how things work will give them the ability to confidently explain and teach the ideals of our research to others around them, laying new foundations for rural development.

2.1.3. Education

When used in an educational context, the importance of transparency in design is further emphasized. A technologically-tranparent system will give teachers the power to explain how things work and at the same time encourage students to understand the inner workings of a complex system in small, easy-to-comprehend parts[◦].

In an academic setting where resources are often scarce, an easily-reconfigurable system can provide vast opportunities. If the system is completely modular in nature, the same parts could be used for a variety of different projects[□]. The same modules that students would use to build a weather station for measuring temperature, humidity and light values, could also be used to make a robot or a kinetic sculpture.

When a complex system can be broken up into many functionally-distinct pieces, students can be introduced to the new technology piece-by-piece, understanding modules as they become needed in their projects. Eventually, the students will learn how everything in the system works and will have gained a fundamental understanding about how complex systems are

◦ See reference [35], Beyond Black Boxes paper, and reference [38], Pianos Not Stereos paper.

◦ See reference [4], Design Rules by C. Baldwin and K. Clark.

designed and built, without the intimidation associated with facing a large solid block of new technology all at once.

For example, if a student had built a robot using a fully modular electronics system, he would have been introduced to the parts used for reading sensors and driving motors. Now imagine that same student later walking through our laboratory space, past a much more complex project built using the same modular system. Even if he had no previous knowledge of what the project was supposed to do, he could break it apart mentally, observing the different pieces and how they interacted until he began to grasp an understanding of the project at hand. Seeing knobs plugged into a sensor module would make them appear to be the likely input devices. Some modules might have lights and motors on them; those are obviously used to relay information to the user by motion and light. But there might be new modules the student has not seen before: ones that allow independent sets of modules to connect to each other, with lights flashing back and forth on both ends of the wires. The student might guess that those are communication modules, as it looks like they are being used to send data back and forth. He would have guessed correctly, applying the knowledge he gained by exploring the system.

By designing a system that emphasizes the strict design constraint of one-module-per-function, users will be given the basis to guess as to the functionality of new modules when they see them in action. Far more powerful than a system itself will be the knowledge and understanding that users gain from working with it, which will help them along their way to comprehending more complex systems, and even designing new ones.

2.1.4. Industrial Prototyping

More than just for novices, it is inherently important for the system to work well even for professionals and those experienced in technology and design. In addition to being useful for educators and communities in the developing world, the system must also be powerful enough for experienced users to benefit from. Professional engineers should be able to use the system to accent their own projects by rapidly building functioning prototypes, even if only to initially experiment with the eventual intent of later creating proprietary electronics.

An ideal prototyping system will reduce the emphasis on the technological requirements of an application, allowing those with ideas for projects to

◦ See reference [42], Serious Play by M. Schrage, and reference [29], Metacricket paper.

actually build them. As applications mature, layers of transparency in the system can be peeled off, allowing a technological maturity to develop as the project in question is refined. Designing a system to bridge the gap between designers, engineers and managers will pave the way for powerful industrial and economic applications.

2.2. Our System

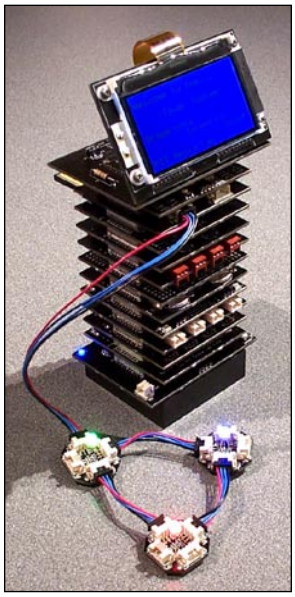


Figure 2.5 - The Tower system, a fully modular computational construction kit that we created to meet the needs set forth in this chapter.

To meet the needs as outlined above, we have developed the Tower◦ system *(Figure 2.5)*, a fully-modular computational construction kit created to simplify the design of embedded hardware systems. The Tower is comprised of a variety of hardware modules, which can be easily interconnected based on the needs of specific applications. More than just a prototyping tool, the Tower exemplifies the ideals of toolbuilding and transparency in a context that allows it to reach a wide audience both domestically and abroad.

2.2.1. Design Rationale

From the beginning, we wanted to emphasize the use of industry-standard protocols throughout the system. A key disadvantage of our previous work has been its lack of standard protocols. As successful as these previous implementations had been, the use of proprietary communication methods not only made it difficult for novice users to extend the system, but also made for a tougher sell when it came to encouraging highly-technical users to accept certain aspects of the implementation. Through our experience, we learned that most technical users are reluctant to use non-standard communication protocols, as it simply adds to the difficulty of new hardware design. By using standard hardware communication protocols from day one, we were able to build in the low-level support needed in order for users to be able to easily communicate with many commonly available devices. In addition, there is a wealth of documentation available regarding the implementation of standard protocols, which provides a solid basis for system extension by allowing users to learn by examples found, and easily understand how and why the extensions work.

While an emphasis on system modularity has been critical since the early stages of development, it initially arose as the result of technical difficulties in debugging early models. With single circuit boards at first containing multiple functional elements, it became easier to break the board into separate interconnected modules, each performing one and only one func-

◦ See reference [48], Tower system website.

tion. With modularity comes versatility: the same system that allows you to build a robot can be easily reconfigured for science experiments and other learning and artistic activities. With proper module division, one tool can serve a vast number of needs.

2.2.2. Technical Overview

Physically, the Tower is comprised of a foundation module containing a core processor, and other layer modules that stack on top of it *(Figure 2.6)*. The layers provide a wide range of functionality including sensing, actuation, data storage, communication, visual and audio output. In addition to the growing set of layers created by our research group, we have also provided the necessary prototyping tools to make it easy for anyone to create their own layers for the system as specialized applications demand.

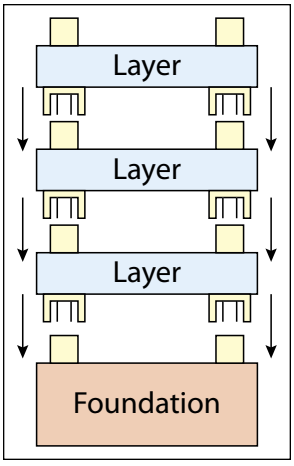


Figure 2.6 - A diagram of the physical structure of the Tower system. In addition to providing structural support, the connectors provide the electrical datapaths between modules.

Every foundation available for the system runs a Logo[◦] programming language interpreter directly on its processor, which allows even novices to program quickly and easily. These "virtual machines" provide users with a Logo-style syntax, but encourage assembly-style[□] programming through the use of low-level processor functions. Normal assembly instructions are supported as primitive functions, so even complex processor-level programs can be written in this language, benefiting from control structures not normally found in common assembly mnemonics. This combination lowers the barriers to hardware development by providing relatively inexperienced electronics designers a low entry point into the technology and an easy means by which to communicate with it in a variety of contexts. Even for more skilled users, the Logo interface provides a means of rapid system prototyping, which allows for robust and functional major system modifications in a short amount of time.

While definitely the most common option, it is not necessary for users to program the Tower in Logo. Any assembly program can be loaded onto a Tower foundation, causing it to function as a basic microcontroller development platform, even when other features of the system are not being used. In addition, some of the more powerful foundations in the system can be programmed directly with commercially available compilers for C and other languages. Even if some compilers cannot communicate directly with the foundations, they can be used to generate the machine-code, which can then in turn be programmed directly onto foundations through our software interface. This wealth of options also allows users to write their own virtual machines if they wish to support existing pro-

◦ See reference [34], Mindstorms by S. Papert.

□ The term "assembly" is used to refer to processor machine-code, the absolute lowest level at which computational systems can be programmed. The term will be used throughout this document.

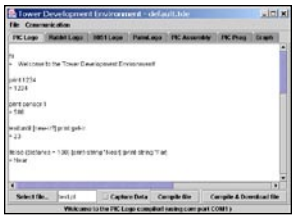


Figure 2.7 - The Tower Development Environment, providing a text-based interface for direct programming of Tower system foundations and layers.

gramming languages, or even to invent their own.

There are many ways to talk to a Tower. The host computer interface software we have created *(Figure 2.7)* is called the Tower Development Environment (TDE). The TDE has built-in Logo compilers to provide low-level access to virtual machine functions on the Tower. Again, the Logo host software is just the current implementation, and users could potentially interface with the Tower in any language they choose. While one could easily control the Tower simply by sending byte-code instructions over the serial line, higher-level languages with more complex control structures and functions have an obvious advantage. Currently, the Tower's development environment supports text-based programming, with program libraries containing pre-written functions available for performing common input and output operations as well as talking to any of the modules connected to the Tower. While the interface software can be used for direct tethered control, user code can also be programmed onto a foundation and mapped to either the on-board start button, or set to run at power-on, allowing the Tower to function completely independent of a host computer.

2.2.3. A Variety of Options

While the Tower represents a complete, integrated system design, it should be emphasized that many of its individual components are strong enough to stand on their own. In fact, the Tower itself finds its place in an entire family of electronics development devices that we have created *(Figure 2.8)*. Based on the needs and constraints of a given application, the best solution can be chosen and employed, while always leaving room for upwards design mobility, if at any time more processing power is needed.

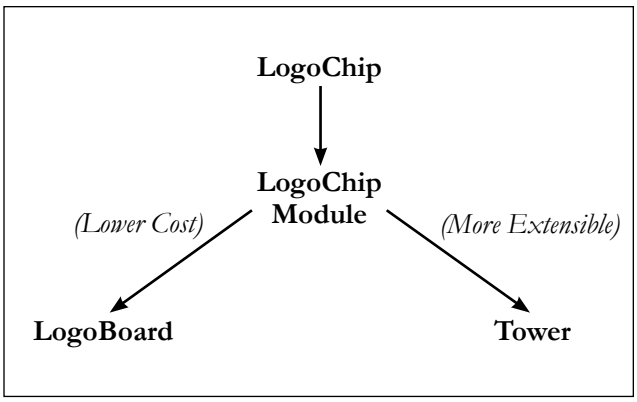


Figure 2.8 - The Tower system family tree, illustrating different hardware options available based on the constraints and needs of a given application.

When the use of a full Tower is not necessary or feasible, individual microcontrollers can be programmed directly via the same software design tools we have created for the Tower. These LogoChips[◦] *(Figure 2.9)* are identical in their processing capabilities to the basic Tower foundation, and their underlying program is carefully designed to provide tight application integration with an actual Tower via built-in multiprocessor communication functions.

Figure 2.9 - The LogoChip

In cases where more than just a chip is needed but a full Tower still is not, other options fall into place. We have created LogoChip modules *(Figure 2.10)*, containing all of the necessary support circuitry for operation, that can be plugged into any standard electronics prototyping boards to allow for rapid development of chip-level projects. In fact, these modules can be easily used to design new layers for the Tower itself, by just connecting a cable between matching ports on the two devices.

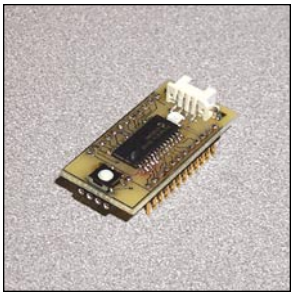


Figure 2.10 - The LogoChip module.

Continuing up the Tower family tree, a branch occurs where different alternatives become available based on a user's preference for a low-cost, or highly-extensible application solution. For those wishing to still have a complete development system, but at a lower-cost, the LogoBoard *(Figure 2.11)* is a viable option. The LogoBoard itself is essentially a scaled-down Tower foundation, with one important difference. While the actual Tower modules use small parts and their assembly is best left to fabrication houses, the LogoBoard can be assembled by anyone, anywhere in the world for less than half of the cost of a Tower foundation. With on-board processing, communication and power circuitry, as well as breakout points for the procesor's I/O lines, the LogoBoard provides for the rapid on-board prototyping of small electronic circuits. The processor itself runs the same firmware as every other device in the Tower family, allowing for tight integration with actual Tower modules as new needs arise. At the decreased cost, however, comes a lack of reusability. While users can easily extend the available on-board prototyping area with add-on modules, once circuits are built they essentially become a permanent part of the module, and it becomes difficult to repurpose them for other projects.

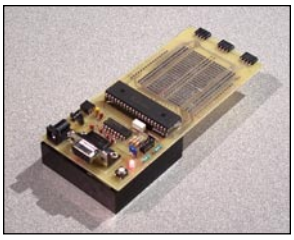


Figure 2.11 - The LogoBoard.

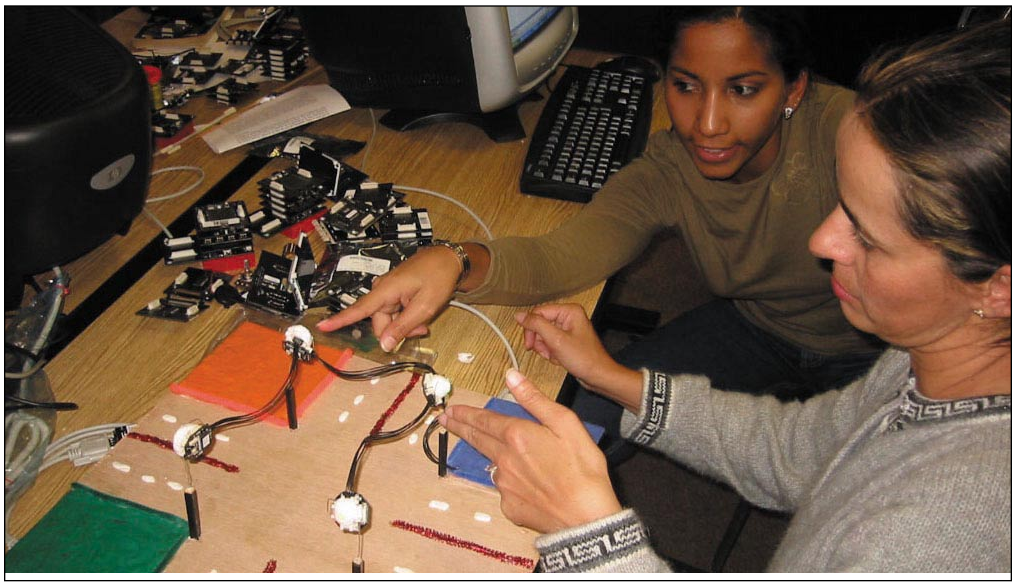
In the other path, lies the most extensible option, the Tower itself. It is important to note that using a Tower by no means implies that the full set of available layers must also be used. Just using a Tower foundation with a prototyping module provides for the development of many possible applications. Functionally similar to the LogoBoard in its having just a processor and prototyping area, this option takes users one step further

◦ The LogoChip project itself was originally a separate initiative for allowing people to easily program embedded software. As the Tower and LogoChip development paths crossed, both projects grew to strengthen each other, and are now at a point where their development is fully intertwined. A more extensive explanation of this cross-development is discussed in section 6.4 of this document. See reference [26], LogoChip project website.

by allowing them to swap prototyping layers between projects as needed. Due to the modular nature of the Tower system, users wishing to take this route have the ability to tie their additions directly into the Tower's low-level system architecture without adversely affecting the operations of other modules. If at any point, users wish to take advantage of the wealth of module resources we have already created, they can begin acquiring Tower layers as needed to perform specific functions. With a wide variety of hardware options currently available, many more under continuing development, and the ability for any user to create their own, the Tower itself represents the most versatile of every device in the Tower family.

Regardless of the option chosen by a user at a given time, it is always possible to migrate to more powerful solutions. With every processor running the same firmware, extensive developmental work is never lost, as all programs written and applications designed can be easily ported to the higher-level devices.

While this section represents a simple technical overview of the system that has been developed, full technical details of its implementation are covered in Chapter 6 of this document.



Workshop participants discuss different ways to extend the functionality of their project at a workshop in Puebla, Mexico.

Chapter 3 - Scenario of Use

Imagine high school students in a rural village. They are using Towers to design science experiments in order to learn about the environment around them. To do so, they have created a weather station: a Tower with a sensor layer, a clock layer, and a wireless communication layer that takes light, temperature, wind, and humidity data readings and sends the time-stamped data back to a base station in their classroom via radio-frequency signals. The base station is equally simple: another Tower with a display layer, a CompactFlash◦ layer, and a wireless communication layer. As the data is received over the radio link, it is stored on the CompactFlash card and plotted in real-time on the display layer. If the students want to do more complicated data analysis and interpretation, they can take the CompactFlash card out of the Tower and plug it into a personal computer. After downloading the data, they can use commercially-available software to plot and analyze the data, comparing it to data taken previously.

3.1. Learning

In this scenario, students are not only learning about environmental systems, but also about tool-building and how to design and implement an application that has an immediate use to them and others. As they analyze

◦ A CompactFlash memory card is one that can be easily read by most standard personal computers and handheld devices, providing for easy cross-platform data transfer.

weather patterns, they gain valuable knowledge about growing seasons and learn how to optimize crop growth by knowing how much to water the plants in relation to given rainfall and humidity data.

More importantly, a learning experience is taking place. Consciously or unconsciously, students are exploring and building objects that are personally meaningful to them, gaining confidence in their design abilities and a basic intuition about higher-level system design. Additionally, the students are solving a real-life technical challenge that has been identified in many rural communities around the world, arriving at a low-cost solution for monitoring climate conditions.

But what if the students want to do something more visual at their base station? Looking through their Tower kit, they find a multicolored LED◦ module. As one of the students starts looking for the document on how to communicate with it, another just plugs it in and tries to figure out how it works. He notices the program include file for the module and after reading the file, gives it a try. He types a command on the computer and the LED lights up red. The other student gives up on looking for the document and together the students start making the LED change to other colors. With just a few simple lines added to their previous program, the LED now fades from red to blue, depending on the temperature being measured outside.

3.2. Extending

The students start to get a bit more ambitious: they want to connect a propeller to the indoor receiver Tower, hoping to make it spin so they can feel just how windy it is outside. Digging around their classroom, they find an old broken electric pencil sharpener and remove the motor from it. After fashioning a propeller blade out of cardboard and attaching it to the motor, they run into a small problem. They don't have a motor layer in their kit because other students making a robot have used all the motor layers for their own project.

What begins as disappointment sparks invention. They find some old electronics books in their classroom and look for a simple circuit for driving a motor, finding one that only seems to need a few transistors. Luckily, there are some electronics parts lying around and they happen to find what they need. But how are they going to connect this to the rest of the

◦ An LED, or Light-Emitting Diode, is a small low-power light that can be turned on and off easily by a microprocessor.

Tower? Perhaps they can use the Tower prototyping layer they still have in their box. Its use seems pretty straightforward to them.

All they have to do is build this little motor circuit on the board, and connect it to a pin of the processor that's already there. By following the example laid out for them, they write a simple program for the processor on the prototyping layer. The first thing the program does is wait for a command to come from the foundation. If it receives a "1", it turns the motor on. If it receives a "2", it turns the motor off. Once programmed, they put their new layer on the Tower and connect the motor to it. Now, just like every other layer, their motor layer needs an special file for the foundation to use in order to know how to talk to it. By copying one of the existing ones as an example and making a few minor modifications, it works on the first try. They are able to turn their fan on and off in the main program. Even though the motor only has two states- on and off, they quickly learn by playing around that they can adjust the speed of it just by turning it on and off quickly in their program, varying the amount of time that it is on for each pulse cycle. The students have not only constructed a working electronic extension for the system but they have also inadvertently stumbled across the concept of pulse-width-modulation, discovering that they can vary the power delivered to a device simply by turning it on and off very quickly.

3.3. Sharing

The students have just successfully extended the system and they can now take things even further. They have the ability to share this new extension with other users around the world. But before doing so, the students would like to make a more permanent circuit board for it. Using special software[◦] for personal computers also under development[□] in our research group, the students can design and lay out a circuit board to be fabricated. Starting with a basic template for a Tower board with the connectors and a processor already on it, the students find footprints for the transistors they used and draw lines where they want the wires to go. Once they have the software generate the actual fabrication files, any circuit board manufacturing house could easily create the board for them.

But fabricating boards can be an expensive process, and if the students are in a remote area the boards would take many months to arrive. For their purposes, they only need to make one or two copies to test before they can

◦ See reference [12], Etch software website.

□ While Etch is not currently in a state of development where it is capable of exporting the necessary fabrication files, other alternatives are available at this point in time. We have created Tower layer templates for many major circuit board layout software packages, which any Tower user can download and use to design new layers for the system.

distribute the design to other Tower users around the world.

The students look around in their kit and find a document explaining how they can use a Tower to build a tabletop milling machine[◦]. It may not look like much: a few motors connected to a moving platform and another motor used to spin a tiny drill bit, but it will get the job done. They put together the pieces and program the supplied program onto their Tower.

In the Tower kit, they have some blank circuit boards with a thin layer of copper attached to the top of them. All the milling machine has to do is remove the copper from the board in between electrical contacts, and they will have a circuit board.

Plugging the Tower into the computer and telling the software to "Make Board", the drill starts spinning and the platform moves around, cutting out the board they have just designed. Once it is done, the students solder on a processor and the Tower connectors (also available in their Tower kit), as well as more of the transistors they had originally used.

After putting the chip programming layer on the Tower, they program their code onto the new processor. When they put the layer on the stack, the Tower reports that it has found their new motor board. Just to be sure that everything is working right, the students want to check the electrical signals on the drive transistors before plugging in a motor, so the motor won't break if something isn't working properly.

The same Tower they have been using all along can be quickly made into a simple oscilloscope[□] by connecting two wires to a sensor layer, and putting a display layer on top. The oscilloscope program is easy for them to understand: it just looks at the sensor value and turns on a pixel corresponding to the voltage level while moving from left to right on the screen, giving a visual representation of the electrical waveform that is being looked at.

Now, the students turn on the motor at medium speed, and use the Tower-based oscilloscope to look at the signal. The voltage at the motor port pulses on and off just as they had expected. It is now time for the next test. They plug in the motor and the fan whirs to life, much to the students' excitement. With the motor layer now working, it is time to put the whole application together and see it functioning.

◦ A milling machine is a device used to cut patterns out of material using a spinning drill head, usually under computer control. The Tower-based tabletop milling machine is a project under development. In its current form, it is only capable of drawing images with a marker. The next step in its development will be the addition of features to enable circuit-board etching. At this point in time, there are several commercially availablile devices that we have successfully used to create circuit boards. The development of the Tower-based milling machine is discussed in further depth in section 4.2 of this document.

□ An oscilloscope is a device used to measure electrical signals over time and display them on a screen, so that users can see what's happening electrically at various points in a circuit. While the Tower-based oscilloscope is currently functionality, it operates at low speeds, which may or may not be sufficient for measuring a motor's drive signal. The development of the Tower-based Oscilloscope is discussed in further depth in section 4.1 of this document.

After putting the wireless communication and display layers back on and connecting the multicolored LED module, they put on their full program, and... nothing works. At first the students are devastated, but then quickly start taking things apart to find out what went wrong. They try putting just their new motor layer on the Tower and it works fine. Thumbing through the documentation, they find an function they can use to scan the Tower and make sure every board on it is working. When they ask the Tower to tell them what layers it has plugged in, it finds their new layer and tells them it is on address 20.

They repeat the process with each layer, testing them one at a time to make sure they work. But when they test the multicolored LED module, they notice something odd. It also has address 20. When they had chosen the address for their new motor board, they had completely forgotten that another layer might be using the same one. The students are sure that the address conflict is the reason why things wouldn't work properly. By reading the documentation, they figure out how to easily change the address of their new motor layer and adjust the program file accordingly.

Plugging everything back together and turning it on yields the desired result. The fan on the Tower in front of them is spinning, its speed directly correlated to the wind speed being measured by the other Tower outside their classroom which is sending back the environmental data.

While the students did run into a problem, they learned something valuable in the process. Accidental learning opportunities such as this one help users gain confidence in their knowledge about how the system works. Instead of just assuming that the Tower will always work, they have learned that even if things do not work properly, it is easy to look inside, figure out the problem, and fix it. When designing complex systems, technical bugs are a natural occurrence and debugging is a powerful skill to be learned.

Now, with everything up and running perfectly, there is only one step left. It is time to share this new layer with the rest of the world. The students head to the online Tower user network◦ and post a description of their new layer, the fabrication files, and the program code needed to make it work. Within minutes, other users around the world will be able to download and build the new module the students have designed.

◦ See Reference [47], Tower Support Forum website.

University students explain their project to classmates at the culmination of a workshop in Cartago, Costa Rica.

Chapter 4 - Suite of Applications

A wide variety of applications have already been developed using the Tower system. Personally, I have designed and built a number of different ones that represent a broad cross-section of the types of applications that users often wish to pursue. Some were built for fun, others as demonstrations, and some initially to test the capabilities of the system. What follows is an overview of six very different projects built with the Tower, along with all the information needed to implement them, including program code.

4.1. Test and Measurement

ScoobyScope *(Figure 4.1)* is a small oscilloscope: a device capable of measuring electrical signals and plotting them on a screen in real time so they can be observed visually. Oscilloscopes are commonly used for debugging electronic projects and ScoobyScope is an example of how easy it is for people to build their own tools using the Tower system.

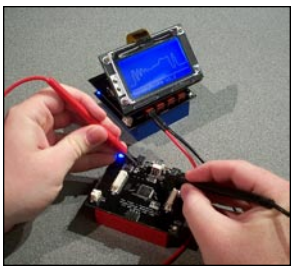


Figure 4.1 - ScoobyScope, being used to measure an electrical waveform generated on another foundation module.

The implementation of ScoobyScope is very simple, using only a sensor layer and a display layer for providing the visual output. Two probes (essentially just wires with handles and pointy tips) are plugged into the

"sense" and "ground" pins of the first sensor port.

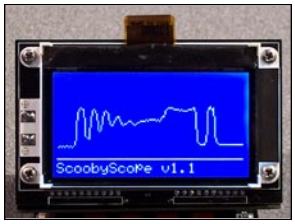


Figure 4.2 - A picture of the ScoobyScope screen displaying a captured waveform.

The Logo program code, which can be found in Appendix U, is equally simple. On startup, it clears the screen and writes the phrase "ScoobyScope v1.1" along the bottom line of the display. Next it falls into a main loop, which takes sensor readings and plots them accordingly. To plot a data point, the current vertical pixel column is erased by clearing every pixel in it and a line is drawn from the previous point to the new pixel location. Each time a sensor value is read, the value is scaled down to a smaller number between 0 and 50 corresponding to a vertical location on the display. Higher values are mapped towards the top of the screen and vice versa. After a data-point has been successfully captured and displayed, the line counter increments. The process then repeats, wrapping back to the left side of the screen when the right edge is encountered. Overall, the screen displays a continually-updating waveform pattern, with the fixed text-string always located on the bottom character line *(Figure 4.2)*.

The rate at which the program takes sensor values is intentionally slowed down to provide a more interesting visual display of sensor data coming in. A great deal of the valuable processing time is used up talking to the display layer each time a point needs to be plotted. If users wanted to speed up the program, they could buffer a hundred or so data points at once from the sensor layer, then write them all to the display in rapid succession. This would allow the program to capture faster moving signals but would lead to short pauses after every block of data has been taken, while the acquired data blocks are being sent to the display. While the sampling speed of ScoobyScope is not yet comparable to that of commercially available oscilloscopes, an important first step has been taken towards eventually being able to use a Tower-based oscilloscope to aid in the design of new modules for the system.

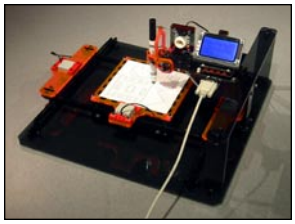


Figure 4.3 - The 2D plotter, built out of about 20 plastic pieces and driven by a Tower running a simple line drawing program.

4.2. Personal Fabrication

The 2D Plotter *(Figure 4.3)* is a simple, two-axis plotting system. It allows users to draw using a computer-controlled Crayola™ marker. Still in its developmental infancy, it will soon be able to communicate with our personal computer design software[◦] and allow people to print their creations. Eventually people will be able to cut out their designs from different materials if the drawing head is replaced with either a knife or spinning drill bit, similar to commercially available table-top milling machines.

◦ See reference [22], Jet project website.

The Tower used to drive the plotter has motor layers for moving the platform around and lifting the pen up and down, and a display layer for providing visual output of the plotting process.

Made up of about 20 pieces of plastic, the plotter has two sets of rails (one for each axis) upon which a central platform slides. The drawing/cutting head is fixed at a stationary point in the middle of the movement plane. Each axis is driven by two motors, positioned directly opposite each other and having rubber wheels mounted on them that contact with the rails.

To move in a given direction, the two motors turn on very briefly in opposing directions, pushing the platform one step in either direction. While this motion is precise enough for many applications, it fails when the platform is met with resistance and slips- a rather common occurrence when attempting to cut rather than draw.

To remedy the problem, a high-resolution optical mouse is embedded into the platform, whose location can be determined through a keyboard/mouse interface layer on the Tower. This allows the program to step the motors until the desired location is reached, regardless of interference encountered along the way. The optical mouse support is not yet fully functional, but has been supported structurally and electrically in anticipation of implementation at a later time.

The Logo program code for the 2D Plotter, found in Appendix V, is quite simple. Four separate functions were written to step the platform in each of its four directions. The functions themselves turn on two motors in opposing directions (necessary due to the motor-mounting method employed), wait a short amount of time, then turn them off. The duration was chosen experimentally to maximize precision while reducing slippage on the drive axes. Two more program functions are used to turn a servo that controls the pen-up and pen-down operation.

The main program is essentially a loop which waits for data packets coming into its main serial port. A data packet contains both a starting x-y point and an ending x-y point for a line to be drawn. As soon as valid data is received, the pen is lifted up and moved to the starting point. Once the starting position is reached, the pen is put back down on the paper and the platform moves to create the desired vector. When the end point has been reached, the pen is raised back up.

While currently functioning with a marker, this basic framework will soon be extended to include cutting capabilities. The current design allows for easy removal and replacement of the head by utilizing a simple two-bolt connection mechanism. In order to extend functionality one only needs to create a new cutting head and attach it to the support arm. For example, a simple cutting head could be made by mounting a razor blade in a free-rotating cylinder attached directly to the arm. As the platform moves, the blade will be dragged along, properly aligning itself for cutting. Another method for cutting could employ a spinning drill bit, which would be powered by another motor and raised and lowered by a mechanism similar to the one that controls the pen movement.

If one is feeling more adventurous, it would even be possible to implement a simple 3D-scanning mechanism by using a head with a stiff wire which can be raised and lowered freely. At each x-y coordinate on the platform, the wire is lowered until resistance with an object below is met, at which point its vertical position is determined and mapped to a height in a 3D computer model. The wire position could be determined by any means, but a simple method would involve making register marks on the wire that could be counted by a reflectance sensor as the wire moves by.

With a basic mechanical configuration in place, the possibilities for creating custom personal fabrication tools with a Tower are limitless. The framework we have created enables anyone using the Tower system to explore the developing field of tabletop fabrication in a hands-on manner, by adapting existing tools to meet new needs as they arise and sharing these new designs with a global community of users.

4.3. Environmental Sensing

The WeatherStation makes use of wireless, multi-Tower communication to take sensor data and send it back to another Tower or personal computer for observation. Designed as an example of both multi-Tower communication, and of how educational classroom activities can be designed using the Tower, the WeatherStation allows students to monitor the environment around them. This provides valuable insight into scientific issues, such as climate patterns and also gives them extensive experience in data collection and analysis.

There are two Towers involved in this setup. The first one is used as a

transmitter *(Figure 4.4)* and has a clock layer, an infrared (IR) communication layer, and a sensor layer with light and temperature sensors plugged in. Data packets are formed containing time-stamped sensor data, and then sent across the infrared link. A second Tower is configured as a receiver *(Figure 4.5)*. Using an IR communication layer and a display layer, it receives the data and then displays it on the screen.

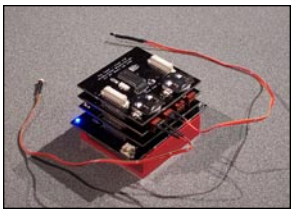


Figure 4.4 - The Tower configured as a WeatherStation transmitter.

The Logo programs for the transmitter and receiver modules in this example are located in Appendix W. The transmitter code is essentially a main loop that sends a value of 255 over IR indicating a packet start, followed by the current hour, minute, and second timestamps, and then the data from sensors 1 and 2. Finally, a zero is sent to terminate the packet, and the process repeats. A delay of one-tenth of a second has been placed in the loop to give the receiving end a chance to display the data before new values are sent. While not absolutely necessary, this ensures that data is processed correctly without skipping packets if the receiving end is busy at the time. The code for the receiver module is equally simple- a main loop first waits for a complete data packet to be received. As soon as the data has been captured, the new time-data is printed on the proper display line, followed by the sensor data further down the screen. Additionally, the data values for sensor 1 are sent out over the serial port at the time of reception. Using the Tower interface software, it is possible to capture these values as a stream of data and plot them on a personal computer *(Figure 4.6)*.

Figure 4.5 - The Tower configured as a WeatherStation reciever.

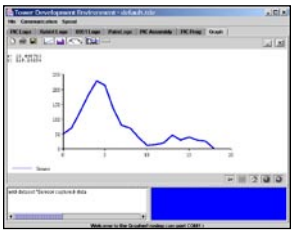


Figure 4.6 - An example of what sensor data would look like when plotting in the graphing portion of the Tower Development Environment.

There are many obvious extensions for an activity like this one. It would be very easy for a user to replace the IR communication layers with radio-frequency (RF) ones for longer-range communication. For example, a multicolored LED module could be connected to the receiving Tower and programmed to fade between different colors, providing a visual representation of different sensor readings. Alternatively, one could store data taken over long periods of time in order to analyze at a later date. The received data could also be written to a text file on a CompactFlash card via the CompactFlash layer. At a later time, the card could be removed from the Tower and the data analyzed by a personal computer using Excel™, Matlab™, or another data processing application.

One main reason this application was developed was to demonstrate how easy it could be for people to build their own tools for monitoring environmental data. In the context of the MIT Media Lab's LINCOS° project, individuals in rural communities had been given self-contained weather-

◦ See reference [24], LINCOS project website.

monitoring stations[◦], created by Richard Fletcher of the MIT Media Lab, which could be placed outdoors and would then send the data back to a host computer for processing. While these stations met the project's needs for environmental sensing, we were curious to see what additional knowledge users could gain when given the tools to build these stations on their own, as well as the ability to extend the functionality of those very tools as they see fit. We have introduced members of the LINCOS community to the Tower system and are looking forward to seeing how they will use it to create their own weather-monitoring stations and other electronic devices capable of accomplishing much more diverse goals.

Figure 4.7 - ALF is an easy-to-program robotic character designed to teach children the basics of procedural thinking in a tangible context.

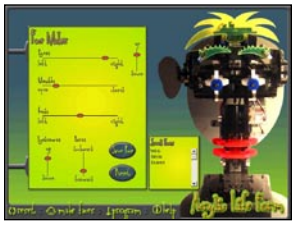


Figure 4.8 - The ALF programming software, which allows kids to program expressions and animations, and play them back in user-defined sequences.

4.4. Robotics

ALF - The Acrylic Life Form[□] *(Figure 4.7)* is an engaging animatronic character that provides children with an easy-to-use introduction to programming and mechanical design. ALF was initially created as an exercise in acrylic fabrication, but its development rapidly continued to the point where it became an active project in our group. ALFs are used to present principles of programming to young children through a tangible, engaging interface.

ALF is an entirely laser-cut robotic head with features that can be easily changed. Through simple interface software[◊] *(Figure 4.8)* written by Andrew Sempere, users can program ALFs to move, talk, sing, and even communicate with each other. We are also developing an easy-to-use graphical layout software[†] that will enable anyone to design new features for ALF and give it a personal touch. The software provides a less intimidating development environment for those who are not as comfortable with the higher-end graphic design software packages such as Illustrator™ or CorelDraw™.

ALF's "brain" is a Tower with a servo motor layer on it. There are six servos connected to the layer which control the movement of different facial features. While ALF can remember and run programs without being attached to a computer, the design and interface software requires a connection to the Tower foundation. Of course, users would not be limited to communicating with ALF over a serial connection. Thanks to the tight integration with the Tower system, ALF could even be programmed via a wireless link.

◦ See reference [11], Environmental Sensing and Instrumentation project website.

□ See reference [43], ALF thesis, and reference [1], project website.

◊ A more in-depth description of the ALF software environment can be found in section 6.3.3 of this document.

† See reference [22], Jet project website.

The Logo code for ALF, which can be found in Appendix X, is essentially a main loop that moves servo motors in response to received serial data. After initializing the servos to their home positions on startup, the program begins waiting for a serial data packet. Following the reception of an arbitrarily chosen start byte of value 101, two more bytes are received- a number corresponding to the feature that should move and the position to which it should turn.

The first value is looked up in a nested if-statement and the appropriate move-function is called with the desired argument. There is a move-function for each feature, which takes the argument and turns the target servo to an angle proportional to the number that was passed to it. Each of these functions was scaled mathematically during the development phase to ensure that none of ALF's servos attempt to extend past their range, thereby causing damage to ALF or others. After completing the motion, the program returns to the main loop and waits for the next command from the PC interface software.

There are many possible ways to extend ALF. It could be as simple as adding a sensor layer, and connecting distance sensors so that ALF would be able to interact with people as they walk by. Add a clock layer and he'll know what time of day it is, so that he can properly greet people whether it is morning, noon or night. Feeling even more ambitious? Put on a speech synthesis layer and connect it to the speaker already in his base to make him talk.

Perhaps users could even write their own interface software for ALF. While the programming environment we use for ALF has evolved through many forms over the course of his development[◦], new options are becoming available to allow children to write their own software to talk to ALF. Another project underway in our research group is making it easier for children to design their own interface programs for handheld computers. Coco[□], being developed by Daniel Bryan, is a software development kit that makes it simple for anyone to create interactive program interfaces *(Figure 4.9)*, and integrate them seamlessly into the Tower world. In fact, Coco programs are downloaded onto devices via the same Tower Development Environment software that is used to program Towers, bringing all stages of project development under the same roof.

Figure 4.9 - A simple ALF interface created using Coco.

◦ For an in depth description of software interfaces that have been written for ALF and other Tower-based applications, please refer to section 6.3.3 of this document.

□ See reference [6], Coco project website.

4.5. Fun and Games

Figure 4.10 - ScoobySnake, a tabletop video game built using the Tower.

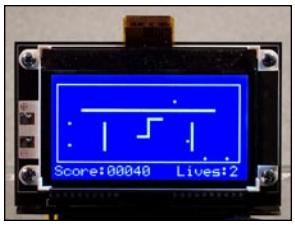


Figure 4.11 - A picture of the ScoobySnake screen displaying a game in progress.

ScoobySnake *(Figure 4.10)*, is a tabletop video game that uses a Tower to run a miniature version of the popular game "Snake". Built into a clear plastic box, the control panel contains a joystick and a start button.

The Tower uses a sensor layer to interface with the button and joystick, a memory layer for storing the high score table, a clock layer to date-stamp the high-score table, and a display layer to act as the viewing screen.

The Logo code for this application is in Appendix Y, and is quite possibly the most complicated Logo program I have ever written, so reader be warned. Essentially, the main game loop handles control over lives and score. After clearing the screen in the main loop, one of five level maps is drawn using the display-draw-line command to make the walls. The snake is then drawn in its home position. After that, 10 pieces of snake food are randomly placed around the level, checking to make sure that they don't collide with either the walls or the waiting snake *(Figure 4.11)*.

Once running, the snake is constantly having pixels added at its head, and removed at its tail to simulate movement. The display itself is used as the computational scratch pad for snake movement via a combination of commands for setting, clearing, and testing the state of individual pixels. If a directional control is triggered, pixels start adding in a new direction. The tail removal algorithm is quite complex, but if you follow it carefully you will see that it has no knowledge of what the snake itself actually is, it just removes the next two solid pixels touching its current point and adjusts the tail-pointer accordingly.

As food is encountered, tail removal is briefly halted to simulate snake growth and the score is incremented. When all ten pieces of food have been eaten, a hole in the wall opens and driving through it takes the player to the next level. If at any point the player dies, the internal "lives" counter decrements and the current level is reset. This entire process repeats until the player has no more lives, at which point the game is over.

Next, the player's score is checked against those in the high score table stored on the memory layer. Starting at the bottom and moving up the table, if the new score is high enough, its proper location is found. The score currently in that position, as well as those below it, are all shifted

down by one and the new data is inserted into the necessary memory locations as the user is prompted to spell out their name using the joystick and start button. The current date is then read out of the clock layer and stored as part of the player's high score entry. Once this process is complete, the game again sits waiting for someone to hit the "start" button.

The act of using the Tower to create a video game was inspired by the Gameweaver[◦] project, also under development in our research group. Initially begun as a thesis project by Alice Yang, Gameweaver is a video game construction kit which is designed to teach programming by allowing children to create games that are personally meaningful to them. The software allows for games to be written that will run on handheld computers, and recent extensions by Daniel Bryan enabled support for Tower-based I/O operations such as button interfaces and visual output.

Building the game entirely using Tower and removing the dependence on a handheld computer was a step that I personally took. The goal was to show how a topic as complex as video game programming could still be broken down into a small set of functional subunits and easily implemented with the system we have created.

Beyond the current implementation of ScoobySnake, there are many options for users to extend the application. People could very easily change the levels, maybe even storing the new ones on a CompactFlash card to trade with their friends. Or how about making another game? The basic control and graphics engine is already in place, and with a little work, anyone could adapt it to make a different tabletop arcade game in the likes of PacMan™ or Frogger™. If one is feeling very ambitious, the networking capabilities of one of the more powerful foundations could be used to create a personal online gaming network with multi-user support.

4.6. International Development

The Diesel Engine Meter was created as part of a special project for use in Vigyan Ashram, a small educational setting just outside of the rural village Pabal, in Maharashtra, India[□]. The primary industry in this village is the creation of small diesel engines which they use for everything from powering motorized vehicles to providing electricity for their daily needs. The engines are crudely built and it is estimated that 60% of fuel is wasted due to poor engine tuning. Fuel is difficult to obtain and quite expensive.

◦ See reference [50], Gameweaver thesis.

□ See reference [3], Engine Tune-Up Project internal document.

The goal was to provide the engine designers with the tools needed to optimize their engine design for fuel efficiency while at the same time teaching them valuable skills needed to successfully apply technology in other aspects of their lives.

Figure 4.12 - Engine designers in India using a Tower to monitor performance of a diesel engine.

Figure 4.13 - The engine flywheel with copper strips attached for interrupting an optical sensor.

The project was led by Professor Isaac Chuang of the MIT Media Lab and Center for Bits and Atoms, Amy Sun of Lockheed Martin, and Dr. Srinath Kalbag of Vigyan Ashram. Together, we set out to create a measurement system using the Tower, that could be used to determine the rotational velocity of the motor flywheel and send the data back to a host computer for graphing and analysis *(Figure 4.12)*. The idea was to measure speed by placing register marks on the wheel and counting the time intervals between them when observing from a fixed point. The marks ended up being little slices of copper attached to clear plastic and wrapped around the wheel. The copper strips would interrupt an infrared reflectance sensor every time they cut through the beam *(Figure 4.13)*. The flywheel would ideally be spinning at more than 2000 rpm and we hoped to have a resolution of at least 50 measurements per rotation, so our program needed to be run exceptionally fast. Due to tight speed constraints, it was necessary to implement the program in assembly code as opposed to Logo. In addition, we wanted to connect the sensor directly to the foundation processor as opposed to going through the sensor layer, since inter-layer communication takes up valuable time.

The meter itself was built with a Tower that had a prototyping layer on stacked on top. The prototyping layer had a reflectance sensor attached to one of the pins capable of performing a high-resolution timing capture. Additionally, red and green LEDs were connected to two standard I/O pins for use as indicators when data was being taken. The green LED toggled on and off each time a mark was encountered on the wheel, and the red LED switching state each time the memory buffer filled up and data was sent back to the host computer.

The assembly code for the application can be found in Appendix Z. The program is interrupt-driven, locking itself in an infinite loop that is broken each time the sensor encounters a mark on the wheel. Once the program jumps into the interrupt routine, the timer register value indicating the amount of time that has elapsed between marks is read and stored in a temporary buffer location. If there is still available space in the buffer, the interrupt service routine is completed and the program returns to its infinite loop, waiting for the next sensor interrupt to occur. But if the buffer

becomes full, its entire contents is streamed out over the serial line to the host computer for graphing or capture *(Figure 4.14)*.

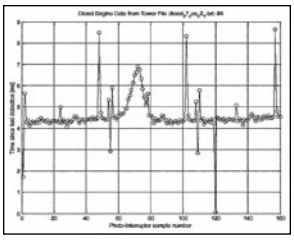


Figure 4.14 - A plot of the time between sensor acquisitons, clearly showing a point in the cycle where the engine slows down significantly, most likely due to a misfiring in one of the engine cylinders.

The local engine designers were very receptive to this new technology, experimenting with the sensing setup and observing the rotational velocity of the flywheel over its course of motion. In the plots generated, they were able to observe interesting patterns including some indicating that one or more of the engine cylinders was misfiring over the course of the stroke cycle. By tweaking their engine designs and observing performance changes in real-time, the designers were able to improve the fuel efficiency of their engines by more than one-hundred percent.

When presented with a simple example such as this one and experiencing first-hand the knowledge gained through its use, the motivation for the local engine designers to continue exploration has been enormous. Using these basic tools for sensing and data analysis, they have been interested to see how the same principles of test and measurement can be extended and applied to other facets of their work. They are actively continuing to use the Tower system to explore the ways in which it can be used to enhance the output their overall design process.

Workshop participants put finishing touches on their projects moments before final presentations at a workshop in Puebla, Mexico.

Chapter 5 - Case Studies

The Tower system has been actively employed in a wide variety of situations ranging from the specific to the general, and from the very technical to the very artistic. While applications differ, the common thread among them has been the degree to which the system has simplified the design process and lead to far richer development of the ideas themselves. The many applications also demonstrate a wide range of involvement by the core Tower development team: from our personal projects, to ones we have actively encouraged and supported, to ones where others have simply used the system on their own to achieve their desired goals.

User case studies can be divided into seven core categories, sorted by our degree of direct interaction with them. Exploring the Tower userbase in this manner provides valuable insight into how the system developed and how it will continue to grow within user communities, even without our direct input and guidance.

5.1. Personal Projects

As a true testament to how powerful and versatile the Tower system is, I have chosen to use it as the electronics core for a number of my personal

projects. From a final project for an academic course to projects directly relating to my research to others done just for fun, the system has successfully met every requirement that I have had for it. My intimate knowledge of the underlying architecture has made it possible for me to not only use, but extend the system as needed and make those new extensions available to the global Tower user community.

5.1.1. Playroom Construction Kit

One of the my most personal Tower applications was my Fall 2002 final project for the Tangible Interfaces course at the MIT Media Lab, taught by Professor Hiroshi Ishii in the Fall of 2002. My team, consisting of Thomas J. McLeish, Elizabeth Sylvan, Carla Gomez-Monroy and myself designed and built an interactive children's playroom construction kit.

We envisioned scenarios where children would explore the functionality of our construction kit by using sensors, sound, and light to create their own play-worlds. We set out to create a set of tools that would be limited only by a child's imagination. The children would start with their own toys and surroundings, then attach sensors and output devices to create fantasy play situations for themselves and others to explore.

While our class assignment was centered on the conceptual design of a construction kit that would enable children to build such an activity to their design, our presentation would be significantly enhanced by a functional demo. Since the goal of the project was to design the activity itself rather than focusing on creating a complex technical implementation, we chose to use the Tower system in order to build a functional prototype quickly and easily.

Figure 5.1 - A "Jungle Adventure" scenario, built as a demonstration of an activity that could be created using the Playroom Construction Kit.

An entire sample play scenario themed around a "Jungle Adventure" *(Figure 5.1)* , was implemented using the Tower system in a period of about four hours. That time frame includes all of the construction, electronics, and programming. The game was started by pulling on a monkey hanging from the ceiling, bringing the whole jungle to life with a number of stuffed animals scattered around the room waking up and making jungle noises. A stuffed chicken began asking for its missing egg, and once the egg was found elsewhere in the room and returned to the nest, the alligator (a cleverly-disguised trash can), proclaimed that it was hungry and demanded to be fed. After tossing the egg into the alligator's mouth, a lion across the room roared to life and said that it was too bright, and he would prefer the

dark. As soon as the lights were turned off, the lion's eyes lit up, and all of the creatures joined in very vocal approval.

Connecting Tower sensing and voice-recording layers to each toy *(Figure 5.2)* gave us the ability to make any of them talk, and also to know when someone approached them or pushed a button. Some toys even received additional layers to drive motors or display multicolored light patterns. Since every layer used in the demonstration is a standard part of the system, no new hardware needed to be developed. All of the toy "nodes" were connected though the main bus cables that are used to tie off-stack layers together. A single Foundation was used to drive the entire activity from a central location.

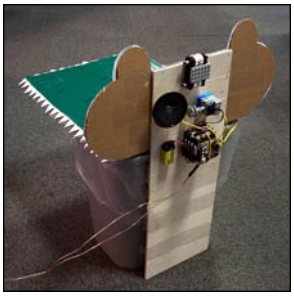


Figure 5.2 - Tower hardware connected to the alligator trash-can, providing voice output, as well as motor control for opening and closing the mouth.

In a more final situation, wireless communication would be ideal but for this simple prototype, wires were acceptable. It is important to note that if wireless nodes were needed, the existing program could be easily adapted to make use of the new radio-frequency communication modules currently under development.

Our actual program was quite simple. It ran through a programmed script for our activity, waiting on specific event sensor triggers and responding by activating voice, light, and motor outputs, then progressing to the next conditional test. Initial testing worked well until a pesky problem was observed. Some layers on the Tower stack began changing their addresses at random, thereby losing communication with the foundation. It was something that we had experienced previously in the development of the system but was proving to be a huge problem now, possibly magnified by the long cables used to connect some nodes together. While a long-term solution to this sporadic problem was later found, for this project I quickly implemented a functional remedy that would ensure that the problem would not bother us again (it was, in fact, the day before our presentation to the class). I simply hard-programmed addresses into all of the boards in the system, preventing them from switching due to noise on the communication lines being interpreted as false address-change commands. While not an ideal solution by any means, it served its purpose and allowed us to present our project to an enthusiastic audience.

The presentation was hugely successful, with Professor Hiroshi Ishii playing the game himself and greatly enjoying it. In fact, the presentation was so engaging that one student in the class even held up a lighter at the end when the lights went out, requesting an encore presentation.

This application represents a situation where my intimate knowledge of the internal Tower system design and my ability to change parameters at the lowest level proved to be of great importance when a bug was found. After the course, developmental efforts were focused on creating a permanent solution to this address switching problem. The problem has now been corrected so that it will not affect other users, many of whom might have greater difficulty solving it that I did.

For our class project, the Tower system enabled us to quickly build a working prototype and rapidly iterate through our design ideas so that we where we were able to present our ideas to a wide audience in a hands-on manner. Time was not wasted with technical details, which allowed us to focus our efforts where they really counted: on the activity design itself. In just a single afternoon, we successfully used the Tower system to model a fantasy play-world construction kit that could allow children to let their imaginations run wild and not be inhibited by technical constraints.

Additionally, this construction kit serves as an interesting example of how designers in the toy industry could benefit from a rapid prototyping system such as the Tower. While different companies currently employ engineers to create proprietary electronics for their designs, there would be a significant benefit in standardizing electronics development across the industry. In the spirit of the MIT Media Lab's Toys of Tomorrow special interest group, an electronics design toolkit such as the Tower would promote interoperability of toys across a spectrum of manufacturers. In an industry driven by tight cost constraints, a great benefit could be achieved by using the Tower. Even more importantly, toy designers would be able to spend their time designing new unique products without losing valuable time struggling with specific technical implementations.

Figure 5.3 - ScoobySnake, a tabletop video game built using the Tower.

5.1.2. ScoobySnake

As discussed in Chapter 4, ScoobySnake *(Figure 5.3)* is a tabletop version of the classic video game "Snake" built with Tower modules. The act of using the Tower to create a video game was inspired by the Gameweaver◦ project, also under development in our research group. Initially begun as a thesis project by Alice Yang, Gameweaver is a video game construction kit which is designed to teach programming by allowing children to create games that are personally meaningful to them. The software allows for games to be written that will run on handheld computers, and recent extensions by Daniel Bryan enabled support for Tower-based I/O opera-

◦ See reference [50], Gameweaver thesis.

tions such as button interfaces and visual output.

Building the game entirely using Tower and removing the dependence on a handheld computer was a step that I personally took. The goal was to show how a topic as complex as video game programming could still be broken down into a small set of functional subunits and easily implemented with the system we have created.

5.1.3. 2D Plotter

The 2D Plotter *(Figure 5.4),* discussed in more depth in Chapter 4, is a simple, two-axis plotting system. It allows users to draw using a computer-controlled Crayola™ marker. Still in its developmental infancy, it will soon be able to communicate with our personal computer design software[◦] and allow people to print their creations. Eventually people will be able to cut out their designs from different materials if the drawing head is replaced with either a knife or spinning drill bit, similar to commercially available table-top milling machines. Creating new heads for the plotter is a simple process due to its underlying modular structure.

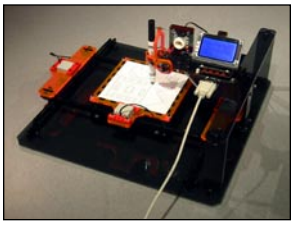


Figure 5.4 - The 2D plotter, built out of about 20 plastic pieces and driven by a Tower running a simple line drawing program.

The Tower used to drive the plotter has motor layers for moving the platform around and lifting the pen up and down, and a display layer for providing visual output of the plotting process. With a basic mechanical configuration in place, the possibilities for creating custom personal fabrication tools with a Tower are limitless.

5.1.4. ScoobyScope

ScoobyScope *(Figure 5.5)*, introduced in Chapter 4, is a small oscilloscope: a device used to measure electrical signals and plotting them on a screen in real time so they can be observed visually. Oscilloscopes are commonly used for debugging electronic projects and ScoobyScope is an example of how easy it is for people to build their own tools using the Tower system.

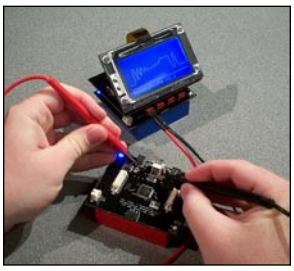


Figure 5.5 - ScoobyScope, being used to measure an electrical waveform generated on another foundation module.

The implementation of ScoobyScope is very simple, using only a sensor layer and a display layer for providing the visual output. Two probes (essentially just wires with handles and pointy tips) are plugged into the first sensor port. While the sampling speed of ScoobyScope is not yet comparable to that of commercially available oscilloscopes, an important first step has been taken towards eventually being able to use a Tower-based oscilloscope to aid in the design of new modules for the system.

◦ See reference [22], Jet project website.

5.1.5. WeatherStation

As discussed in Chapter 4, the WeatherStation makes use of wireless, multi-Tower communication to take sensor data and send it back to another Tower or personal computer for observation. Designed as an example of both multi-Tower communication, and of how educational classroom activities can be designed using the Tower, the WeatherStation allows students to monitor the environment around them. This provides valuable insight into scientific issues, such as climate patterns and also gives them extensive experience in data collection and analysis.

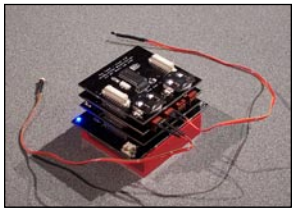


Figure 5.6 - The Tower configured as a WeatherStation transmitter.

There are two Towers involved in this setup. The first one is used as a transmitter *(Figure 5.6)* and has a clock layer, an infrared (IR) communication layer, and a sensor layer with light and temperature sensors plugged in. Data packets are formed containing time-stamped sensor data, and then sent across the infrared link. A second Tower is configured as a receiver *(Figure 5.7)*. Using an IR communication layer and a display layer, it receives the data and then displays it on the screen.

Figure 5.7 - The Tower configured as a WeatherStation reciever.

The setup itself is completely reconfigurable. For example, different types of sensors could be used, or the IR communication layers could easily be replaced with radio-frequency (RF) ones for longer range transmission. In an educational setting, the possibilities are endless. One main reason this application was developed was to demonstrate how easy it could be for people to build their own tools for monitoring environmental data.

5.2. Grassroots Invention Group

A number of projects currently under development in our research group utilize the Tower system as their functional core. While these projects are not ones I am personally guiding, I have been directly involved in the continued hardware development as needed to realize them, and they have all significantly benefited from the intimate system knowledge of the core Tower development team.

5.2.1. Tabletop Process Modeling Toolkit

The Tabletop Process Modeling Toolkit◦ is being developed by Tim Gorton, the lead software developer for the Tower system. It provides a hands-on learning space for teaching the principles of systems design and process modeling.

◦ See reference [16], Tabletop Process Modeling Toolkit thesis, reference [17], conference paper, and reference [44], project website.

Developed in conjunction with one of our sponsors, the United States Postal Service, the system was built to model mail flow through the Boston South-Station mail processing facility *(Figure 5.8)*. The goal was to build a tangible representation of a flow-chart that modeled their facility and find a way for managers to interactively adjust system parameters in order to fine-tune its operation.

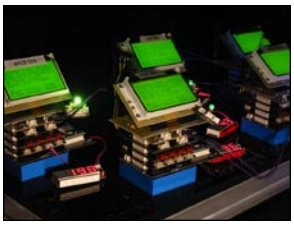


Figure 5.8 - The Tabletop Process Modeling Toolkit being used to simulate mail flow at the USPS Boston South Station mail processing facility.

The technical problem is a classic one- how to optimize complex real-world processes. The engineers from the post office told us about their predicament. Software packages to perform these modeling simulations are readily available, but need to be run by engineers. The effectiveness of these existing tools is significantly limited by the challenges engineers face when communicating the results to managers. However, once confronted with the results, many managers pass them off as untrustworthy and counterintuitive, and disregard them entirely. In order to overcome these communication barriers, Tim set out to create a tangible toolkit that would allow the managers themselves to play an integral role in the design and manipulation of these dynamic simulations. Additionally, a tabletop model provides the ability for small groups of people to gather around it and actively collaborate.

In the model Tim created, the machine and human operations in the factory was represented by a Tower using assorted displays, lights, and sensors to interact with the user. When a virtual "mail-truck" pulled up at one node, packets would begin flowing through the network, simulating blocks of mail. Lights would turn on, indicating the activity level of certain machines and how under or over capacity they were. Managers could turn dials to adjust flow rates, operating times, and readily see the overall productivity of the process increase before their eyes as they collaboratively played through different scenarios of system operation. Results of the entire simulation could be viewed in a custom software environment◦ *(Figure 5.9)* written to interface directly with the Towers.

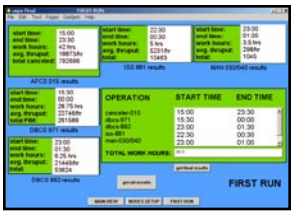


Figure 5.9 - The Tabletop Process Modeling Toolkit software interface.

To implement this project successfully, it was necessary for Tim to develop a new foundation for the Tower system based on a more powerful processor that has much greater built-in support for packet routing and network control. Additionally, a serial-routing layer was created to pass data between Towers. The new layer contained four on-board serial ports that could be connected to any of the other Towers in a user-defined configuration.

◦ A more in-depth description of the Tabletop Process Modeling Toolkit software environment can be found in section 6.3.3 of this document.

Figure 5.10 - A miniature instant-messaging application designed to teach kids about the ways in which individual nodes on networks communicate with each other both directly, and through central servers.

Figure 5.11 - The CodaChrome programming environment.

Figure 5.12 - A light-up hat, made using the CodaChrome system to program complex temporal patterns for visual output.

This example scenario was hugely successful and Tim is now using the same low-level network infrastructure he built for this application to teach children about the principles of networking and data communications. Using the networking toolkit he created for the Tower, children can build their own Tower-to-Tower messaging networks *(Figure 5.10)*, and even write simple multiplayer "online" games such as Tic-Tac-Toe and Battleship™. Implementing this scenario required the development of a keyboard interface layer for the Tower, thereby allowing the children to just plug in any keyboard and use it to communicate with their Tower program. We had already considered making the keyboard interface layer in the past because we felt it could prove useful in many other applications as well, but this project was the reason it was developed at this time.

5.2.2. CodaChrome

Another Tower-based project in our immediate research group is the CodaChrome◦ system, being developed by Margarita Dekoli. The system employs a specially designed software environment□ *(Figure 5.11)* that gives users a full range of control over the design and programming of spatio-temporal light patterns that can be easily integrated into physical objects, which they design themselves. The system makes use of programmable light modules that can display and transition between millions of colors. In initial workshops, professional jewelry artists worked with novice designers, using the system to create new forms of interactive jewelry and wearable art *(Figure 5.12)*.

With the CodaChrome system, everyone from professionals to artistically-inclined beginners can create impressive light patterns to accent their constructions. By using the system, they can gain valuable insights into the creation of complex dynamic patterns and how those patterns evolve over time in the context of creating interactive art.

To implement the CodaChrome project using the Tower, we needed to create new multi-colored LED modules for the system. While the core functionality of displaying RGB values on LEDs was adapted from a device I had developed as part of the MIT Media Lab's Cricket◊ system a few years before, the new addition was the ability to control precisely-timed fade patterns on each LED. This feature was critical to the success of the CodaChrome project, as users would need to have full control over time-based fading light patterns.

◦ See reference [10], CodaChrome thesis, and reference [7], project website.

□ A more in-depth description of the CodaChrome software environment can be found in section 6.3.3 of this document.

◊ See reference [29], Metacricket paper, and reference [8], Cricket system website.

But more than just creating these new modules, a fundamental system expansion was needed to realize their full potential. We were faced with the challenge of how to make these modules small and capable of being embedded in artistic applications without breaking the modular structure of the Tower system that we had worked so hard to create.

To solve this problem, we also created a connecting layer for the Tower, providing points at which layers not directly connected to the main Tower stack, such as the multi-colored LED module, could be attached. With this configuration, even these "off-stack" layers could be connected to the main Tower via the same protocol used to communicate with every layer directly on the stack. This solution proved ideal. It provided the hardware flexibility we needed while preserving the modular nature of the system's software design. Both of these layers are currently some of the most popular ones in the Tower system. The are critical to many of the projects under development in our research group and are currently being used in multiple countries around the world.

5.2.3. Robotany

Robotany[◦] is an artificially intelligent robot created by Sara Cinnamon that allows plants to care for themselves in the home environment. Typically, several of these robots are active at the same time which leads to opportunities for cooperation and competition among these completely autonomous agents. Using the Tower system to remove the burden of input and output (I/O) processing from the main program frees more resources for the artificial intelligence calculations, and allows for faster response times and more realistic behavioral patterns.

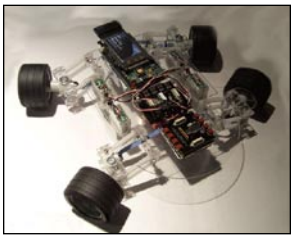


Figure 5.13 - One of the Robotany mobile robots, using a Tower to control motor and sensor operations.

The robot itself *(Figure 5.13)* uses a Tower with layers for sensing and driving motors, to provide precision control over motion and obstacle navigation. Due to the complex nature of the robot's artificial intelligence algorithms, an even more powerful processor was needed. This led to the development of a new Tower foundation that is based on a commercially-available handheld computer core that can run either the Linux or Microsoft[□] Windows CE™ operating systems. With a high-speed version of our virtual machine also available, Tower layers can be used for extremely high-end applications while also providing features normally difficult to obtain on a linux-based personal computer, including pin-level digital I/O and direct analog inputs.

◦ See reference [41], Robotany project website.

□ See reference [31], Microsoft corporate website.

While this Foundation is currently only used in this particular application, we will soon be making it available to the wider Tower user community, hoping that it will find a niche will developers of high-end heavy-processing applications. In fact, many other researchers at the MIT Media Lab currently use this same higher-end handheld computer core, and we are hoping that as our development thread stabilizes the other researchers will be interested in seeing how the Tower system can enhance their own projects, and increase the speed of their iterative development cycles.

Figure 5.14 - ALF is an easy-to-program robotic character designed to teach children the basics of procedural thinking in a tangible manner.

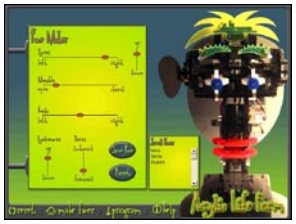


Figure 5.15 - The ALF programming software, which allows kids to program expressions and animations, and play them back in user-defined sequences.

5.2.4. ALF - The Acrylic Life Form

As previously discussed in Chapter 4, ALF - The Acrylic Life Form[◦] *(Figure 5.14)* is an engaging animatronic character that provides children with an easy-to-use introduction to programming and mechanical design. I personally created ALF, but Andrew Sempere is now actively extending the project through new interface software[□] design *(Figure 5.15)* and he is also running workshops with young children, giving them the chance to build up complex multi-ALF interaction scenarios.

ALF is an entirely laser-cut robotic head with easily-interchangeable features. His brain is a Tower with sensing and motor layers, but we are currently in the process of giving him the ability to speak. In fact, ALF is the motivation behind our design of a new Tower layer for allowing children to record their own voices into ALF, and be able to play them back at a later time in conjunction with his movements. Providing children with the ability to give ALFs their own voices deeply personalizes the learning experience for them. While the voice-recording layer had already been planned for development, this immediate need brought its development to the forefront and it is leading to the design of a full audio subsystem for the Tower, including everything from speech recognition to music synthesis, that will be useful to the many people wishing to work with sound in their Tower-based projects.

ALF, as well as all of the other applications discussed above, was technically demanding and required system extensions in order to reach its full potential. The important observation is how easily the system was extended and how the design architecture supported these extensions. While needed for specific projects, these extensions benefitted the system as a whole, providing a greater depth of technical infrastructure to user communities around the world.

◦ See reference [43], ALF thesis, and reference [1], project website.

□ A more in-depth description of the ALF software environment can be found in section 6.3.3 of this document.

5.3. MIT Media Lab

Outside of our immediate research group, the Tower is also being used elsewhere in the MIT Media Lab because it provides a reliable prototyping system for both students and faculty. Even technically skilled individuals have turned to the system to quickly visualize their ideas in the form of functional models.

Students in other research groups have employed the Tower as a means of creating activity-oriented construction kits for teachers and students to use directly. By using the Tower system to enable early system prototypes, they were free to devote their attention to the actual activity development itself without having to worry about the precise details of hardware implementation.

5.3.1. Flogo

Chris Hancock of the Lifelong Kindergarten group has used Towers as the electronic building blocks for his Flogo[◦] robotics workshops. Flogo is a programming language and environment designed to enhance students' abilities to construct complex autonomous behaviors for robots, kinetic sculptures, and other animated creations *(Figure 5.16)*.

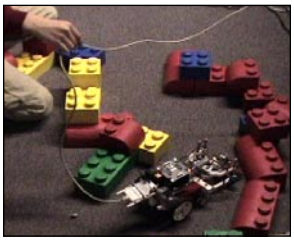


Figure 5.16 - A Tower-based mobile robot, programmed by children using the Flogo language to navigate its way through a maze.

For his Flogo robotics workshops, Chris used Towers with layers for sensing and driving motors. Programming the robots was accomplished by serial communication with a host computer, but Chris is hoping in the future to make use of the new wireless communication layers for the Tower that we are currently developing.

By writing a simple program for the Tower that receives serial data from a personal computer and processes the sensing and actuation functions accordingly, Chris was able to tightly integrate his software with the Tower, thereby providing a compelling hands-on programming experience for the participants in his workshops. By relying on the Towers to control the physical functions of the robots, Chris was free to spend time developing his core research: the Flogo language itself.

5.3.2. System Play

Oren Zuckerman, another student in the Lifelong Kindergarten group, has been using the Tower to implement a prototype of his System Play[□]

◦ See reference [19], Flogo paper, and reference [14], Flogo project website.

□ See reference [51], System Play conference paper.

Figure 5.17 - A four-year-old girl playing with a network of System Play blocks.

project, a set of building blocks designed to teach the concept of system dynamics to children◦. The System Play blocks *(Figure 5.17)* are designed to promote system-thinking intuition by allowing users to configure small networks of blocks, each representing a different concept in system dynamics such as feedback, causal loops, accumulators, flows, interdependencies, side effects, and delays.

Each block in the system contains a Tower with enhanced networking capabilities and layers on top for sensing and display. Dials and buttons are connected to the sensor inputs on the Tower and are mounted on the outside of the blocks, giving users direct input over the parameters controlling the functionality of a specific block.

This project builds upon themes similar to those explored in the work that Tim Gorton began at the MIT Media Lab within in our research group, and in fact uses the same hardware infrastructure that Tim helped to create for his own applications. Using the Tower system to quickly build a working prototype of the System Play blocks allowed Oren to focus on the design of the blocks themselves and the activities that would be used to introduce them to children.

Figure 5.18 - A Topobo structure, built with a network of interconnected intelligent modules and capable of memorizing and playing back kinetic patterns.

After a few months of exploration using this prototype, Oren moved to the next step and is now fabricating custom circuit boards for the blocks, which use LogoChips as their core processors. This project is an ideal example of how someone might use the Tower system to prototype complex ideas in a highly-iterative manner, and then later design custom electronics to embed in a more permanent application.

5.3.3. Topobo

In addition to those whose work is using Tower system to enhance learning opportunities, others in the lab are using the system for more artistic and design-oriented applications. Hayes Raffle, of the Tangible Media group, is using the Tower to prototype Topobo□, a modular system for creating dynamic programmable kinetic structures.

In an attempt to explore the ways in which patterns emerge in artificial networks, Hayes has created a connected structure of identical structural modules *(Figure 5.18)*. Each module contains sensors and actuators, giving it the ability to record and play back sequences of motions created by a user. Additionally, users are able to input motion data directly via a

◦ This work was partially inspired by the Function Blocks created at the MIT Media Lab. The Function Blocks emphasized the building-block nature of digital systems, giving children the ability to design simple programs using blocks representing different logic operators. See reference [40], Digital Manipulatives proposal.

□ See reference [46], Topobo project website.

programming "widget," allowing for the physical visualization of complex dynamic behaviors.

Each module in the system is programmed to receive data, act upon it, and pass on a modified version of the data. This network configuration allows complex structural patterns to emerge based on the topology of the overall structure. If motion commands are being sent through a chain of modules, interesting shapes tend to appear. For example, if a "bend" command is sent to the first module, it will physically angle itself in respect to those on both sides of it. If the same command is passed down the chain, a circle structure will eventually emerge. However, if the modules are set to scale up the bend angle before passing it along, a spiral pattern will form. There are numerous possibilities for allowing inputs to the system at any node, as well as creating branch patterns and seeing how the patterns react when they encounter irregularities in the structural chain.

An early prototype of the Topobo structure has been built using custom electronics, but Hayes is hoping to use the Tower system to test out his new ideas for more complex module networking procedures. While this project is still in its early stages of development and will eventually require the fabrication of miniature electronics to embed within the modules, Hayes is hoping to use the Tower as a means of testing out the feasibility and functionality of his ideas in a low-risk, highly-iterative environment.

Topobo and the other projects outlined above are just a few examples of the varied ways in which people are using the Tower system within the MIT Media Lab's research community. It is intriguing to see how individuals possessing the technological skills to create their own electronics choose to use this system as a means of prototyping new hardware designs. In an environment where working demonstrations of big ideas are a highly-valued commodity, the ability to create functional models quickly and efficiently has proven to greatly increase the productivity and focus of those choosing to use it. Rather than getting bogged down in the details of a specific technical implementation, the Tower system is allowing people to focus their valuable time on the issues directly pertaining to their actual research questions.

5.4. How to Make Almost Anything

In the fall of 2002, I was the microcontroller electronics Teaching Assis-

tant for the "How to Make Almost Anything" course at the MIT Media Lab. Teaching microcontroller programming to a group of 15 students, mostly from the fields of art and architecture, proved to be a challenging task.

In previous years, students had just been given a circuit board to assemble and a pre-written program for them to load onto the processor. While the students were able to assemble and program the boards as instructed, it was not often clear what longer-term benefits they would gain from doing so.

As a departure from the old method, we gave each student a Tower and LogoChip. We then spent about one and a half hours teaching them the basics of programming the devices in Logo. By the end of that first class, almost all of the students were successfully turning LEDs on and off in response to sensor input. It is a major achievement for someone new to electronics to accomplish this in such a short time.

The students in the class were given a homework assignment to "make something using the Tower that produces an interesting output." They were supplied with a wide variety of LEDs and other fun things that light up. While most students experimented with displaying patterns on LED arrays, often dynamically changing them in response to sensor readings, a few others explored the "output" qualifier of their assignment in further depth.

5.4.1. A Tactile Output Device

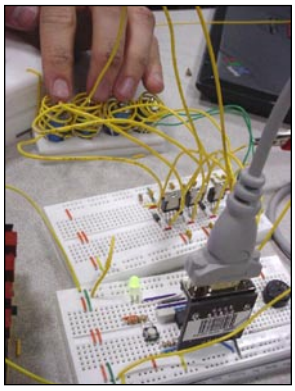


Figure 5.19 - A tactile output device using the Tower to trigger small solenoids in response to sensor inputs.

Axel Kilian and David Merrill created a tactile output device *(Figure 5.19)*. To use the device, a person would place their hand on top of it and five miniature solenoids would pulse to convey data to the user in a physical manner. Since solenoid driver modules had not yet been created for the system, it was necessary for them to create their own driver circuits. While neither of these students had significant experience in electronics, after being pointed in the right direction they were able to extend the functionality of the Tower system to the point where it could control a series of solenoids, thereby making their project possible.

5.4.2. Moto Photo

Another weekly project that took a unique approach to visual output was

Parul Vora's Moto Photo *(Figure 5.20)*. The concept was to represent a unique way by which people could experiment with different clothing options, by providing a manner for them to combine different picture fragments in a variety of patterns. She created four separate motorized wheels, each containing an array of pictures representing different looks for that portion of the body. Using a Tower to control the movement of the four wheels, assorted combinations could be selected and viewed by indexing the wheels at different positions. To implement the project, she needed to build simple motor-driving hardware since those Tower modules had not yet been released for public use. By experimenting with motor control by toggling pins on and off, she learned how to control both the speed and direction of motors, and her project was a success.

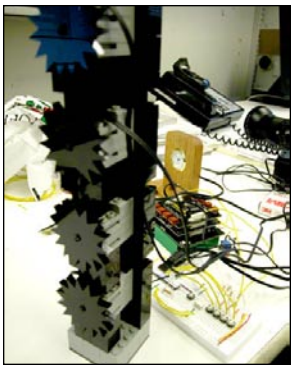


Figure 5.20 - The Tower-driven motor structure used to rotate the Moto Photo picture wheels.

5.4.3. The World's Smallest Violin

For their homework assignment, Brian Alex Miller and Kyle Steinfeld worked to create a miniature violin *(Figure 5.21)*. The idea was to have the strings fixed in space and use two small motors to move the bow over them. One motor would move the bow back and forth over the strings while the other would move it up and down, allowing it to angle in order to make contact with the different strings. Two input devices were used: one slider and one rotating dial to control the positions of both motors. Since the motors they were using were so small, they were able to power them just with pins from the LogoChip itself, and did not have to create more elaborate drive circuitry. By the end of the week, they had successfully created a working model built out of acrylic.

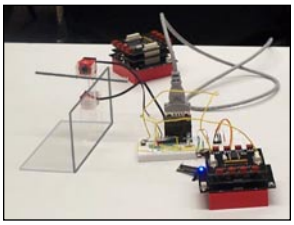


Figure 5.21 - A model of a miniature violin played by a Tower controlling two micro-motors.

The students truly got into the spirit of making things with the Tower and LogoChip, creating functional and often quite complicated projects in the span of just a week. One of the impressed instructors even commented that our tools had made electronics "too easy" for the students. In only a week, some students had created applications as technically complex as final projects in the course typically are.

5.4.4. A Giant Music Box

With the products of the weekly assignment being so impressive, the final projects themselves were orders of magnitude more complex. One of the more impressive final projects created with Towers and LogoChips was a giant motorized music box *(Figure 5.22)*, made by Brian Alex Miller and Parul Vora. Essentially, there were eight pipes of different length, each

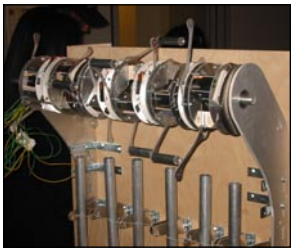


Figure 5.22 - A giant music box using a Tower to control hammers that would strike hollow sound pipes based on a predetermined note sequence.

tuned to a respective musical note. A horizontal column of solenoid-driven hammers rotated next to the pipes, activating the hammer corresponding to the note that was to be played. Each pipe had one hammer wheel containing three separate hammers that allowed notes to be played frequently despite the slow rotation of the hammer column. Because of this structure, it was necessary to drive a total of twenty-four solenoids. Based on his experience with driving solenoids for the weekly project, Brian built twenty-four interconnected solenoid driver circuits to handle the hammer action.

Figure 5.23 - A position encoding wheel for the Giant Music Box.

For the hammers to trigger properly, it was important to know the rotational position of the column at all times. To accomplish this, they made a plastic encoding wheel with a set of cutouts that were different for each segment of the wheel *(Figure 5.23)*. By mounting the wheel at one end of the hammer column, and using reflectance sensors aligned with the slots in the wheel, it was possible for them to determine the position of the column with reasonable accuracy. In the same spirit, interchangeable metal disks were made to encode different songs. Just as the position disk rotated with the column, so did the song disk, with punch patterns corresponding to the sequence of notes to be played. Similar in operation to how a player-piano reads music, playing a different song was accomplished by simply switching the song disk at the end of the column.

This project was an ideal example of the design and engineering process at work, demonstrating how using the Tower system allowed students to focus their efforts almost exclusively on the design of the project itself without having to worry excessively about precise technical details of its operation.

It is clear that using the Tower system as a teaching tool in the "How To Make Almost Anything" class proved to be an empowering experience for many of the students. The class brought together students from diverse backgrounds and allowed them to develop and explore their own applications of the technology and extend the system as they saw fit, with only a supervisory level of assistance from the core Tower development team. Perhaps the most important observation from this experience is that the Tower system remained extremely useful even when other, off-the-shelf solutions for microcontroller electronics design and programming were available to the students.

5.5. Costa Rica / ITCR

The first users of the Tower system outside of the United States were university students at the Instituto Tecnológico[◦] (ITCR) in Cartago, Costa Rica. Over the past several years, we have been steadily building a collaboration with the university, organizing the LuTec project as part of Esperanza, the Learning Independence Network[□] in Costa Rica. LuTec seeks to contribute significantly to Costa Rica's development by helping to create interconnected communities of academic and industrial entities and encouraging interactions between the educational, research, and outreach initiatives at the university.

A key component of the LuTec project is an academic course created by three professors at the university working in conjunction with Bakhtiar Mikhak of the MIT Media Lab. Entitled "Development by Design" (as inspired by a similarly named student-run conference held at MIT), the course is comprised of students from the departments of Electrical Engineering, Computer Science, and Industrial Design. Students in the course are encouraged to create pieces of technology that will benefit the people of nearby rural communities in the context of strengthening local industries, providing new educational opportunities, or directly improving the lives of individuals by means of improved health monitoring and care.

For the first two terms the course was offered, the students built their projects from raw electronic components, using microcontrollers and other assorted parts. While their project ideas were wonderful, many of the students lacked the means or sufficient time to implement their projects to a functional state within the course of the term. At best, something would just start to work as the term ended, leaving them little time to improve upon their original design.

However, as a testament to the students' motivation, most continued on past the end of the class to continue working on their projects. Even though basic functionality would eventually be achieved, they would miss out on the opportunity to iterate their designs and develop the core ideas behind them to a greater depth.

We were interested to see how these students would make use of a system such as the Tower so we provided them with ten Tower kits in time for the third offering of the course. Additionally, Bakhtiar Mikhak, Tim Gorton,

◦ See reference [21], Instituto Tecnológico de Costa Rica website.

□ See reference [25], Learning Independence Network project website.

and myself (the core Tower development team), traveled to Costa Rica in order to run a training workshop with those who would be teaching the new course. In an intense three-day workshop, we worked with the faculty and student teachers (many of whom had been enrolled in the class the previous term) to prepare them with everything they would need in order to teach the new students how to use the Tower system as a way to rapidly prototype their project ideas.

Over the period of those three days, we worked around the clock with the teaching staff to help implement some of their projects from the previous terms with this new technology. The speed at which they learned to use the Tower system was remarkable. Ideas from previous terms that had never made it past the design phase were up and running within just a day or two.

5.5.1. Tekno the Fire Dog

Figure 5.24 - Tekno the Fire Dog, with a fan on his collar to put out miniature "fires".

One project in particular was a robotic dog designed to teach fire safety to children *(Figure 5.24)*. The dog would walk around looking for a "fire" (a bright light), and when he found it, he would turn on a fan attached around his collar to "put the fire out". For this project, the students needed the ability to drive high-powered motors. At that time, motor layers were unavailable for the system. After researching ways to drive motors, the students decided that they needed to build a circuit out of transistors. With a bit of guidance from us, they assembled the circuit on one of the Tower prototyping layers and by connecting the control lines to the processor, they were able to write a simple program to make this homemade "motor layer" function just like every other layer in the system.

The program on the layer was quite simple. It waited in a loop for data to be sent to the layer from the foundation and once data was received, it would process it and act accordingly. If the received data was a "1", the layer would turn the motor on in the forward direction. If the data was a "2", the motor would turn on in reverse. Sending a "3" would turn the motor off. Talking to their new layer was accomplished in the same manner as every other layer in the system: send either a "1", "2" or "3" to the layer depending on the desired motor function. Within a single afternoon, the students had extended the functionality of the core system to meet the needs of their application. But the one step that remained for them to complete was finding a way to turn their prototype into a more permanent layer that they would be able to share with other users around the world.

One important aspect of the Costa Rican site is that it is also home to one of our FabLabs[◦], a portable fabrication suite containing tabletop equipment such as a miniature milling machine, vinyl cutter, oscilloscope, and microscope *(Figure 5.25)*. As part of the outreach initiative of the National Science Foundation's Center for Bits and Atoms, our research group has been directly involved in the development and deployment of these miniature fabrication laboratories to countries around the world. Providing these tools to rural communities around the world gives students the ability to create things they could have never even imagined before. While the Towers are currently included with the FabLab toolset, eventually Tower-based milling machines, oscilloscopes, and other fabrication equipment will replace the commercial machines currently employed, illustrating the power behind building one's own tools to all FabLab users.

Figure 5.25 - The Costa Rica FabLab, complete with tabletop milling machine, vinyl cutter, oscilloscope, and more.

With the FabLab, students were able to produce a prototype of their motor layer by using the milling machine to etch their design onto a copper-coated board. After fabricating and testing their new circuit board, students were able to share the design with other users via the online Tower user network. In addition to creating circuit boards, other students at the university have used the FabLab equipment to develop a variety of parts for their own projects.

5.5.2. The Clock that Teaches

One group in the class had been working on a device to help children learn to tell time[□]. Designed as designed for use in a school classroom or similar environment, the idea was to have large-scale parallel representations of digital and analog clocks, alongside a display showing time-zone information *(Figure 5.26)*. In one activity, a time would appear on the digital display. The children would then have to stand in front of the child-size analog clock face and use their own hands to represent the hour and minute hands of the matching time. Distance sensors located around the clock face would detect the positions of their hands, and the program would convert that to a time value, which could be compared against that represented on the digital display. In another activity, lights on the clock face would then indicate a different time of day and the children would then have to make the digital display match by pressing buttons mapped to different numerical values.

Figure 5.26 - The Clock that Teaches, a wall-mounted digital and analog clock that encourages children to convert time back and forth between the two representations.

At a later point in time, the activity was extended to reach a broader age group of children. For older users, the clock was also used to teach about

◦ See reference [32], FabLab paper, and reference [13], project website.

□ See reference [49], LuTec Internal Notes.

time zones, and how to add and subtract time. In the context of a game, a child would choose a card that had a question on it. The question would ask them what time they would need to leave Costa Rica on a plane if they were traveling to a given country and needed to arrive by a specific local time. When given the flight time, they would need to count time zones to determine the desired departure time. They would enter their "answer" by moving their hands in front of the analog clock face as explained above. For this activity, the digital displays were used to keep track of the player's score.

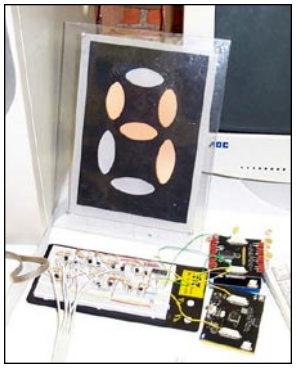


Figure 5.27 - A light-up digit created with FabLab equipment, as part of the Clock that Teaches project.

The students used the FabLab equipment to create the digital displays for their clock. Extremely resourceful, they decided to use deodorant container caps to form the light boxes for the segments in the display, placing two Christmas lights in each segment *(Figure 5.27)*. After designing the physical framework that would hold the segments, they used the milling machine to cut the frames for each digit out of acrylic. The entire clock was controlled by a Tower, using sensor layers to handle inputs and with the lights themselves being driven directly by Tower pins through connections made on a prototyping layer. This project serves as an excellent example of how the Tower system, when used in conjunction with powerful fabrication tools, can enable students to quickly realize ideas that they have been developing for some time.

5.5.3. Thinking About Buoyancy

Figure 5.28 - A student programming a Tower as part of the buoancy exhibit under construction.

For another project, students designed and built a science museum exhibit to teach children about the concept of buoyancy[◦] *(Figure 5.28)*. The exhibit presents visitors with a physical representation of a boat floating on the water. Below the display, are a number of different "weights," each of which with a corresponding holder. As different amounts of weight added and subtracted, the little boat moves up and down in the water accordingly. To implement this project, the students used a Tower with different sensors connect to each "weight" holder. As the sensors were triggered, the Tower would perform the calculations and change the position of the little boat.

Overall, these projects have received a great deal of attention from members of the Ezperanza community, who are encouraged to see the students creating projects which will provide long-term educational benefits to their communities. In addition to using the Tower for their own projects, the students are sharing their growing knowledge of the system

◦ See reference [49], LuTec Internal Notes.

by running electronics workshops with local high school students three afternoons a week. As part of their learning process, the high school students were also encouraged to help their mentors with the projects already underway. With current users actively bringing new members into the Tower community, strong steps are being taken towards fostering a level of technological independence among them all. This trend will continue to spread the spirit of invention and innovation inspired by the new tools we have worked so hard to create.

In Costa Rica, the Tower system has found a home with those wishing to use it in ways that will benefit individuals and communities in the developing world, and especially in their own countries. Students have explored the extension of the system through the FabLab infrastructure we have helped to put in place. The Tower is still being regularly used in this environment as the reach of the LuTec project continues to extend. Each term, more students are introduced to the Tower system and its unique approach to encouraging exploration of the project design process.

5.6. Mexico / INAOE

Based upon the success of our work in Costa Rica, we were recently invited by one of our research sponsors, Telmex, to run an intense five-day Tower workshop in Puebla, Mexico. The workshop was conducted at the Instituto Nacional de Astrofízica, Óptica, y Electrónica◦ (INAOE), with scientists and educators from Mexico (Telmex and others), Brazil (Bradesco Foundation), Panama (SENACYT), and Costa Rica (INCAE). The goal of the workshop was to help the participants explore the ways in which our technology could be used to promote education and community development.

We developed the workshop in conjunction with members of the Institute, as well as representatives from the local government. The participants were carefully selected, representing diverse backgrounds ranging from engineering to education to pure science. The projects they created with the Tower represented a broad spectrum of educational activities,\ and it was amazing to see how quickly complex ideas were realized in the form of working models.

Figure 5.29 - A TableTop Greenhouse, used to teach about environmental and agricultural interactions.

5.6.1. Tabletop Greenhouse

One project built in the workshop was a tabletop greenhouse *(Figure 5.29)*

◦ See reference [20], Instituto Nacional de Astrofízica, Óptica, y Electrónica website.

that integrated environmental sensing with actuation and visual output. The greenhouse was created to represent a model of existing life-sized ones, but was just a few feet long and about a foot and a half tall. It was completely monitored with humidity, light, and temperature sensors, and could be programmed to regulate water flow, shield sunlight, and increase airflow in response to the sensor readings. In addition to the basic operation, the Tower controlling the greenhouse used multicolored LED and display modules to provide easily-readable visual output of system parameters. The idea was that a greenhouse model like this could be used in a classroom setting to teach about environmental factors and the ways in which they influence the growth and health of ecosystems. Additionally, the project was designed to teach children the concept of automating repetitive tasks within the agriculture industry. Automation is a longer-term goal that has the potential to promote sustained economic growth within rural communities.

5.6.2. Livestock Feeding System

Figure 5.30 - The Livestock Feeding System, using an automated funnel to dispense grain to a number of simulated animal corrals.

Another project designed in the workshop was a model of a system that monitors the eating habits of livestock on a farm *(Figure 5.30)*, tracking the amount of food that each animal eats and dispensing more as needed. At the center of the model were four separate bins, each representing a corral for a different type of livestock. Above every bin was a funnel acting as a food supply,\ with a servo-controlled valve regulating the flow, and a sensor at its base to indicate when it was empty. Inside of each bin was a touch sensor that when triggered would indicate that the animals were "eating." As the animals continued to eat, valves above the bins would open to allow more food to flow down into the bins.

The Tower driving the whole system would also sense when the dispensing funnel itself had emptied and would initiate a refill procedure. To refill the funnels, a mobile carriage containing a larger bin would drive to positions above the funnels and refill them with grain, again using servo-controlled valves to regulate the flow. When the larger bin on the mobile carriage was in need of more grain, it would travel to a position underneath a dispensing unit and automatically be refilled. As this whole process was taking place, the main Tower program would record how much each animal had been eating and at the touch of a button, a bar-graph could be displayed on the Tower's screen indicating the relative amounts of food consumed by the different livestock.

The creators of this model envisioned children using it to understand how animals are fed and kept track of on a farm. By doing so, the children would gain valuable insight into how complex systems operate, all the while learning about an industry that is the core of their local economy. Similar to the greenhouse, this project promoted the importance of automation for increasing economic productivity. Its creators hoped that it could also serve as a prototype for a large-scale version for use on an actual farm.

5.6.3. A Reconfigurable Physics Experiment

Inspired by more traditional hands-on learning activities, one group created a reconfigurable physics experiment *(Figure 5.31)* that would allow students to design and study their own experiments about the velocities of moving objects on ramps of different steepness to learn about proportions and ratios. The initial setup was simple: a ramp with a servo-controlled starting gate at the top and a touch sensor at the bottom. To adjust the angle of the ramp, the user rotated a dial that told the Tower program to raise and lower the starting point by turning a second servo attached to the ramp via a control rod. After setting the ramp angle, the ball was released by the push of a button. Data points were taken for the ball's position over time, allowing for velocity values to be dynamically calculated. The Tower's display layer would plot the final data, allowing students to perform a series of experiments and then compare their results to learn about the relationships between speed and inclination angle. Ideally, an activity such as this would provide a starting point for classroom discussions and would eventually lead to more complex simulations to explore mathematical relationships between different physical properties of systems. One of the individuals working on this project is professor of education and he was hoping that experiments like this one would encourage his students to explore new approaches for creatively using technology in their future classrooms.

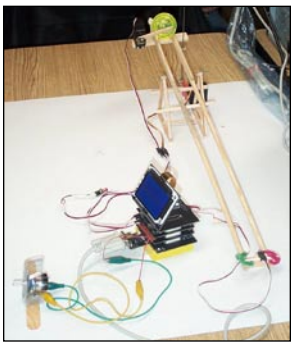


Figure 5.31 - A Reconfigurable Physics experiment, encouraging students to explore the mathematical relationships between different physical quantities

5.6.4. 3D Scanner

One of the most inspiring projects created in this workshop was a simple, low-cost 3D scanner *(Figure 5.32)*. Alberto Muñoz, a professor of robotics, set out to design a project that would help people better understand how objects in three-dimensional space are represented in two dimensions. By building something similar to commercially available arm-style scanners, he created his very own input device for capturing information

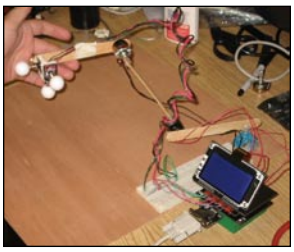


Figure 5.32 - A 3D scanner, using an arm structure with sensors at each joint to determine its exact position in the physical world, for mapping points and motion in a corresponding computer model.

about an object's structure and reporting the collected data back to his modeling software. After experimenting with using the Tower to read sensor values, as Alberto glued together a network of wooden craft sticks and sensor dials to form the overall structure. A different dial would rotate as each branch of the arm would move, allowing the Tower to calculate the resulting x, y, and z, data points for the location at the tip of the arm. Additionally, Alberto put a set of buttons at the tip, so that as points of the target object were touched, their locations would be captured and stored.

To visualize this data in real-time, Alberto first plotted moving dots on the Tower's display to reflect the current position of the arm. A few days later, he wrote a program for his computer that would take the data points being sent across the serial line and create a virtual model of the object being scanned. As his project evolved, he was joined by Daniel Kornhauser, another workshop participant (and former MIT Media Lab student), and the project began to shift in an entirely different direction than Alberto had originally intended. He ended up removing the buttons at the tip of the arm and instead of using it to scan, used it to manipulate object on his computer screen. Objects could be moved around and rotated in three-dimensional space simply by moving the arm around. Essentially, he had created his very own 3D mouse for manipulating objects in a virtual world, a feat just as commendable as creating the original scanner. In fact, the same physical structure was used for both applications and serves to show how quickly and easily major application changes can arise when the hardware and software tools being used are easily reconfigurable.

Alberto's experience was a remarkable one and showed how a system such as the Tower allows powerful ideas to rapidly evolve, allowing projects to change direction as fast as the thoughts of those creating them. With a more rigid system architecture, or in a case where proprietary electronics had to be designed specifically for a given application, users would be more tied to their original idea, unable to make significant changes or completely rethink their solution.

Collectively, the projects created in the Puebla workshop represent a powerful subset of the many things possible with the Tower. The system worked well for those who just wanted to use it in its simplest form while encouraging the curiosity of those who preferred to "dig around under the hood," tweaking the low-level software and connecting things in a less-conventional manner. With the emphasis of the workshop focused on ways in which our technology could be used to promote education and

community development, it was gratifying to see how well the participants themselves adapted and built upon the abilities of the Tower system in order to design powerful new learning experiences which have the ability to directly benefit their own students.

5.7. India / Vigyan Ashram

In addition to our structured Tower workshops around the world, the system has been used successfully to meet specific technical challenges that have arisen in various contexts related to ongoing work at the MIT Media Lab. Recently, as part of the Media Lab Asia / Center for Bits and Atoms outreach initiative, Professor Isaac Chuang accompanied by Amy Sun of Lockheed Martin travelled to Vigyan Ashram, a small educational setting just outside the village of Pabal, in Maharashtra, India[◦]. There were two primary goals to the trip: to teach an introductory electronics workshop to local villagers, and also to show them how they can use technology such as the Tower system to solve engineering challenges that arise in the context of their lives.

5.7.1. Electronics Workshop

In the electronics workshop that was conducted, there was a markedly different focus from the ones that we had personally conducted previously. While our workshops were focused on teaching the design process itself, this workshop was centered on teaching electronics at a much lower level. In a situation like this, the language that is used to teach has to change. Before you can talk about sensors and motors, you must talk about currents and voltages. The basic concepts of how electricity works must be introduced first and foremost.

But one of the most powerful things about the Tower system is that it works just as beautifully on this low level as it does on much higher ones. In fact, the LogoChip project was created to specifically meet needs such as this one. Ideally, workshop participants would begin by learning how to program the chips to perform simple functions. Since full Towers are not needed in a case such as this one, the students were just given foundations with prototyping layers. Even without using the more sophisticated Tower layers we have developed, there are still a wealth of opportunities available with just the basic hardware configuration. In addition to a small area for soldering electronic components, the prototyping layer has tappoints for every I/O pin from the main processor on the foundation, as

◦ See reference [5], Vigyan Ashram Trip Report internal document.

well as locations for connecting power and ground. This makes it very easy for anyone to connect a few LEDs or simple sensors directly to the processor pins themselves.

Figure 5.33 - Two Towers with LEDs connected directly to prototyping layers, made by students in the workshop as part of an introductory exercise to processor programming.

One can easily program the foundation to make LEDs flash by turning processor pins on and off, perhaps in response to a button being pressed that is connected to another pin. In this simple manner, students can be taught basic electrical concepts like voltage and current. In this workshop, students were first introduced to controlling LEDs and small piezo beepers with the Tower. They were shown how to make the LEDs turn on and off in response to sensor inputs *(Figure 5.33)* and how they could make sound with the beepers by just turning a pin on and off very quickly. By experimenting, they quickly found that changing the speed at which they toggled the pin produced different tones.

Figure 5.34 - A student in the India workshop writing a simple Tower program to experiment with the capabilities of the system.

As the students proceeded to experiment with controlling simple electronics parts connected directly to the Tower, one group continued to focus on the beeper they were given. Without the Tower virtual machine in place, the students would have been faced with difficult assembly-level programming from the outset. However, with the Logo program structure it is extraordinarily simple for anyone to begin programming microcontrollers with very little difficulty *(Figure 5.34)*. While all of the low-level functions for bit and pin manipulation are represented in the virtual machine, higher level program control functions are also present, such as loops and if-statements, making it much easier for beginners to develop complex projects quickly and easily.

After writing a simple Logo function that would play a desired note on the beeper, they then wrote a program that played a series of notes to create a song. While they were working, one of the local professors kept receiving calls on his cellular phone. His phone had a very unique ring style to it and without his knowledge, the students wrote a program that would play a tune on their beeper identical to that on his phone. Every time they pressed a button, the tone would play and the professor would come running over, thinking that it was his phone ringing. The students were quite proud of their accomplishments and through this simple exercise of using the Tower system to learn about electronics, had gained a valuable starting point for their continued playful exploration of the field.

While programming in Logo is great for those students who wish to focus on higher-level application design, those who would rather delve more

deeply into the more technical levels of the system are fully capable of doing so by programming the Tower in assembly language instead. The Tower Development Environment contains a full assembler and downloader, everything a user needs to learn how to program a microcontroller in assembly. No expensive chip programmers are needed; a small program preloaded onto the processors allows assembly code to be downloaded simply through the standard serial connection.

Often, programming in assembly language becomes necessary as applications get more demanding in terms of processing speed and the required complexity of operations. In fact, one particular application where assembly programming became necessary was when we were showing local villagers how they could use the Tower technology to solve real-world engineering problems that would significantly benefit their local economy. In fact, this project was a motivating factor for the India trip itself.

5.7.2. Diesel Engine Meter

As previously discussed in Chapter 4, the economy of this particular village is supported by a small industry producing diesel-powered engines, which are used to drive motorized carts and provide electrical power to the entire community. The engines themselves are crudely built and lack precision, leading to very poor fuel efficiency. It is estimated that almost 60% of fuel is wasted due to poor engine tuning. We set out to find a way in which they could use the Tower system in order to improve their engine design process, and by doing so, increase engine efficiency to reduce fuel costs and strengthen their economy as a whole[◦].

Ideally, the engine designers would be able to build a simple monitoring system with the Tower that would allow them to measure the rotational velocity of the engine flywheel and plot the data on a computer to observe and evaluate non-idealities in engine performance. While precise implementation details are discussed earlier in this document, the overall idea was quite simple. Copper strips were attached around the edge of the flywheel, which would interrupt a reflectance sensor beam as they rotated through it *(Figure 5.35)*. The optical sensor and two LEDs were connected directly to the foundation processor through the connection points on a Tower prototyping layer.

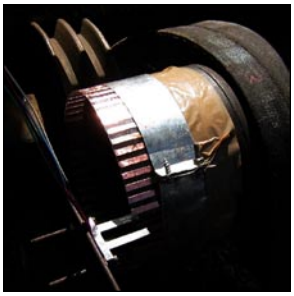


Figure 5.35 - The engine flywheel with copper strips attached for interrupting an optical sensor.

With the engine flywheel spinning in excess of 2000 rpm and the designers requiring precise measurements taken about 50 times per rotation, it

◦ See reference [3], Engine Tune-Up Project internal document.

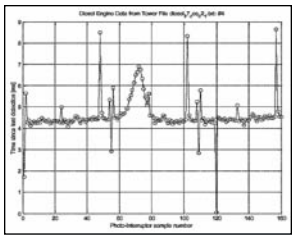
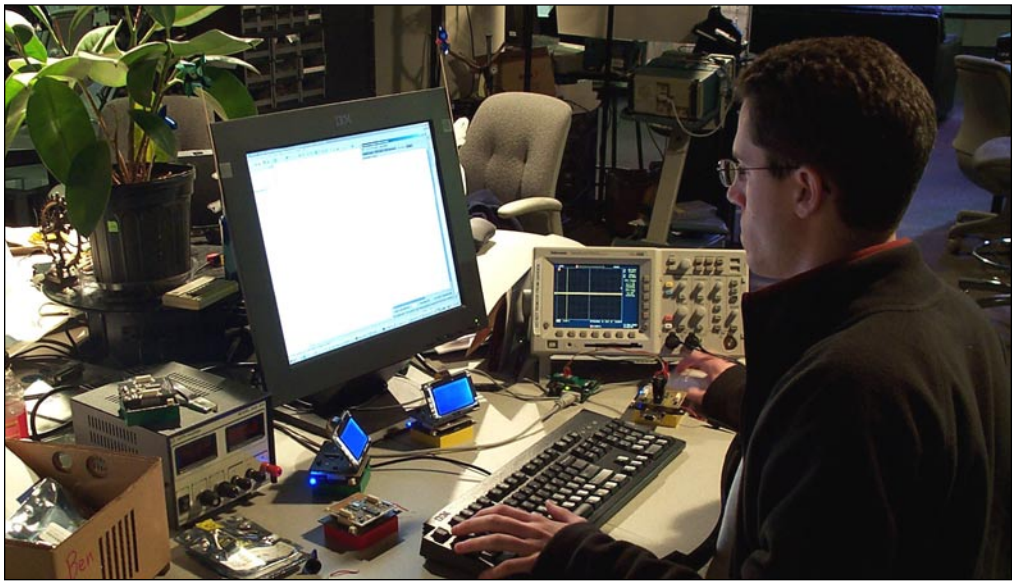


Figure 5.36 - A plot of the time between sensor acquisitons, clearly showing a point in the cycle where the engine slows down significantly, most likely due to a misfiring in one of the engine cylinders.

was necessary to write the main program in assembly language. To further boost speed by reducing the overhead necessary to send the data back to the computer for plotting *(Figure 5.36)*, data was buffered at high speed in large blocks. When the buffer filled up, the data block would be sent to the host computer in its entirety, at which point the program would begin taking data again. Simple on-board indicator LEDs were used to observe operation. A green LED would turn on and off as the copper strips were encountered and a red LED would switch state every time the buffer overflowed and data was sent back to the computer.

Together, we arrived at a relatively simple solution to an important, real-world problem. While this particular application used the Tower in its most raw form by building simple circuits on a prototyping layer, the layer itself could easily be removed and others put in its place to open the door for more complex applications to be built using Logo or another high-level programming language. When resources are scarce, having a system that is highly reconfigurable and reusable, yet still powerful on many levels proves immensely useful to the local population.

It is clear that our extensive outreach initiative has laid a strong foundation for a core Tower userbase that is continuing to grow day by day. As even more people start to realize the true power of the system, we are looking forward to seeing how it will be applied it to the various technical and design challenges facing them in the context of their own lives.



A member of our research group works to create new layers for the Tower system in Cambridge, MA.

Chapter 6 - Technical Detail

At every technical level, the Tower system *(Figure 6.1)* represents a fully modular system architecture. In addition to the inherent modularity of the structural and functional hardware elements that snap together, the firmware and software implementations are designed to be modular as well. All levels of the system architecture are open for users to modify if they wish, and industry-standard protocols are employed wherever possible.

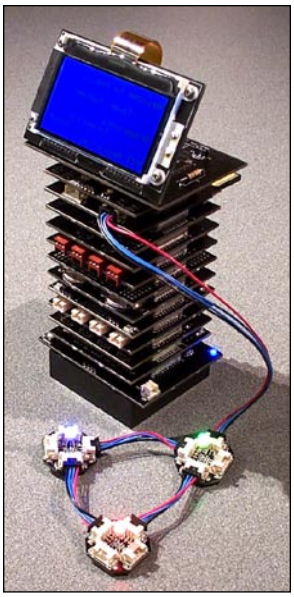


Figure 6.1 - The Tower system, a fully modular computational construction kit.

While technical details are discussed in the following sections, it is important to notice the outlined distinctions between deployed portions of the system and those currently under development. The system itself is by no means a finished product and it will continue to grow as long as a strong userbase exists. The work outlined in the "continuing development" sections that follow constitute work currently underway within our direct research group, as supervised by the core Tower development team.

6.1. Hardware Design

The modular nature of the Tower system hardware is due to the fact that every I/O line from the main processor on the foundation is passed up

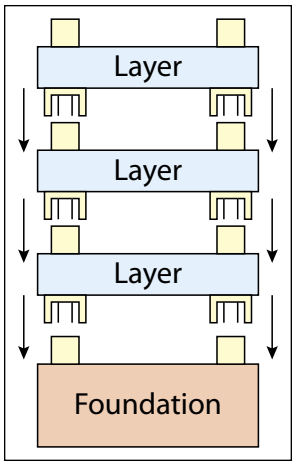


Figure 6.2 - A diagram of the physical structure of the Tower system. In addition to providing physical support, the connectors provide the electrical datapaths between modules.

through the entire stack to every connected layer. In addition to providing electrical connections for all of the data lines, the connectors serve a structural purpose as well. With male connectors attached to the bottom of each board, and female connectors on top, the boards can be easily stacked *(Figure 6.2)*.

While various foundations are available with different processors onboard, every Tower layer has its own independent Microchip◦ PIC™ microcontroller. Communication between the processors on the foundation and those on the layers, is accomplished via the industry-standard I^2C (inter-integrated circuit) serial protocol. This two-wire protocol contains separate clock and data lines, which utilize two of the data I/O pins that are already being passed up the stack. To talk to the foundation, each layer has its I^2C communication pins wired to those specified on the main connectors.

Power for the layers can be drawn from either a primary or secondary power bus, both of which are passed up the stack through the main connectors. Modules that anticipate higher power drain such as the DC Motor and CompactFlash layers are designed with on-board switches to allow the user to decide from which bus they will receive their power. On all of the boards with power-select switches the microcontroller is always powered off of the primary bus, ensuring that all of the digital circuitry in the system remains on the same power supply. This power configuration protects the logic circuitry from power dips and also allows for higher voltages to be applied to the secondary bus, enabling motors and other high-powered devices to be driven at different voltage levels if desired.

Figure 6.3 - The Tower Support Forum website, where users around the world can share ideas and system extensions they have created.

6.1.1. Deployed Hardware

There are two foundations and fourteen layers available in the current official distribution. Any user around the world is capable of requesting available layers through our online Tower Support Forum□ *(Figure 6.3)*. Thanks to extensive world-wide-web database work by Tim Gorton (also part of the core development team), we have a full online tracking system implemented, showing us exactly which Tower modules are in the possession of every user.

All of the currently available modules were created by us, in direct response to our needs as they arose. Starting from a core set of what we felt was absolutely necessary such as sensors, motors and memory, more foun-

◦ See reference [30], Microchip corporate website.

□ See reference [47], Tower Support Forum website.

dations and layers were gradually added to the system, being released to the user community as development stabilized. Listed below are detailed technical descriptions of the functionality of every module in the system that is currently available.

6.1.1.1. Currently Available Foundations

The PIC foundation *(Figure 6.4)* is built around a Microchip PIC16F877 microcontroller, with 8 kilobytes of program memory, 256 bytes of data EEPROM, and 368 bytes of RAM. The processor is clocked at 8 MHz, and all 33 I/O pins are broken out to the rest of the stack via two 18-pin connectors. Full documentation for the PIC foundation can be found in Appendix E.

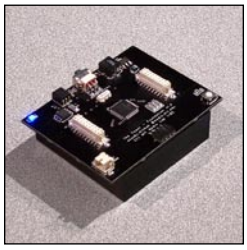


Figure 6.4 - The PIC foundation.

The Rabbit foundation *(Figure 6.5)* uses a Rabbit 2300 Ethernet-enabled microcontroller from Rabbit Semiconductor[◦] running at 22.1 MHz. The Rabbit processor has 128 kilobytes of SRAM, 256 kilobytes of program memory, and 29 I/O pins. However, with the Ethernet port in use, only 13 of the I/O pins are passed up the stack to the rest of the Tower layers by connectors identical to those on every other foundation. The Rabbit Foundation was developed by Tim Gorton, a member of the core Tower development team. Full documentation for the Rabbit foundation can be found in Appendix F.

Figure 6.5 - The Rabbit foundation.

Both foundations have built in power circuitry and can be powered from either 4 AA batteries or a 1mm DC power jack, which is regulated down to 5V. Additionally, there is a connector present on the foundations for the secondary power bus that allows external power supplies or additional batteries to be plugged in.

Any foundation can be easily programmed via a 4-pin serial programming header. Using one of our serial-line driver boards, a standard serial cable can easily be connected from a host computer to the Tower for programming and run-time communication.

6.1.1.2. Currently Available Layers

Each additional layer has a PIC16F876 processor on it, which communicates with the foundation via the I^2C serial protocol at a speed of 400 Kbps. We are currently utilizing a 7-bit I^2C addressing scheme, allowing for 128 distinct layers to be connected to the Tower simultaneously. Each

◦ See reference [37], Rabbit Semiconductor corporate website.

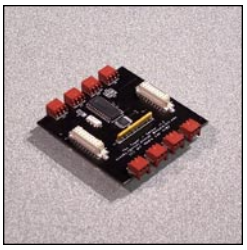


Figure 6.6 - The Sensor layer.

Figure 6.7 - The DC Motor layer.

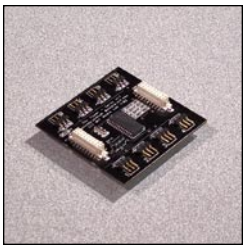


Figure 6.8 - The Servo Motor layer.

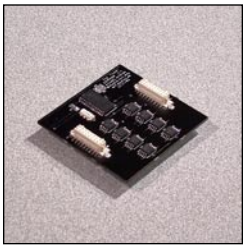


Figure 6.9 - The EEPROM layer.

Figure 6.10 - The CompactFlash layer.

layer has a default I^2C slave address, but addresses can be easily reassigned dynamically during run-time operation.

The Sensor layer *(Figure 6.6)* provides eight high-precision 12-bit A/D conversion ports. Each port has a 3-pin connector, with pins for the direct analog input voltage, ground, and power. Primarily geared towards resistive sensor measurements, there is a 33 kilo-ohm pull-up resistor on the analog input pin, so resistive sensors should be bridged between that pin and the ground connection. However, if a buffered sensor voltage is available, the analog input can be tapped into directly for single-ended voltage readings. Full documentation for the Sensor layer can be found in Appendix G.

The DC Motor layer *(Figure 6.7)* supports simultaneous independent pulse-width-modulation control over four DC motors. Each motor is driven by a separate full H-Bridge circuit and can be controlled at 256 different speed levels in either direction. Additionally, velocity ramps are supported so that motor acceleration can be finely controlled. Full documentation for the DC Motor layer can be found in Appendix H.

The Servo Motor layer *(Figure 6.8)* can be used to drive up to eight independent standard hobby servos. The layer is calibrated to drive servo motors over a range of 100 degrees, with a resolution of 1 degree as controlled by a 1ms-2ms varying pulse width. While the servo motors themselves are capable of moving over 180 degrees, the reduced motion range is to allow for neutral-position variations in servo motors between manufacturers to prevent grinding due to different stop positions. Full documentation for the Servo Motor layer can be found in Appendix I.

The EEPROM layer *(Figure 6.9)* contains a bank of eight 256-kilobit serial-EEPROM chips for a total of 256 kilobytes of user read-writeable non-volatile memory. The memory chips themselves are controlled by the on-board PIC to mitigate possible I^2C address conflicts with other devices on the Tower stack. Full documentation for the EEPROM layer can be found in Appendix J.

The CompactFlash layer *(Figure 6.10)* supports a reduced-IDE data interface to CompactFlash I & II cards, as well as Microdrive™ units. Currently, only direct sector read/write access is available, but Jeremy Walker, a member of our research group, is currently working to implement a full on-board FAT32 filesystem which will allow complete filesystem compat-

ibility between the Tower and personal computers. The hardware for the CompactFlash layer was created by Glenn Tournier, an undergraduate working in our research group. Full documentation for the CompactFlash layer can be found in Appendix K.

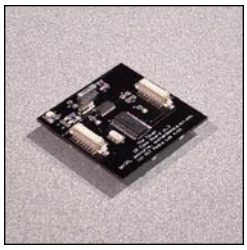


Figure 6.11 - The IR layer.

The IR layer *(Figure 6.11)* allows for wireless multi-tower data transfer via infrared communication at 19.2 Kbps. Since the layer uses standard IRDA-compliant hardware protocols, it is simple to program handheld computers to communicate with the Tower via IR. The hardware for the IR layer was created by Glenn Tournier and Austin McNurlen, two undergraduates working in our research group. Full documentation for the IR layer can be found in Appendix L.

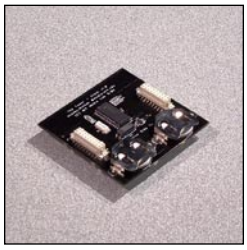


Figure 6.12 - The Clock layer.

The Clock layer *(Figure 6.12)* provides a real-time clock with a battery backup for use in applications requiring accurate time and date information. The clock's registers are user-settable with a timing resolution of 1 second. Time data can be read from and programmed to the clock chip either as a full data stream or by a single parameter at a time. Full Clock layer documentation can be found in Appendix M.

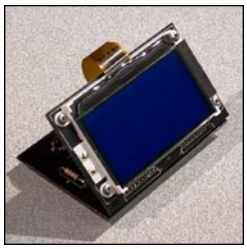


Figure 6.13 - The Display layer.

The Display layer *(Figure 6.13)* uses a prebuilt LCD display module with a chip-on-glass controller. The display is capable of graphical resolutions up to 128x64 pixels, and is available in either black-on-yellow or white-on-blue color schemes. There is a user-controllable backlight that can be driven off of either power bus. Text and graphics modes are both supported by the firmware on the layer, with line and sprite draw functions available as well as string and character commands. Due to the nature of the layer's structure, it must be located on the top of the Tower since no connectors are present on its upper surface. Full documentation for the Display layer can be found in Appendix N.

Figure 6.14 - The Cricket Bus layer.

The Cricket Bus layer *(Figure 6.14)* provides proprietary single-wire bus connectivity with a wide variety of devices created as part of the MIT Media Lab's Cricket[◦] development system. In addition to enabling the use of the many existing input and output devices that have already been created, the Tower can also be used to prototype new devices for use with the Cricket system. For a specific project, the Tower itself was used as an "ultimate bus device," providing the functionality of every layer in the Tower system through a single bus connection point. An interesting point is that the LogoChip[□] project, from which the Tower firmware evolved, was originally begun with the intent of simplifying the creation of new Cricket

◦ See reference [29], Metacricket paper, and reference [8], Cricket system website.

□ A more extensive explanation of this cross-development is presented in section 6.4 of this document. See reference [26], LogoChip project website.

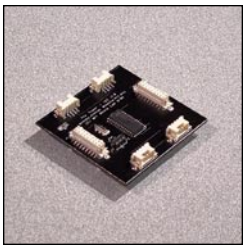


Figure 6.15 - The I²C layer.

Figure 6.16 - The Tricolor layer.

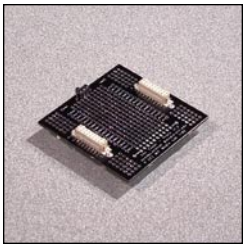


Figure 6.17 - The Proto layer.

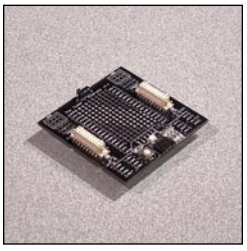


Figure 6.18 - The PICProto layer.

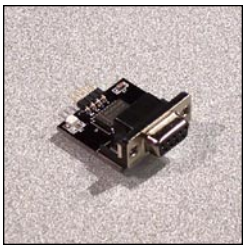


Figure 6.19 - The RS-232 module.

bus devices. While developmental paths have taken the Tower project in its own distinct direction, the Cricket Bus layer allows the Tower to easily meet the original challenges that prompted its firmware evolution. Full documentation for the Cricket Bus layer can be found in Appendix O.

The I²C layer *(Figure 6.15)* is a breakout board for the I²C serial bus connection running between Tower layers. By providing four cable connection points for I²C bus cables, this layer provides the connectivity needed to attach off-stack layers to the Tower such as the Tricolor layers, and the ability for multiple Towers to communicate with each other over a cable connection. Full documentation for the I²C layer can be found in Appendix P.

The Tricolor layer *(Figure 6.16)* is an off-stack Tower layer, connected to the rest of the Tower via an I²C serial bus cable. The layer contains a high-performance full-spectrum LED, which can be set to any RGB value. Additionally, fading between color values is can be performed, with fade times supported at resolution of one hundredth of a second. Full documentation for the Tricolor layer can be found in Appendix Q.

The Proto layer *(Figure 6.167* provides a blank prototyping area with direct tap-points to every I/O pin passed up the stack from the foundation. The Proto layer is often used for connecting simple parts such as buttons and LEDs to the Tower. When the system is being used to teach lower-level electronics design, the layer is extremely useful for building up small circuits and connecting them directly into the Tower architecture. Full documentation for the Proto layer can be found in Appendix R.

The PICProto layer *(Figure 6.18)* is similar to the Proto layer, but in addition to providing prototyping space has an on-board PIC16F876 processor and full support circuitry. This layer allows users to design and prototype entirely new layers for the system using the underlying multiprocessor communication structure. This on-board PIC is running the same virtual machine as the foundations, so anyone can easily program it in Logo to add significant new functionality to the Tower system. After being designed and tested on the PICProto layer, new system extensions can easily be fabricated on custom circuit boards and made available to entire Tower user community. Full documentation for the PICProto layer can be found in Appendix S.

The RS-232 module *(Figure 6.19)* is not technically a layer, but it is a critical

part of the system. The module provides the necessary line-level shifting for direct communication between a host computer and any foundation or layer with a 4-pin serial connector on board. The module uses line-driver chip to perform the conversion, and has an on-board bicolor LED to indicate transmission and reception status. Full documentation for the RS-232 module can be found in Appendix T.

6.1.2. Continuing Hardware Development

As applications both within and outside of our research group continue to demand new and higher-end computational modules, we seek to develop the ones that we feel have the potential to reach the broadest portion of our userbase. Two new foundations and twelve layers are currently being designed and tested, which when complete will become available for any user to request. Detailed technical descriptions of the modules that will soon become available are listed below, along with comments regarding their current developmental status.

6.1.2.1. Foundations Under Development

The 8051 foundation *(Figure 6.20)* uses a Cygnal° 8051 Series-124 processor, with 8 kilobytes of stack memory, as well as 64 kilobytes of storage RAM on an external chip that is dynamically allocated for program usage. The 8051 foundation runs at 50 MHz, and when available, will be one of the most powerful foundations in the system. This board is currently functioning, and in the final stages of development. While most of the virtual machine port is currently functional, support for the on-board RAM banks has not yet been implemented. As soon as the foundation is fully tested, it will enter final production. The 8051 foundation is being developed by Ali Mashtizadeh, an undergraduate member of our research group.

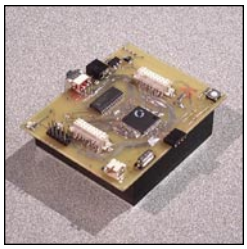


Figure 6.20 - The 8051 foundation.

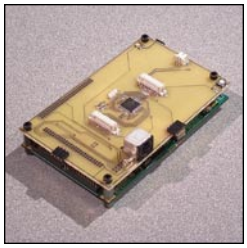


Figure 6.21 - The Bitsy foundation.

The Bitsy Foundation *(Figure 6.21)* is built around the commercially available ADS□ Bitsy module using a StrongARM™ SA-1110 RISC processor running at 206 MHz. The Bitsy module is essentially the core of a handheld computer, and is capable of running either Linux or Microsoft◊ Windows CE™. The module contains a PCMCIA slot, giving it the built-in ability to use 802.11b wireless communication. This foundation provides a powerful option for those more comfortable developing projects on a standard "computer". With a foundation daughter-board on top of the preexisting module, users are free to program in whatever language they

° See reference [9], Cygnal corporate website.

□ See reference [2], ADS corporate website.

◊ See reference [31], Microsoft corporate website.

choose. By means of an on-board serial-port, users can communicate directly with any layer on the Tower, just as with the other foundations on the system. For those preferring to use Logo, we are currently in the process of adapting our virtual machine to run on this new processor, but with support for the higher-level hardware functionality available. This board is in the early stages of development, but an initial prototype currently possesses most of the desired functionality. The Bitsy foundation is being developed by Ali Mashtizadeh, an undergraduate member of our research group.

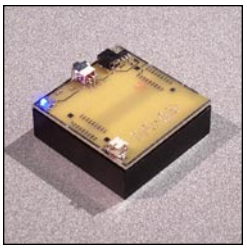


Figure 6.22 - The Power module.

6.1.2.2. Layers Under Development

The Power module *(Figure 6.22)* allows for the connection of additional batteries or power sources to the Tower's secondary power bus. The module can be attached via a cable to any foundation in the system, providing more power or higher voltages as needed by some layers. The Power module is currently in the final stages of development, with a fully-functional prototype awaiting production.

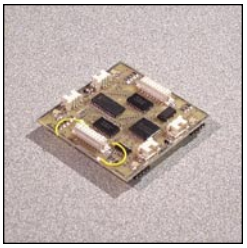


Figure 6.23 - The UART layer.

The UART layer *(Figure 6.23)* provides four logic-level UART communication ports which can be used to connect multiple Towers together, as well as to connect them to personal computers acting as nodes via the RS-232 module discussed above. Each port has a 64-byte hardware buffer and a small tricolor LED to indicate transmission status. This board is currently in the final stages of development. It is functional but awaiting resolution of a bug causing the board to drain excessive amounts of power. The UART layer is being developed by Tim Gorton, a member of the core Tower development team.

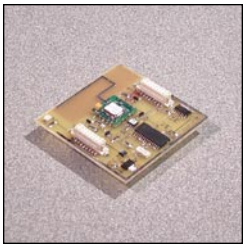


Figure 6.24 - The RF layer.

The RF layer *(Figure 6.24)* is built around a commercially available radio-frequency transceiver module. The layer operates on the free band of 916.5 MHz at transmission speeds of 115.2 Kbps, and is able to communicate with devices at ranges of up to 300 feet away in open spaces. This layer is in the final stages of development, with single-byte communication functioning and packet-level communication currently being implemented. The RF layer is being developed by Benjamin Walker, an undergraduate member of our research group.

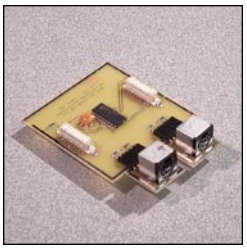


Figure 6.25 - The PS/2 layer.

The PS/2 layer *(Figure 6.25)* provides connections for two PS/2 devices, typically a keyboard and a mouse. The device interface is software driven, with data buffering allowing for the simultaneous use of both devices. At

this point, the hardware is fully operational and the firmware is properly processing keyboard input, but mouse support is not yet enabled. As soon as the firmware is updated and tested, the board will be ready for final production.

The PICProg layer *(Figure 6.26)* is a low-level hardware programmer for downloading assembly code to PIC microcontrollers. The layer itself is quite simple, using a two-transistor setup to drive the high-voltage programming pin, with the other pins of the target processor being controlled directly by the PIC on-board the layer. Programming can be performed either by sending assembled data through the Foundation and up the stack or by directly connecting the host computer's serial line to the layer itself and allowing the Tower Development Environment to handle the actual programming procedure. This layer is in a moderate stage of development, built-up and working on a PICProto layer, but awaiting final board layout and testing.

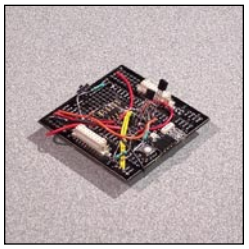


Figure 6.26 - The PICProg layer.

The LED Array layer *(Figure 6.27)* uses an 8x8 bicolor LED grid to allow for the visual representation of interesting user-specified patterns. With a network of serial-shift-buffers driving the columns of the array, any pattern can be easily displayed or animated over a period of time. Due to the structural nature of the array itself, this layer must be located on the top of the Tower, as no connectors are present on its upper surface. An on-board LogoChip allows this board to run Logo program code, simplifying the development cycle. This layer is in the early stages of development and an initial prototype has been built. While functional, the LED drive circuitry is being redesigned to allow for brighter display capabilities. The LED Array layer is being developed by Larisa Egloff, an undergraduate member of our research group.

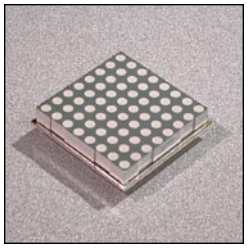


Figure 6.27 - The LED Array layer.

The Alphanumeric layer *(Figure 6.28)* is capable of displaying eight separate text-characters with LEDs. Similar to the Tricolor layer, the Alphanumeric layer is an off-stack layer, connecting to the Tower through I^2C bus cables. Additionally, the layers can be strung together to allow for more extensive text display capabilities. While other options are available for displaying information in an application, this layer represents the option that can most easily be read by many individuals from a distance. The layer code itself is written in Logo and runs on an on-board LogoChip for ease of programming. This layer is in an advanced stage of development, with a mostly-functioning prototype in need of a few minor revisions. The Alphanumeric layer is being developed by Amir Mikhak, a high

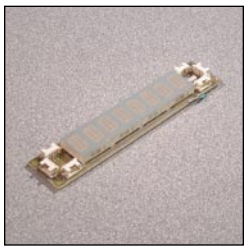


Figure 6.28 - The Alphanumeric layer.

school student working in collaboration with our research group.

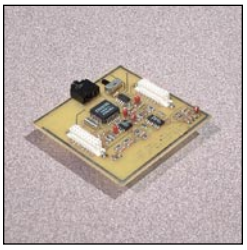


Figure 6.29 - The MIDI layer.

The MIDI layer *(Figure 6.29)* contains a full polyphonic MIDI synthesizer chip on-board, and provides a line-level analog output. All standard MIDI audio commands are supported, including environmental effects. Audio output can be obtained directly from the board itself through a 1/8" audio jack, or passed up the stack to the Audio Mixer layer for combination with the output signals from other audio-producing boards. While initially part of the Tower system in an earlier implementation◦, this board is being entirely redesigned to meet the audio subsystem specifications recently defined. It is in the early stages of development and a prototype is not yet available, but initial functionality has been proven with the original version.

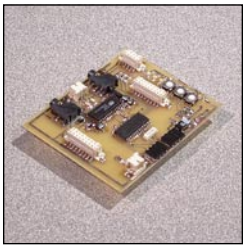


Figure 6.30 - The Voice Recorder layer.

The Voice Recorder layer *(Figure 6.30)* has a small on-board microphone, and records directly into the memory of a single-chip voice recorder. A full eight minutes of audio space is available, which can be divided up into a number of audio samples of varying lengths. Additionally, digital audio data stored on a personal computer can be directly downloaded to the layer for storage and playback. The audio output signal is handled in the same manner as described for the MIDI and Text-to-Speech modules. This layer is currently under development and awaiting testing of the newest revisions of the circuit board layout. As soon as final tests are complete, it will be ready for production. The Voice Recorder layer is being developed by Jonathon Downey, an undergraduate member of our research group.

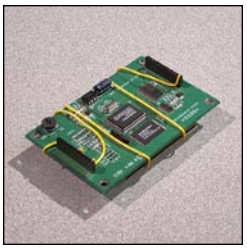


Figure 6.31 - The Text-to-Speech layer, using the manufacturer's development kit connected to a Tower PICProto layer.

The Text-to-Speech layer *(Figure 6.31)* uses a two-chip speech synthesis engine, and provides high-quality simulated voice in response to text input strings. Audio output is handled in the same manner as the MIDI board described above. Initial tests of this layer using a factory-provided development kit were successful, but the layer itself is being reengineered in order to eliminate our dependence on these expensive prefabricated modules. The Text-to-Speech layer is in the early stages of development and a prototype is not yet available, but initial functionality has been proven with the manufacturer's development kit connected directly to a Tower PICProto layer.

The Audio Mixer layer allows for digital control over mixing and equalization operations on the outputs of the other audio layers. All of the connections to this board are routed through the stack on separate con-

◦ Details of the earlier implementation and the reasons for the redesign are discussed in section 6.4 of this document.

nectors, ensuring that the primary Tower data lines are not taken over for audio processing. Line-level stereo or surround-sound output is available, as well as an amplified output for a headphone connection. This board is in the early stages of development, and a prototype is not yet available. The electronic components have been chosen and initial circuit board layout will begin as soon as more of the audio-generating layers described above are completed.

The Capacitive Sensor layer is used to obtain touch and proximity-based sensor readings. The layer has four separate sensor ports, each of which is controlled by a special capacitive sensing chip. Electrodes can be connected directly to each port, and configured in any user-desired manner based on the needs of the application. The layer is in an early stage of development, and although a circuit board has been designed, it has not yet been fabricated. However, basic functionality has been proven with test circuits. The Capacitive Sensor layer is being developed by Amir Mikhak, a high school student working in collaboration with our research group.

6.2. Firmware Design

The Tower firmware is the low-level program code stored directly on each processor in the system. Firmware code essentially falls into two different categories. First, there is the virtual machine, the main program that runs on the foundations, which handles the downloading of Logo code and interprets the code itself at run-time. Secondly, there is the layer code, permanently programmed onto each layer in the system, which handles the multiprocessor communication and performs all of the board-specific functions such as reading sensors values or generating a pulse train to control a servo motor position. Both types of firmware are of critical importance to the overall operation of the system and will be discussed in detail in the following sections.

6.2.1. Adapting and Extending the Virtual Machine

The Tower virtual machine is largely adapted from a virtual machine written primarily by Brian Silverman several years ago for the MIT Media Lab's Cricket development system. In fact, every member of the core Tower development team has previously made significant contributions to the Cricket system, and we were interested in expanding upon the strengths of the virtual machine to help it reach a wider user community.

◦ See reference [26], LogoChip project website.

In the initial migration from the Cricket virtual machine to the LogoChip° one, the focus of the firmware development shifted significantly, tending away from higher-level functions. Functions that used to exist for directly controlling sensors and motors were replaced with lower-level functions for manipulating bits and directly accessing memory locations. In the new implementation, higher-level procedures have been created for sensor and motor control, providing novice users with a more transparent entry point into their operation.

While the LogoChip additions were initially implemented by Brian Silverman, further extensions were needed to support the Tower system architecture. We have made significant additions to the virtual machine that have increased its power, such as adding features for I^2C multiprocessor communication and in-line assembler directives, and enhancing the machine's memory structure to make use of more powerful processors.

In the end, the resulting firmware implementation closely resembles assembly language, but with Logo-like syntax. Through the continued course of our development we have chosen to retain the underlying Logo architecture, because in the past it has proven to be extremely powerful and versatile when it comes to rapidly prototyping complex applications.

This careful intersection of the Logo and assembly language provides a programming environment that is not only simple to use, but simple to teach as well. When first learning assembly programming, perhaps the most difficult concept to convey is how to handle conditional operations. Testing bits and branching to different program locations can be a terrifying thing to someone who has never programmed before, especially in assembly. However, by combining bit-level operations for register manipulation with higher-level control flow primitives such as loops and if statements, we have achieved the best of both worlds.

6.2.2. Multiprocessor Communication Simplified

When building complex embedded electronics applications, multiple processors are often needed to distribute computational power. In the specific case of the Tower, multiple processors also become essential to the emphasis of the modular system nature. Being able to dedicate one processor to each overall function makes the entire implementation simpler for users to understand on the conceptual level as well. While the Cricket system employed a similar processor distribution technique, it

used a proprietary protocol with fixed devices addresses that often proved difficult to extend. For our work, we wanting to focus on establishing a flexible and adaptable communications core, making it easy for anyone to add their own devices to the system without needing to understand the low-level protocols.

Unfortunately, achieving fast and reliable communication between different processors can often prove to be a daunting task. In keeping with the open architecture of our system design, we chose to use the industry-standard I^2C serial protocol, a simple method for using two wires to communicate with a number of processors on a common bus.

However, when processors must also perform more computationally intense operations, implementing the communication protocols to run entirely in the background can prove to be challenging. Over an extended period of time, we implemented and refined a fully interrupt-driven I^2C serial communications firmware package written entirely in assembly. As data is sent between boards, transmit and receive buffers are dynamically allocated in memory and return data is pulled from the layers at the foundation's convenience. Internal flags on each device are used to represent the availability of stable buffer contents and indicate when all data values have been successfully transferred. Additionally, the foundation has full control over address distribution, and can dynamically reassign board identifiers as needed. In the end, these communication routines were organized and packaged in such a way that they can be simply inserted into any assembly program file being written for a foundation or layer under development.

However, a user does not need to program in assembly in order to take advantage of the functionality we have provided. The communication package is just as powerful when used through the Logo programming environment, as it is on the lower levels. Since the I^2C functions are built into both the virtual machine and the layer firmware, Logo primitives are available for direct buffer and flag manipulation. This allows even novice users to write simple programs for the LogoChip or PICProto layer that communicate with the foundation just as every other layer in the system does.

6.2.3. Templates for Extension

Since a long-standing goal in the design of our system architecture was

for the system to be easily extensible, it was important to ensure that the firmware implementations themselves would be as easy to understand and build upon. When it comes to extending firmware, there are primarily two different paths that a user might need to take. First, and most common, would be the development of firmware for new layers that are being created. Secondly, while rarely needed, it is possible for end-users to extend the functionality of the virtual machine itself by adding new low-level primitive definitions as applications require them.

6.2.3.1. Creating New Layers

For the development of new layers, users are faced with multiple options. New firmware can be written in Logo, assembly, C, or any other language that a compiler can be obtained for. For most boards, any choice is a valid option. Logo would be the obvious choice for beginners, but assembly programming becomes necessary for layers that must operate at exceedingly higher speeds.

Writing layer code in Logo is quite simple, requiring very little understanding of the underlying communication principles, but simply the willingness to learn how to use the three or four primitives that make it all possible. The process by which one would write layer code in Logo is extensively documented in Appendix N, which gives detailed examples of how to use the PICProto layer◦. To communicate with the foundation, the user's program simply waits until new I²C data is received, then checks the receive buffer to see what values have been sent. After processing these values and acting accordingly, any result that needs to be sent back to the foundation is stored in the transmit buffer and one final function is called to tell the layer that the data is ready to be sent. In order to make the new layer operate similar to the others, the main program function would typically be bound to the "on-startup" directive, ensuring that the user program starts running when the Tower is turned on, and will be available at any time when the foundation attempts to talk to it.

◦ While every PICProto layer is already running the Logo virtual machine, newly-obtained individual chips will have to first be programmed by a hardware PIC programmer such as the PIC Prog layer, before Logo code can be downloaded.

If the assembly language route is chosen, there is a starting template that includes all of the code necessary to get a basic layer up and running. In fact, the template has a clearly commented point at which user code should begin. At that location in the program, new data has just been received by the layer. It's up to the user's program to parse it, process it, fill the response buffer, and set the flag, at which point the program returns to its background communication processing. While basic knowledge of

assembly programming is necessary in order for a user to take this route, the carefully designed template makes this method rather simple to implement.

When users want to write layer code in languages other than those we directly support, a number of different commercially-available compilers are available. Programming a layer in C is quite simple for those familiar with the language, as our standard communication package can be inserted into the program as in-line assembly code. While we do not currently have a C compiler built into our development environment, members of our research group have successfully used commercial compilers to implement layer firmware.

6.2.3.2. Adding New Primitives

In some cases, users may wish to add new functionality to the virtual machine itself. While the virtual machine already contains primitives for nearly every low-level function we envision users needing, specific cases have arisen where a new addition was needed. In most situations, a new primitive would only be needed when a significant speed boost is required, preventing a function from being written in Logo or another higher-level language. Adding a primitive is essentially accomplished by first writing the function itself, and then placing special commands at its entry and exit points to handle the manipulation of system variables being passed to it. The new primitive must also be added to the definition table present in both the virtual machine program itself, as well as in the software compiler program for the given processor. In both cases, this is as simple as just adding a single line of code.

6.2.4. Virtual Machine Implementation

The virtual machines themselves have a simple stack-based architecture, allowing for easy extension as new primitives and other functionality become needed. This architecture also provides a strong basis for porting the firmware to other processors as new foundations are introduced into the system. The inherent extensibility of the underlying structure is largely thanks to Brian Silverman's original implementation.

In normal operation, Logo program code is downloaded to the virtual machine, which can then interpret and execute it as directed. At runtime, the Logo program code is read out of memory, byte by byte, and the

corresponding primitive functions are evaluated, all while keeping track of the program location and variable states. User-defined procedures supplement the primitives available and are capable of accepting an arbitrary number of inputs and optionally producing an output. Procedures can be initiated automatically at power-on if declared in the "on-startup" directive, or mapped through the "on-white-button" directive to the small white button located on every foundation.

However, users can also communicate with the Tower by direct command-line control through the Tower Development Environment (TDE) software running on a host computer. After downloading program code, the TDE is aware of the precise memory locations of every function it has downloaded, and from there can execute them directly, incorporating user-specified arguments. Additionally, single lines of code can be executed from the "command center" window in the software environment. This approach proves useful for running simple operations as well as debugging more complex routines, and is accomplished by transparently compiling, downloading, and executing a single line of code.

All of our virtual machines are essentially the same in terms of operation, but possess slightly different primitive sets based on available features in the different processors used. Precise implementation details, such as multi-threading, variable storage, and memory management, are handled differently on each processor, relying heavily upon a tight integration with the underlying device architecture.

Currently there are two virtual machine implementations in active use, one for each foundation publicly available. While similar in most ways, their differences are outlined in the comparison chart *(Table 6.1)* shown below:

	PIC VM	**Rabbit VM**
Target platform	PIC16F876 and PIC16F877	RCM2200 and RCM2300
VM implementation language	PIC Assembly	Rabbit C
Processor clock speed	8 MHz or 20 MHz	22.1 MHz
Processor data storage	384 bytes RAM 8 kilobytes EEPROM	128 kilobytes RAM 256 kilobytes EEPROM

Math	16-bit signed integers	32-bit signed integers and floating-point
Timer	32-second timer	24-day timer 128-year clock
Multitasking	2 processes	20 processes
Logo execution speed	~26,000 Logo instructions/sec (8 MHz) ~65,000 Logo instructions/sec (20 MHz)	~8,000 Logo instructions/sec
Variables	96 global variables (16-bit) Procedure arguments	128 global variables (32-bit) Procedure arguments Local variables inside procedures.
I²C bus	Hardware support built-in	Uses external PIC16F876 on foundation
On-board serial ports	One	Four
Other		String manipulation Arrays TCP/IP stack 10 Base-T Ethernet

Table 6.1 - A feature comparison between the PIC and Rabbit virtual machine implementations.

Technical details of the two virtual machines are discussed to a greater depth in the following sections. While the original virtual machine was written by Brian Silverman, other members of the Cricket system development team, including Robbie Berg, Fred Martin, and Bakhtiar Mikhak all made significant contributions. The most recent extensions for its use in the Tower system have been implemented by Tim Gorton and myself. The Rabbit virtual machine port discussed below was implemented almost exclusively by Tim Gorton, but is discussed here for completeness of the system architecture overview.

6.2.4.1. PIC Virtual Machine

The PIC virtual machine is implemented in PIC assembly language. Both the virtual machine itself and the downloaded user code reside in the 8

kilobytes of EEPROM built into the PIC. The PIC's 384 bytes of RAM hold the stack, global variables, and the two buffers for I²C communication.

Operationally, the PIC virtual machine has two process threads- a foreground process and a background daemon. In most user programs the foreground thread handles all of the work, but in some cases the background daemon proves valuable. For example, the background daemon can be used to initiate a periodic activity, or be set to take some action when a specific event occurs.

In addition to providing control flow, math, and multitasking support, our recent extensions also include hardware-specific functions for interacting with on-chip hardware such as the UART, A/D converters, I/O pins, and the I²C communications port.

6.2.4.2. Rabbit Virtual Machine

The Rabbit virtual machine was written by Tim Gorton in Rabbit C, providing a much more transparent and portable implementation of the virtual machine. The use of C does slow down operational speed somewhat, but it has also allowed us to leverage a very large body of code provided by the Rabbit processor's manufacturer, as well as decreasing the time needed to port the virtual machine. The Rabbit processor implementation supports all of the features described above for the PIC processor version, but the added capabilities of the Rabbit processor have allowed us to add a number of new features to the toolset.

The large amount of additional RAM available to the Rabbit processor provides the space necessary to implement local variables within procedures, and enables a much more powerful multitasking mechanism. By saving twenty separate program stacks, the virtual machine is able to execute up to twenty different Logo programs simultaneously. The additional RAM also makes it practical to store and manipulate strings in memory, so we have created primitives to use the Rabbit C's built-in string manipulation functions.

Rabbit C also permits us to easily use 32-bit signed integers, allowing for vastly larger numbers to be stored in variables. One effect of this change is seen in the fact that the PIC's 15-bit timer overflows after 32 seconds, but the Rabbit's larger one takes 24 days to do so. The Rabbit virtual ma-

chine also provides access to the processor's built-in floating-point math functions, including conversion to and from integers and trigonometric functions.

Another key feature of the Rabbit processors is the fact that the RCM2200 module, which our foundation is built around, includes an on-board Ethernet connector. To complement this, the Rabbit VM provides a number of functions to set the Rabbit's IP address, open socket connections, accept incoming connections, and read and write to these sockets. Using these primitive functions, Tim has written Logo code for standard networking applications, such as a web server and a function to send email. Some of the networking support was enabled by Martin Kang, an undergraduate working with our research group. Custom servers can also be easily written to pass data between Rabbits or between a Rabbit and a desktop computer.

6.3. Software Design

The Tower Development Environment (TDE) *(Figure 6.32)* is an easily-extended integrated development environment for programming and communicating with Tower foundations. The software can be used to program in Logo, downloading PIC assembly code, and visually graph data collected on a Tower. The environment also provides functionality to save and load its state, including all commands previously entered in the command center windows.

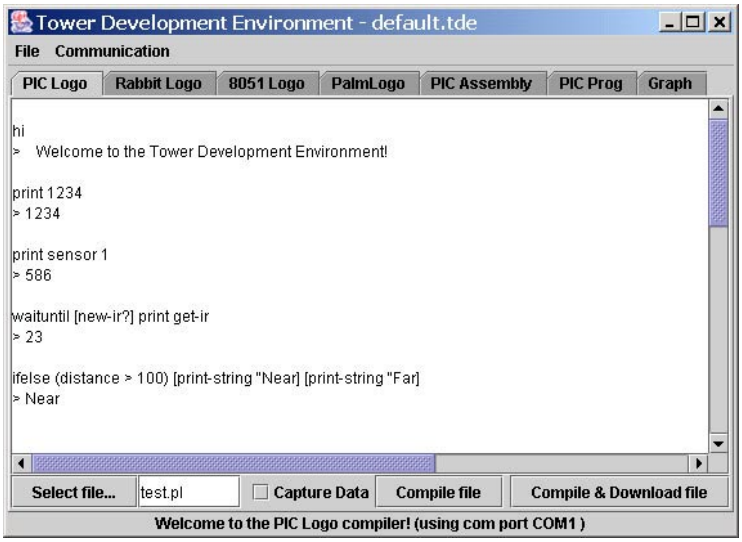


Figure 6.32 - The Tower Development Environment, providing a text-based interface for direct programming of Tower system devices.

The software itself was written by Tim Gorton, based on an underlying toolset originally implemented by Brian Silverman. An overview of the software's structure and operation is provided here to ensure completeness of the system architecture overview.

6.3.1. Interface and Layout

The interface itself uses a tab-based approach to handle the different classes of operation. The current public release has four tabs- PIC Logo, Rabbit Logo, PIC Assembly, and Graph.

The PIC Logo and Rabbit Logo tabs are nearly identical, providing a large text area used as a command center to run single lines of Logo code. When the enter key is pressed in this window, the TDE compiles and downloads the code to the target foundation, which then immediately runs it. Any text sent back across the serial port by the foundation is printed directly into the text area where the line of code was entered, providing an interactive interface for real-time design and debugging. The two buttons at the bottom of the window also provide a means for locating, compiling, and downloading program files containing Logo procedures. By using a special "include" directive in these files, users can dynamically append other text files containing libraries written in Logo, such as those that provide functions for the foundation to talk to particular layers. The Logo compiler functions were originally written by Brian Silverman, but have since been extended by Bakhtiar Mikhak, Fred Martin, Robbie Berg, Tim Gorton, and myself.

The PIC Assembly tab is used for downloading assembly code directly to the PIC Foundation. Using our own language mnemonic, code can be assembled into low-level machine code. The data can then be downloaded by the TDE to a PIC Foundation through its serial port, or exported as a binary file for use by a commercially-available programmer. In this software mode, the main text window is used as a status display for monitoring the assembly and download progress. Similar to the PIC Logo and Rabbit Logo tabs, two buttons are present at the bottom on the window that are used for locating files and initiating the download procedure. The PIC assembler functions were originally written by Brian Silverman, but has been extended by Bakhtiar Mikhak, Fred Martin, Robbie Berg, Tim Gorton, and myself.

The Graph tab allows users to access the sophisticated data-graphing

◦ See reference [6], Coco project website.

module built into the TDE. The graphing module provides a way to capture data from sensors or other input devices, so that it can be graphed within the TDE or saved as a text file for analysis by external software tools. The software supports a number of different standard graphing formats, and the module also utilizes its internal Logo structure to allow data processing in order to generate histograms or other complex data visualizations. The core graphing functionality is enabled by a commercially-available Java graphing package, which has been directly tied in to the TDE's underlying structure by Lele Yu, an undergraduate working with our research group.

In addition to the four main tabs described above, our personal versions of the software used for development contain many more. For example, the foundations under development have corresponding software tabs similar to those for PIC Logo and Rabbit Logo, and an additional PIC Prog tab is available for programming assembled data onto chips through a hardware PIC Programmer such as the PICProg layer. Additionally, a tab is present for Coco°, another project in our group that makes it easy to integrate the Tower system with handheld computers. Using Coco, users can easily program Palm and PocketPC devices running a port of the same virtual machine, making it easy to create tightly integrated applications which communicate with the Tower.

While none of these additional software tabs are currently available in the official distribution, they will be made available as their corresponding hardware and software counterparts are officially released. Consolidating development for the many different sides of an project under a single software umbrella makes it easier than ever before to inspire tight system integration at even the lowest levels of design.

6.3.2. Software Implementation

The significant versatility and modularity of the software environment, is due largely to the fact that the software itself is written in Logo, running on an underlying JavaLogo interpreter. The JavaLogo program structure makes it easy to create user interface software that is both robust and easy to extend. Adding a new tab with its corresponding functionality is as simple as creating the necessary back-end logo functions, and putting a few simple statements in the startup procedure in order to actually generate the tab.

While the bulk of the user interface itself is implemented in Logo, the nature of the JavaLogo structure makes it possible to tightly integrate native Java code into the TDE as well. As previously mentioned, the commercial graphing package we use is written entirely in Java, but was brought into the software and built into the Logo interface structure with relative ease.

The compilers and assembler themselves are also written directly in the JavaLogo language, making the TDE just as easy to extend as Tower-based applications themselves. In fact, adding a primitive to the compiler is as simple as putting it on a list at the end of the file, with information about the number of arguments it expects and whether or not it will return a value. It is important to note, however, that primitive additions must be paralleled both on the compiler side, and in the virtual machine firmware as discussed earlier in this chapter. Providing users with similar Logo syntax for extending the system at all levels encourages them to really explore the ways in which all elements of the Tower system work together..

6.3.3. Interface Alternatives

While the Tower Development Environment is the primary means by which the programming of and direct communication with the Tower are accomplished, it is by no means the only software solution for end-user applications. For many projects, more complex and user-friendly software environments have been created that provide users with a more graphical application-specific interface.

The Tabletop Process Modeling Toolkit[◦] uses a custom interface environment *(Figure 6.33)* implemented in MicroWorlds™, a commercially available Logo software package made by LCSI[□] that makes it easy to create multimedia-rich software prototypes. The interface itself was written by Tim Gorton and Bakhtiar Mikhak and contains representations of each node in the process model, allowing the user to set initial parameters regarding the setup and throughput operations of a particular node. The software runs the same packet-parsing protocol as every node in the system and essentially "delivers" data to a given node. Throughout the entire simulation, the software environment is capable of monitoring packet progress as the data flows through the system. At any point in time, the user can view exactly how and where packets are flowing and reflect upon the ways in which changing model parameters affects the overall system operation.

ALF - The Acrylic Life Form[◦], uses a friendly graphical interface *(Figure*

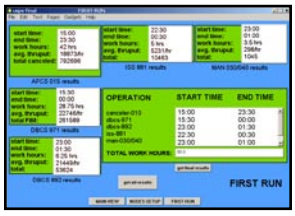


Figure 6.33 - The Tabletop Process Modeling Toolkit software interface.

◦ See reference [16], Tabletop Process Modeling Toolkit thesis, reference [17], conference paper, and Reference [44], project website.

□ See reference [23], LCSI corporate website.

6.34) for the Tower designed to help young children learn to program robots. The software itself maps virtual sliders to each feature on ALF and allows children to interact with him directly by moving feature sliders manually, or by programming up complex sequences and then playing them back. The interface was designed and programmed by Andrew Sempere in Macromedia□ Director™, using a commercial serial communications package to talk directly to the Tower. On the Tower side, a simple Logo program was written to function as a serial wrapper, waiting for communication from the computer and processing it accordingly when received. Early prototypes of graphical software for controlling ALF *(Figure 6.35)* were also implemented in the MicroWorlds software mentioned above. In that implementation, communication with the Tower was accomplished on a lower level, sending byte-code data directly to the foundation to initiate specific function calls. At one point in time, a physical control box *(Figure 6.36)* was created for ALF. With buttons, sliders, and an LCD display built into a tabletop unit with a Tower inside, input commands were processed locally within the controller, and sent by serial communication to the Tower in ALF's head.

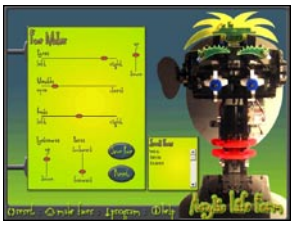


Figure 6.34 - The current ALF software interface.

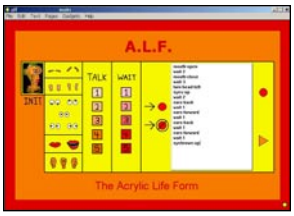


Figure 6.35 - An earlier ALF software interface.

Figure 6.36 - The ALF control box.

The CodaChrome◊ system uses custom interface software *(Figure 6.37)* to give artists the ability to experiment with sequences of colored light patterns to be displayed on a network of Tricolor layers connected to a Tower. The software allows for temporal sequencing of visual patterns by employing a time-line structure, encouraging users to create sequences that swirl, loop, and play off of each other- at times in response to sensor data. At runtime, these complex light sequences are downloaded to the Tower through a special serial-wrapper Logo program on the Tower, which stores the user-defined color sequences on an EEPROM layer for playback at a later time. In this case, the computational power for displaying patterns and handling the corresponding sensor input resides almost exclusively on the Tower itself, allowing the host computer software to focus just on the creation of these sequences. A key benefit of this approach is that sequences can be displayed and repeated without the need for a tethered connection to the design software. The software itself was written in Java by Margarita Dekoli with help from Alex Patino, an undergraduate member of our research group.

Figure 6.37 - The CodaChrome software interface.

All of the software environments discussed above were created for specific applications, and integrated well with the underlying Tower system architecture. Giving users the ability to create their own proprietary software communication environments greatly extends the reach of the

○ See reference [43], ALF thesis, and reference [1], project website.

□ See reference [28], Macromedia corporate website.

◊ See reference [10], CodaChrome thesis, and reference [7], project website.

system far beyond the average technical user. In addition to creating powerful self-contained electronics projects, users are able to prototype complete applications that in the end can have the appeal of professional-looking completed projects, yet developed using our advanced toolkits in a mere fraction of the time.

6.4. System Evolution

Before the Tower was a tower, it was a serial interface module. Initially created as the electronics component of the Musical Instrument Construction Kit (MICK)◦ *(Figure 6.38)*. MICK was designed to teach children about music by allowing them to design and build their own musical instruments. For MICK, the purpose of the interface was to provide high-speed sensor readings and generate MIDI-based musical output, in response to direct communication from a handheld computer.

Figure 6.38 - The Musical Instrument Construction Kit.

At first, the interface we created was just a single circuit board *(Figure 6.39)* with processing, sensing, MIDI, and serial communication hardware all packed into a two-and-a-half-inch square space. While the assembled board was quite appealing visually, a major problem was quickly encountered. With components so densely packed, the board was impossible to debug because we were unable to even fit an oscilloscope probe into the spaces between parts. While some levels of debugging were attempted at this stage, within a few days frustration gave way to a complete redesign.

Figure 6.39 - The original serial interface for the Musical Instrument Construction Kit.

To simplify the testing and development process, the hardware was broken apart into three separate boards *(Figure 6.40)*: one for processing with serial connectors build in, one for sensing, and one for MIDI music generation. To keep the overall electronics package relatively small, these boards were stacked vertically with pin connectors passing the data lines up and down between the layers and the processor at the bottom. Thus, the Tower was born out of necessity, as most good inventions are.

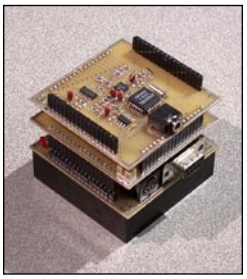


Figure 6.40 - The very first Tower.

At this point in time, the current virtual machine was not in use, so a special firmware program was written with just this application in mind. After specifying a serial packet protocol for handling sensor and MIDI data processing, the program for the foundation was implemented with precisely this functionality in mind. While the system worked successfully for this particular application, it lacked the degree of extensibility it has today.

◦ See reference [45], Musical Instrument Construction Kit thesis, and reference [33], project website.

As the virtual machine extensions discussed earlier in this chapter were beginning to take place specifically for the LogoChip project, a developmental branch occurred as one particular version of the firmware became directly associated with the Tower system◦. With the greater processing power and hardware capacities, the virtual machine that we continued to develop grew into the one used today. Due to the significant extensions that were made for our project, the version that we extended eventually came to replace the one in use on the LogoChips based on more powerful PIC processors. This developmental divergence and re-integration proved to be extremely valuable, as the tight integration between Towers and LogoChips was created, giving both the ability to seamlessly integrate with each other.

Figure 6.41 - The original Tower programming environment.

In addition to the continued firmware development, a new software programming environment was also implemented, initially in the MicroWorlds programming environment *(Figure 6.41)*. Eventually, as support for additional foundations and assembly program was integrated, the migration was made to the current JavaLogo implementation of the Tower Development Environment.

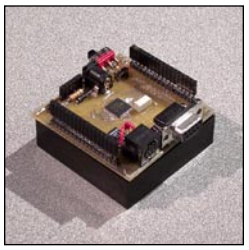


Figure 6.42 - The original PIC foundation.

With the added power of modularity rapidly developing, the hardware system itself began to grow as well. New modules were created with specific applications in mind- a servo motor layer for ALF, as well as an IR communication layer for remote sensing and data collection applications, and a blank prototyping layer to allow for the easy connection of basic electronic parts such as LEDs and buttons. However, while the system was just beginning to grow, we were already having trouble with power issues, and the pin connectors that we had initially used for inter-layer connectivity were beginning to degrade.

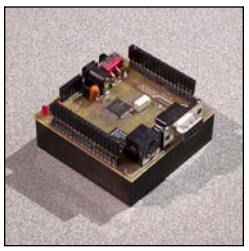


Figure 6.43 - The PIC foundation with its newly redesigned power circuitry.

The Foundation itself went through several design iterations. The original PIC Foundation had built in battery charging *(Figure 6.42)*, but the circuitry was later removed *(Figure 6.43)* to prevent incidents arising from the attempted charging of non-rechargeable batteries. This new modification allowed for power to be provided by either pre-charged batteries, or a plug-in transformer via a power jack. In the next revision, the connectors themselves were switched for more structurally stable surface mount ones *(Figure 6.44)*. In addition to solving the main problem, the new connectors greatly reduced the wiring overhead needed between top and bottom connectors on each layer, since their surface-mounted nature allowed them to be stacked directly on top of each other.. While changing connectors at

Figure 6.44 - The PIC foundation with the new surface-mount connectors.

◦ A more in depth description of the relationship between the LogoChip and the Tower can be found in section 2.2.3 of this document.

this stage in development did require the redesign of some layers that had already been developed, it was a worthwhile change that to this day we are still very happy with.

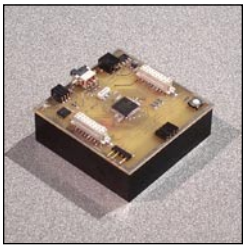


Figure 6.45 - The PIC foundation with its serial circuitry moved off-board.

However, with the integration of the new connectors, came another problem. The vertical space between stacked boards was significantly reduced, requiring larger connectors such as those for serial cables to be moved off of the foundation *(Figure 6.45)*. The serial connector was replaced with a separate line-driver board, the Tower system's RS-232 module, which connects directly to the foundations through a small 4-pin header. This module has proven to be an extremely versatile option for connecting serial cables to electronics, and is still widely used by many of our modules.

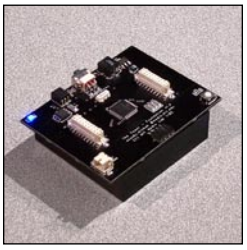


Figure 6.46 - The current PIC foundation.

The last foundation revision to occur was based on an influx of comments from initial users who were finding it impossible to reach the small white button on the foundation while layers were stacked on top. To remedy this situation, the foundation itself was widened slightly *(Figure 6.46)*, providing both a more accessible start/stop button and a more visible power indicator LED.

Despite the stabilization of the foundation infrastructure at this point in time, one major hardware architecture revision was still yet to take place. In the early stages of Tower development, only some layers had microcontrollers on-board. Layers that did not need to perform timing-critical functions were left to communicate directly with the foundations, partly in an effort to reduce the costs of each layer.

With the system in a stable design stage, my undergraduate thesis was written on the underlying hardware architecture[◦]. But as development continued, we ran into a major roadblock. Many electronic components used in the system such as real time clock chips and serial EEPROMs had been chosen for their I^2C compatibility. However, with different manufacturers choosing to use the same hard-coded addresses as each other, device conflicts would often occur. Unfortunately, there was no simple solution to this, as all of the I^2C parts were simply connected to the same bus, with nothing capable of arbitrating communications meant for one layer versus another with the same address.

◦ See reference [27], Tower System Advanced Undergradute Project.

As a solution to this, we began putting microcontrollers on each layer, allowing them to handle all of the I^2C addressing between boards. In addition to solving this major problem, it gave us the opportunity to em-

phasize the modular nature of the system to an even greater level. With the new modifications, users are now able to easily "scan" the Tower to see exactly what is plugged into the stack at a given time. Additionally, this provides for the ability to change the address of any layer during real-time operation, allowing for multiple copies of the same layer to be used in a given application. What initially appeared to be a solid obstacle in our path led to a fundamental rethinking of the system architecture itself. To this day, the modifications we made continue to successfully support the addition of new layers created by a growing number of developers around the world.

Due to our extensive work on the underlying framework of the entire system architecture, complete modularity has been preserved at all levels throughout the course of the Tower system's evolution. With the simplicity of the tools we have created to encourage continued development of the system, significant extensions have been made possible for any user with virtually any depth of technical expertise.

A father and daughter reflect upon their shared experience creating electronically-enhanced articles of clothing at a workshop in Dublin, Ireland.

Chapter 7 - Reflections

In the end, the Tower system is a tool, in some ways similar to others that we all use everyday, but in some ways very different. It is a tool that promotes a global culture of toolbuilding, and encourages aspiring designers and inventors to dig deeper into the challenges they face. By using the system we have designed, users are learning how to take the complex problems that they are faced with and break them into simpler parts in order to arrive at optimal solutions.

In fact, the power of a system such as the Tower lies not in the solutions it helps people to create, but the ways in which it guides people to those solutions. The process of exploration is something that encourages people to keep thinking, effectively recombining ideas and solutions they have encountered in the past and applying them in new configurations to solve even more complicated problems that arise. This is the heart of innovation, putting together a bunch of small solutions to create an all-encompassing one that is greater than the sum of its individual parts.

7.1. Discussion and Conclusions

We set out to design an electronics development system that encourages

innovation. The core idea was simple: to create something powerful, flexible, and extensible that users of any skill level would feel comfortable with, but would never impose technical limitations on those pushing its boundaries.

The Tower system is the final product of almost two years of work, and by all measures it meets the goals set forth in our original design objective. What started with a single foundation and a core set of layers for basic functionality has continued to flourish, with layers now being developed by other individuals both within and outside of our research group.

Out in the real world, the Tower system has found a home with a wide variety of users representing vastly different communities. Successfully used for applications ranging from education to development to rapid prototyping, our system design has proven itself extremely powerful and versatile, as well as capable of adapting to meet new technical challenges that arise as applications are developed.

On an even more important level, those using the system have experienced the joys of finding their own solutions to problems that are personally meaningful to them. The list of applications successfully using the Tower system is long: an environmental sensing station, a tabletop greenhouse, interactive robotic creatures, a toolkit for modeling industrial processes and teaching how complex networks are built and operated, a model for teaching the laws of physics, a musical instrument construction kit, a tool for enhancing the performance of diesel-powered engines, a system for allowing aspiring artists to play with light and color in fun and unique ways, a miniature oscilloscope, a food dehydrator, a video game construction kit, a 3D scanner and multi-dimensional input device, a pinball machine, a low-cost irrigation system for flower farmers, a novel toolkit for programming moving sculptures and robots, a miniature city with moving cars and working traffic lights, an integrated system for prototyping kinetic structural networks, a miniature violin, spinning bicycle wheels signs, a giant music box, a tabletop 2D plotter, and many, many more...

We will soon realize the ultimate Tower-based application: a Tower itself. With the ability to build a miniature milling machine, oscilloscope, and chip programmer, all the building blocks are now in place to allow someone to design a circuit board, cut it out of copper, solder it up, program it, test it, and use it. These are the first steps towards the creation of a self-replicating system. While these tools are currently being used to extend

the system itself, it is not difficult to imagine ambitious students deciding to create their own system: perhaps more powerful and better than ours in many ways but building on the foundations we have laid for them.

By creating a system whose technical boundaries dissolve as approached, we have successfully engineered the learning curve for those exploring the art of integrated system design. Just using the Tower system itself teaches people the importance of finding solutions with high ceilings, and how not being limited by a specific "correct" answer to a problem allows for uninhibited growth and development of original ideas. As users solve problems by piecing together knowledge they have gained through their personal explorations, they are innovating at the purest level. Encouraging innovation is the greatest goal that we can hope to accomplish through our work.

7.2. Future Directions

While my work thus far has focused largely on the principles of electronic systems, I am deeply interested in extending the current toolset to encompass mechanical building blocks as well, building off of and alongside the developing field of personal fabrication here at the lab. I hope to use these new tools to rethink the ways in which integrated systems are both envisioned and realized, believing that there is a great potential for fundamentally revolutionizing the way in which technical design principles are not only taught and learned, but also how they are employed in practice. Over the years, my various projects have encompassed a significant range of depths from low-level circuit design on the silicon level to programming, mechanical, and system design on room-sized and larger scales. I would like to bring my extensive skill set to bear on a wide range of social and educational problems facing the world today.

I hope to remain an active part of our continued collaborations with educators and students around the world as part of the Learning Independence Network, a flagship project of our research group. Recently, I have been directly involved in these efforts and I have seen the students we worked with receive praise for how much they have accomplished in such a short time frame using the tools we have created. By introducing these students to the ideas and tools of rapid prototyping and the beginnings of our online user community, we are hoping that they will benefit not only from the assistance we can give them, but also from working with

students here and in other countries and institutions as well. The current implementation of this online community is still in its beginning stages and requires a significant time investment to truly succeed. Working to develop and structure this community to the extent where students around the world begin to use it regularly will be a major step forward. Doing so will provide an interesting opportunity for us to explore the degrees to which collaborative technical design knowledge can be used to bridge social, technical, and international borders and build interactions among people with vastly different technical and creative mindsets.

With these new research directions, it will be exciting to see the strengths of our current user network successfully augmented by the mechanical and fabrication tools I plan to develop, as well as the creation of a powerful framework for the exchange of ideas and interactions between aspiring designers around the world. In the end, the best possible outcome would be to see the user community continue the development process of the electronic and mechanical systems themselves, and use that as a basis for designing their own systems and exploring new concepts directly relevant to all the dimensions of their lives.

References

[1] ALF project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/alf *(accessed 5/2003)*

[2] Applied Data Systems corporate website. http://www.applieddata.net *(accessed 5/2003)*

[3] Bahuman, A., Sun, A., Kalbag, S., and Chuang, I. (2003). Vigyan Ashram - Engine Tune-Up Project. MIT Media Lab/Center for Bits and Atoms internal document.

[4] Baldwin, C. and Clark, K. (2002). Design Rules, Volume 1: The Power of Modularity. MIT Press. Cambridge, MA.

[5] Chuang, I. and Sun, A. (2002). Vigyan Ashram Trip Report. MIT Media Lab/Center for Bits and Atoms internal document.

[6] Coco project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/coco *(accessed 5/2003)*

[7] CodaChrome project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/codachrome *(accessed 5/2003)*

[8] Cricket System project website. MIT Media Lab, Lifelong Kindergarten Group. http://cricket.media.mit.edu *(accessed 5/2003)*

[9] Cygnal corporate website. http://www.cygnal.com *(accessed 5/2003)*

[10] Dekoli, M. (2003). Coloring Time with CodaChrome. Master of Science Thesis, Massachusetts Institute of Technology. Cambridge, MA.

[11] Environmental Sensing and Instrumentation project website. http://web.media.mit.edu/~tagdata *(accessed 5/2003)*

[12] Etch project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/etch *(accessed 5/2003)*

[13] FabLab project website. MIT Center for Bits and Atoms. http://cba.media.mit.edu/projects/fablab *(accessed 5/2003)*

[14] Flogo project website. MIT Media Lab, Lifelong Kindergarten Group. http://llk.media.mit.edu/projects/flogo *(accessed 5/2003)*

[15] GoGo Board project website. MIT Media Lab, Future of Learning Group. http://http://learning.media.mit.edu/projects/gogo *(accessed 5/2003)*

[16] Gorton, T. (2003). Tangible Toolkits for Reflective Systems Modeling. Master of Engineering Thesis, Massachusetts Institute of Technology. Cambridge, MA.

[17] Gorton, T., Mikhak, M., and Paul, K. (2002) Tabletop Process Modeling Toolkit: A Case Study in Modeling US Postal Service Mailflow. Demonstration at CSCW 2002, November 16-20, New Orleans, Louisiana.

[18] Greenberg, S. and Fitchett, C. (2003). Phidgets: Easy Development of Physical Interfaces Through Physical Widgets. Proceedings of the ACM UIST 2001 Symposium on User Interface Software and Technology, November 11-14, Orlando, Florida. ACM Press.

[19] Hancock, C. (2002). Toward a Unified Paradigm for Constructing and Understanding Robot Processes. IEEE Symposium on Human Centric Computing Languages and Environments 2002.

[20] Instituto Nacional de Astrofízica, Óptica, y Electrónica website. http://www.inaoep.mx *(accessed 5/2003)*

[21] Instituto Tecnológico de Costa Rica website. http://www.itcr.ac.cr/ *(accessed 5/2003)*

[22] Jet project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/jet *(accessed 5/2003)*

[23] LCSI corporate website http://www.cygnal.com *(accessed 5/2003)*

[24] LINCOS project website. http://www.lincos.net *(accessed 5/2003)*

[25] Learning Independence Network project website. http://gig.media.mit.edu/projects/lin *(accessed 5/2003)*

[26] LogoChip project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/logochip *(accessed 5/2003)*

[27] Lyon, C. (2002). The Tower: A Modular Electronics Design Environment. Advanced Undergraduate Project, Massachusetts Institute of Technology. Cambridge, MA.

[28] Macromedia corporate website. http://www.cygnal.com *(accessed 5/2003)*

[29] Martin, F., Mikhak, B., and Silverman, B. (2000). Metacricket: A Designers Kit for Making Computational Devices. IBM Systems Journal. Volume 39, Numbers 3 & 4.

[30] Microchip corporate website. http://www.microchip.com *(accessed 5/2003)*

[31] Microsoft corporate website. http://www.microsoft.com *(accessed 5/2003)*

[32] Mikhak, B., Gershenfeld, N., Lyon, C., Gorton, T., McEnnis, C., and Taylor, J. (2002). FAB LAB: An Alternate Model of ICT for Development. Development by Design Conference, December 1-2, Bangalore, India.

[33] Musical Instrument Construction Kit project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/mick *(accessed 5/2003)*

[34] Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. Basic Books. New York, NY.

[35] Parallax corporate website. http://www.parallax.com *(accessed 5/2003)*

[36] Phidgets corporate website. http://www.phidgets.com *(accessed 5/2003)*

[37] Rabbit Semiconductor corporate website. http://www.rabbitsemiconductor.com *(accessed 5/2003)*

[38] Resnick, M., Berg, R., and Eisenberg, M. (2000). Beyond Black Boxes: Bringing Transparency and Aesthetics Back to Scientific Investigation. Journal of the Learning Sciences. Volume 9, Number 1.

[39] Resnick, M., Bruckman, A., and Martin, F. (1996). Pianos Not Stereos: Creating Computational Construction Kits. Interactions. Volume 3, Number 6.

[40] Resnick, M., Eisenberg, M., Berg, R., Mikhak, B., and Willow, D. (2000). Learning with Digital Manipulatives: New Frameworks to Help Elementary-School Students Explore "Advanced" Mathematical and Scientific Concepts. Proposed to the National Science Foundation.

[41] Robotany project website. MIT Media Lab, Grassroots Invention Group. http://web.media.mit.edu/~sarac/main.html *(accessed 5/2003)*

[42] Schrage, M. (2000). Serious Play: How the World's Best Companies Simulate to Innovate. Harvard University Press. Cambridge, MA.

[43] Sempere, A. (2003). Just Making Faces? Animatronics, Children and Computation. Master of Science Thesis, Massachusetts Institute of Technology. Cambridge, MA.

[44] Tabletop Process Modeling Toolkit project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/process *(accessed 5/2003)*

[45] Thibault, S. (2001). MICK: A Design Environment for Musical Instruments. Master of Engineering Thesis, Massachusetts Institute of Technology. Cambridge, MA.

[46] Topobo project website. MIT Media Lab, Tangible Media Group. http://web.media.mit.edu/~amanda/topobo/topobo.html *(accessed 5/2003)*

[47] Tower Support Forum website. MIT Media Lab, Grassroots Invention Group. http://tower-support.media.mit.edu *(accessed 5/2003)*

[48] Tower System project website. MIT Media Lab, Grassroots Invention Group. http://gig.media.mit.edu/projects/tower *(accessed 5/2003)*

[49] Villegas, M. (2003). Lutec Project Notes. LuTec internal document.

[50] Yang, A. (2001). Gameweaver: A Construction Kit for Kids to Create Video Games for Handheld Devices. Master of Engineering Thesis, Massachusetts Institute of Technology. Cambridge, MA.

[51] Zuckerman, O. and Resnick, M. (2003). A Physical Interface for System Dynamics Simulation. CHI 2003 Extended Abstracts. ACM Press.

Appendices

Appendix A: Getting Started with the Tower

Setting up and using any device in the Tower system is quite simple. This tutorial covers the basics of operation, including:

- Setting Up the Tower
- Using the Tower Development Environment
- Downloading Assembly Code
- Programming Logo Code
- Using the Command Center
- Standalone Operation
- Adding Layers
- Graphing Data

While the Tower family encompasses a broad range of devices, they have been designed to operate in a very similar manner to each other. Due to the many similarities between them, this document covers only the use of the PIC Tower itself.

The LogoChip and LogoChip modules operate almost identically to the PIC Tower, with the exception of needing external power and more support circuitry to function. While no connectors for attaching layers are present, a special cable connector on the LogoChip module can be used to connect to a Tower via the I^2C breakout layer.

The LogoBoard functions almost identically to the PIC foundation, but also lacks the connectors needed to stack additional layers.

The Rabbit foundation is very similar to the PIC foundation, only differing in the manner by which assembly code is downloaded. Assembly code may only be downloaded to the Rabbit foundation via the commercially-available Rabbit processor programming kit. All Rabbit foundations come with the Logo virtual machine pre-downloaded.

While other foundations are not currently supported in the Tower public distribution, their operation will be similar to that described here, due to our tight system design specifications.

Setting up the Tower

To get started, you will need:

- A Tower foundation (In this tutorial we will use a PIC foundation)
- A Tower system RS-232 serial module
- A computer serial cable
- Either four AA batteries or a wall power adapter with a 1mm jack

Let's start by looking at the foundation in detail *(Figure A.1)*:

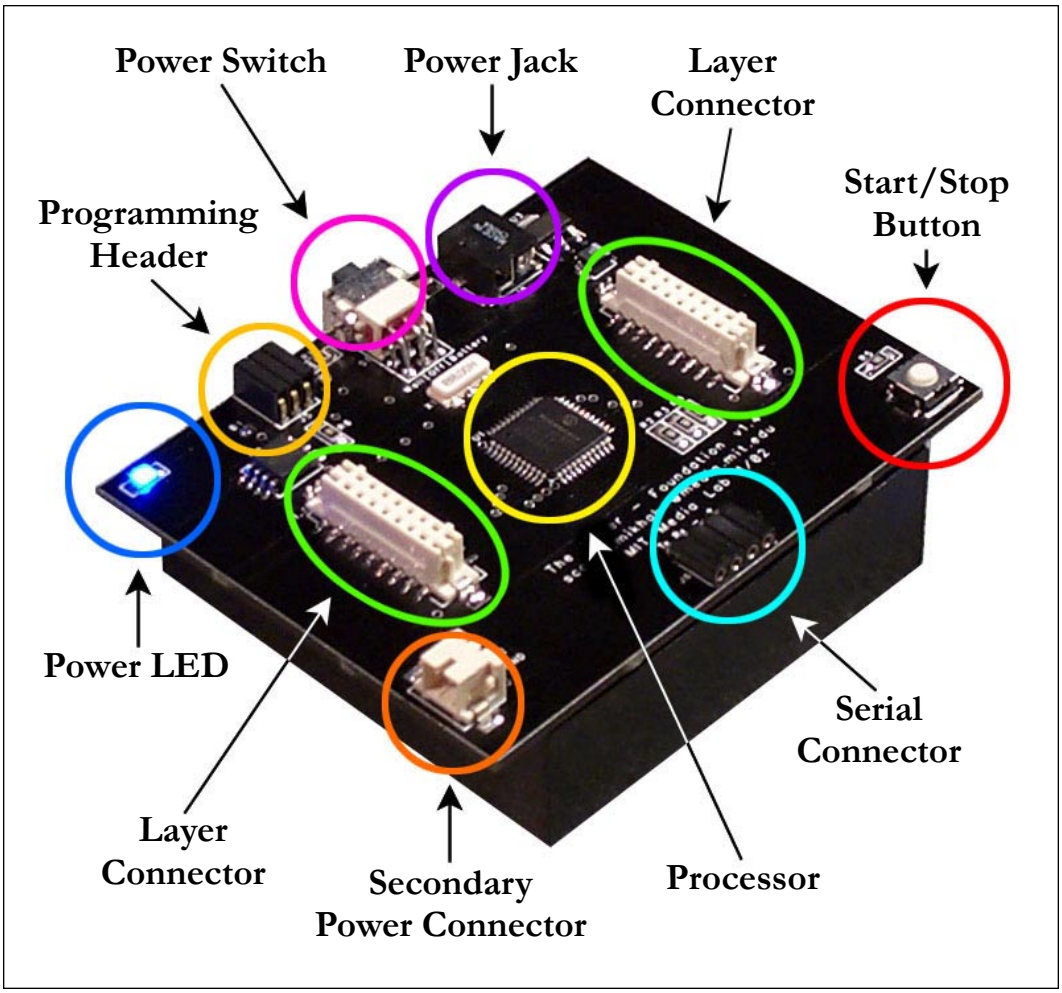


Figure A.1 - PIC Foundation parts diagram.

Before doing anything else, the foundation must be powered. There are two different options for providing power: either a wall transformer, or batteries.

If using a transformer, it must provide an output voltage greater than 6 volts and have a negative-tip 1mm connector. The connector should be plugged directly into the "Power Jack" on the foundation.

If using batteries, four AA batteries should be installed in the battery pack on the underside of the foundation, as shown below *(Figure A.2)*:

Figure A.2 - Installing batteries in the foundation.

The PIC foundation can use alkaline or rechargeable batteries, but the Rabbit foundation will only function properly with rechargeable ones due to tight voltage requirements.

If using a wall transformer, connect the 1mm power connect as shown below *(Figure A.3)*:

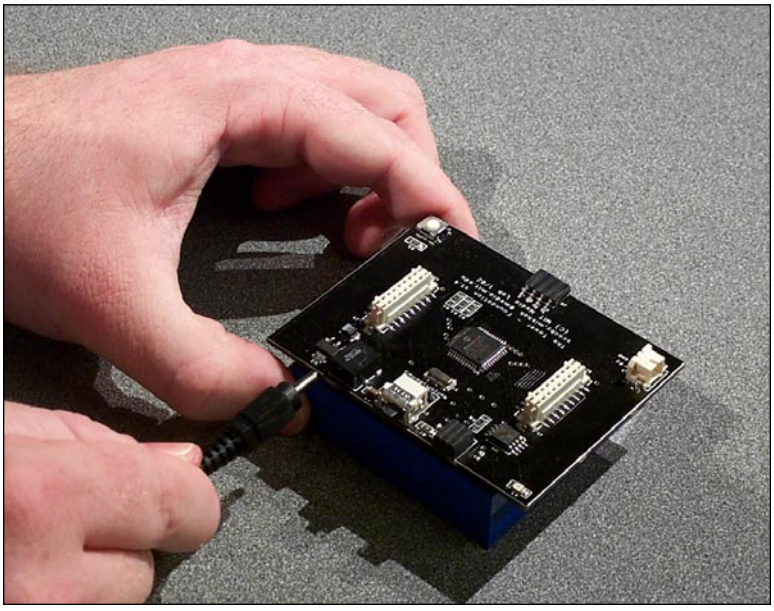


Figure A.3 - Connecting a wall transformer.

Next, we need to connect the Tower to a personal computer for programming. Attach the serial cable to the RS-232 module, and plug the module into the serial header on the foundation as shown here *(Figure A.4)*:

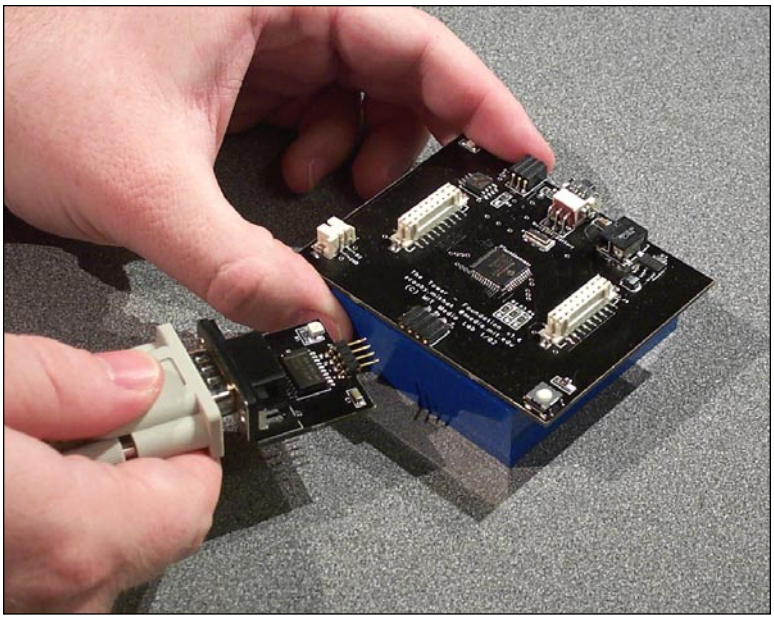


Figure A.4 - Connecting a serial cable to the foundation.

Be sure to connect the other end of the serial cable to one of the COM ports on the back of your computer. Any port will work, but COM1 will be the simplest if it's available.

Now, turn on the power using the main switch on the foundation *(Figure A.5)*:

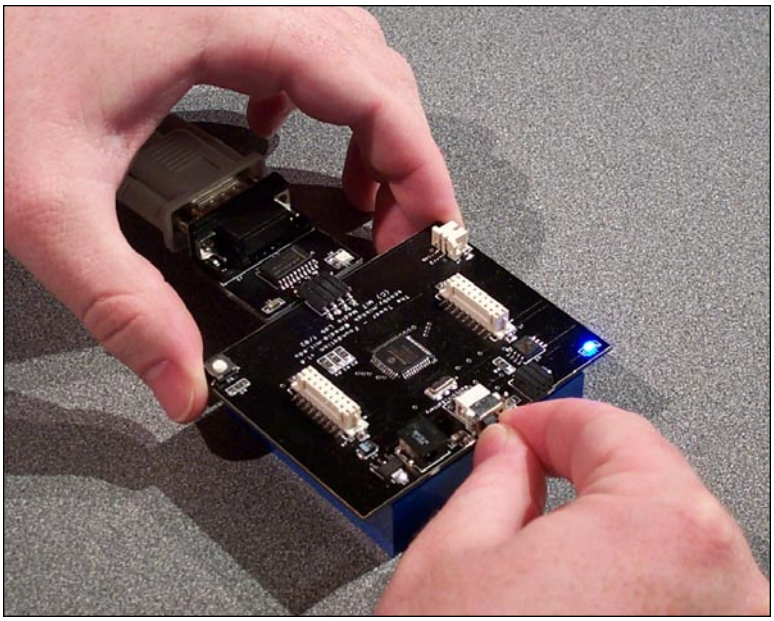


Figure A.5 - Turning on the power.

The power switch has three positions. When in the center position, power to the foundation is off. One side powers the foundation from the wall transformer, and the other side powers it from the batteries. This allows the Tower to be completely powered off even if both the wall transformer and batteries are present at the same time.

At this point, the Tower is set up and ready to use *(Figure A.6)*:

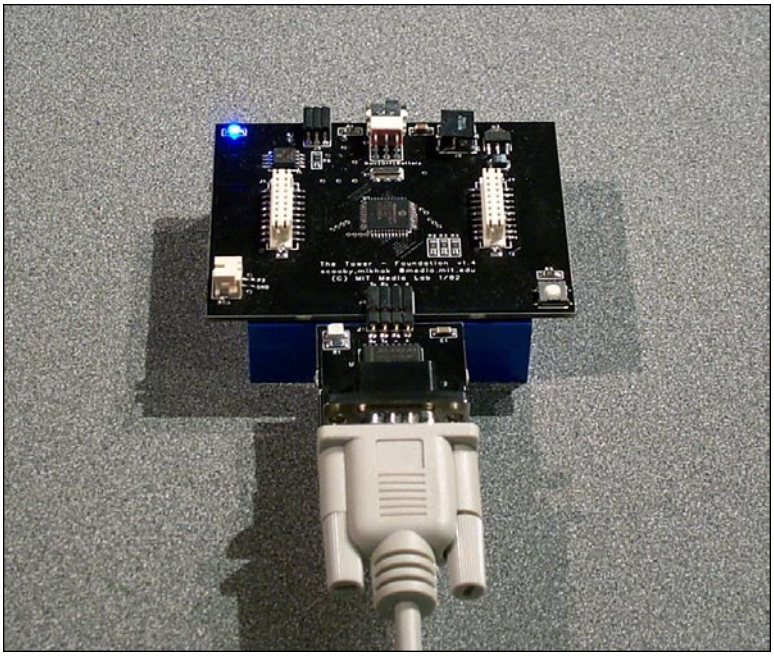


Figure A.6 - A foundation set up and ready to use.

Using the Tower Development Environment

To program the Tower, you will need to download our host computer software, the Tower Development Environment (TDE). The most recent version of the software can be downloaded from the Tower project website at:

- http://gig.media.mit.edu/projects/tower

Unzip the file into a directory of your choice. For the software to run properly, you will need to have Java Runtime version 1.4.1 or later properly installed. The most recent Java distribution can be downloaded from the Sun Microsystems Java website at:

- http://java.sun.com

Once Java has been installed and the TDE has been unzipped, we are ready to start programing the

Tower. In the directory that the TDE was unzipped to, there should be a file called "run-towerdev.bat". Double-click on the file's icon to launch the software.

A command center window will first appear, followed by the graphical TDE interface. When the program first opens, it should look something like this *(Figure A.7)*:

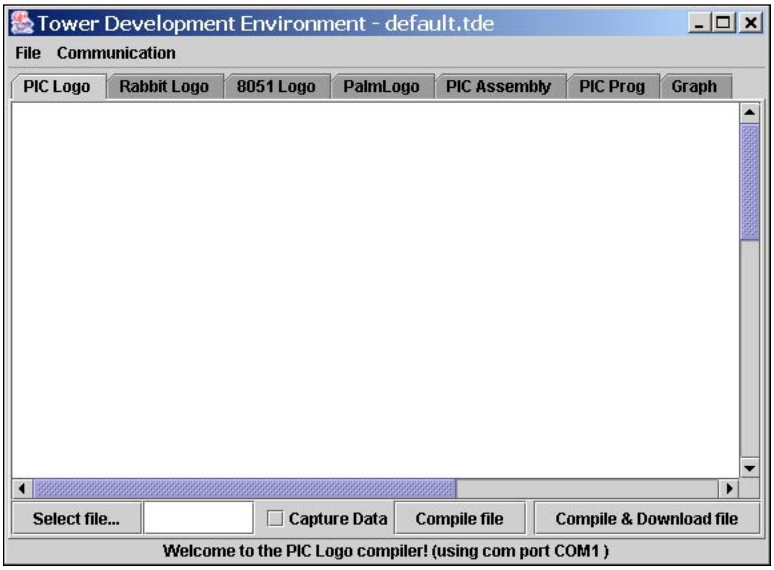


Figure A.7 - The Tower Development Environment.

Let's take a look at the software interface. At the very top of the window are two menus: File and Communication. The File menu can be used to save and load your workspace settings. Saving the TDE workspace will preserve any text present in any one of the command centers, the most recent program file loaded, the COM port selection, and the panel most recently used (we will talk more about panels shortly). The Communication menu allows you to select what COM port the Tower is connected to. By default, it is set to COM1, but if you plugged the Tower into a different port, it should be selected here.

Directly below the menus, are a series of tabs. Each tab corresponds to a different software panel, providing functionality for a variety of devices that are part of the system. Depending on the current state of supported devices, the software you download may or may not have the same tabs present in the picture above.

For now, we will only concern ourselves with the PIC Logo and PIC Assembly tabs, since we are using the PIC foundation. Switching between panels is accomplished by clicking on the tabs themselves.

Below the tabs, is the Command Center. The Command Center is used to communicate directly with Tower system devices. Any line of code can be typed directly into the Command Center, and when the "Enter" key is pressed, that code will be run on the Tower. The command centers are very useful for rapid debugging and development.

Below the Command Center, are the buttons used for loading files, compiling, and downloading them to Tower system devices.

The Select File button is used to load a program file. There are a few sample program files located in the "pic" subdirectory, all of which us the file extension ".pl" to indicate PIC Logo. Assembly code files (.asm) and pre-assembled files (.hex), such as the virtual machine, can be found in the "asm" sub-directory.

To the right of the Select File button, is a checkbox for Capture Data. The Capture Data checkbox turns on data capture for the built-in graphing package. We will discuss its use later in this tutorial.

The next two buttons are Compile File and Compile & Download File (these buttons have slightly different names in some of the other panels). These buttons do just what their names imply. The Compile File button is useful for testing out new code to ensure that it is error-free. Error messages will pop-up if there is a problem during compilation. When a program compiles properly, the Compile File & Download button can be used to load it onto a Tower system device.

While the second button can just as easily be used to test for errors, it will attempt to download the program code, which can sometimes take a bit of time if a file is large. Often, simple program errors can be found without needing to download the code to a Tower.

Downloading Assembly Code

Before using the Tower, the very first thing you should do is download the most recent virtual machine onto it. To do this, switch to the PIC Assembly tab in the TDE by clicking on its name. Now, we need to load the virtual machine file. Click on the Select File button, and when the select file box opens, double-click on the "asm" subdirectory. Select the "tower_vm.hex" file (or the LogoChip version, depending on what device you are programming) as shown here *(Figure A.8)*:

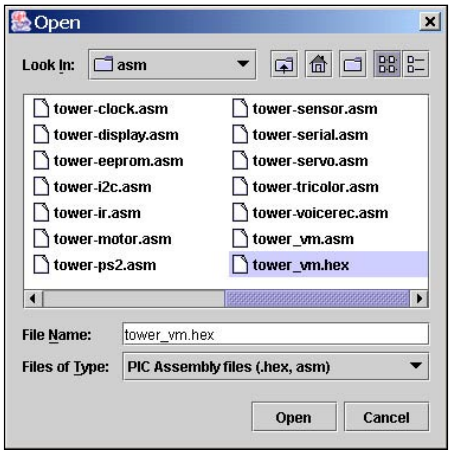


Figure A.8 - Selecting the virtual machine file.

In order to download assembly code to the foundation, it must be in "bootflash" mode. To get it in this mode, turn off the power, then turn it back on while holding down the white button on the foundation. The power LED will begin to flash. On a LogoChip module, the power/run LED will be off completely.

Now, click the Assembly & Download File button at the bottom of the window. After successfully downloading the virtual machine, the TDE should look something like this *(Figure A.9)*:

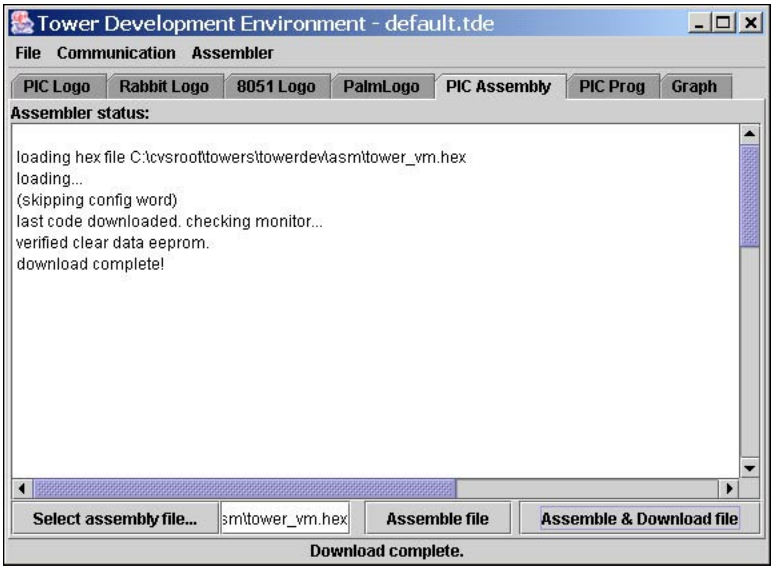


Figure A.9 - Downloading assembly code.

Once the download is complete, turn the foundation off and then back on. The power LED should flash briefly and then remain on, indicating that the virtual machine is running properly.

Downloading Logo Code

Now that the most recent virtual machine is on foundation, we are ready to start writing Logo programs and downloading them to the foundation.

Switch to the PIC Logo panel by clicking on the corresponding tab. Let's start by loading the sample file "test.pl" which uses a few "include" statements and contains one procedure. To look at and edit "test.pl", use a standard text-editor such as Notepad. The initial contents of the file are shown here:

```
include pic/include/standard.inc
include pic/include/print.inc

to test
    print-string "|This is a test.|
end
```

The use of include files and procedure definitions are further explained in the "PIC Logo Language Reference" and "PIC Foundation Documentation" appendices.

Essentially, this sample program includes two of the basic files: one containing variable definitions and pin manipulation procedures, and the other containing functions for printing text and numbers back to the host computer for printing.

The procedure "test" simply tells the foundation to print the string "This is a test." in the TDE Command Center window. Note that the format for a string is a double quotation mark followed by a vertical bar, then the string, then another vertical bar at each end.

In the same manner that we downloaded the virtual machine, click the Select File button and open the "pic" subdirectory. Double-click on the file "test.pl" to load it as shown below *(Figure A.10)*:

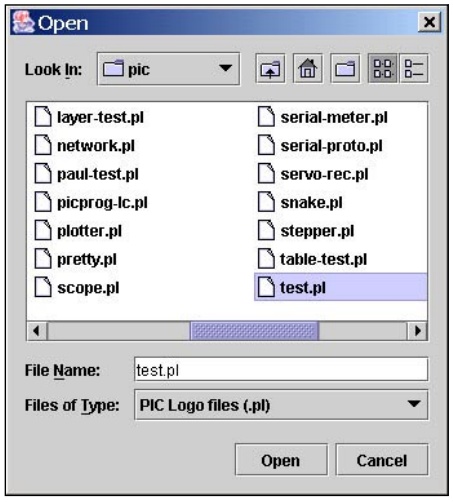


Figure A.10 - Selecting the sample Logo file.

Now, with the Tower connected and turned on, click the Compile & Download File button. Download progress will be shown in the status bar at the bottom of the screen. After successfully compiling and downloading the program file, we are ready to run our program.

Using the Command Center

Remember that procedure we wrote in "test.pl" called test? Type the name of the procedure ("test") into the Command Center and press the "Enter" key. The string "This is a test." should print directly in the command center, like this:

```
test
> This is a test.
```

However, since the procedure "print-string" was also downloaded, it can be called directly from the Command Center as well. Hit the "Enter" key a few times to create some blank lines, and then call the "print-string" procedure directly like this:

```
print-string "|This is a test.|
> This is a test.
```

At this point, the TDE window should look something like this *(Figure A.11)*:

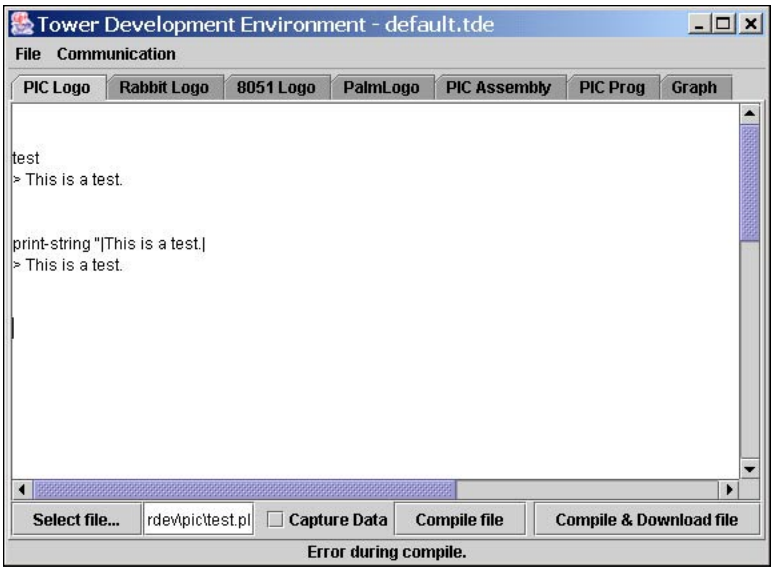


Figure A.11 - Using the Command Center.

When using the command center, you can type anywhere in it, and it is easy to re-run previously written lines of code. Hitting the "Enter" key on any line will cause that line of code to be run immediately. If you want to insert blank lines anywhere in the window, hold down the "Shift" key while pressing "Enter." When inserting new lines, pressing "Enter" will not run any code.

Standalone Operation

While the Command Center is very useful for developing applications, there are many occasions when it is important to not need a personal computer in addition to the Tower.

Any program that can be run from the Command Center can easily be mapped to the white button on the foundation, or made to automatically run whenever the Tower is turned on.

To map the "test" procedure to the white button, add a line like this to the "test.pl" file:

```
on-white-button [test]
```

After saving the program file, press the Compile & Download File button to load this new code onto the foundation.

When the file is done downloading, try pressing the white button. The test should print in the command center like this:

```
> This is a test.
```

If you press the button multiple times, the string will keep printing:

```
> This is a test.This is a test.This is a test.
```

Note that there is no line break between the strings. If we wanted to add one, it would be as simple as calling the procedure "cr" (carriage-return) within our "test" procedure after the string is printed. The modified function would look like this:

```
to test
    print-string "|This is a test.|
    cr
end
```

After downloading the new code, when the button is pressed multiple times the following will be printed:

```
> This is a test.
> This is a test.
> This is a test.
```

If we want to make our "test" procedure run every time the Tower is turned on, the following line can be added to the program file:

```
on-startup [test]
```

After downloading the new program, the text string will be printed to the Command Center when power is turned on:

```
> This is a test.
```

Adding Layers

Now that we've explored the basic operation of the foundation, let's try adding a layer to make things more interesting. To continue, you will need:

- A Sensor layer
- A light (or other) sensor with a connector to match the Sensor layer

To put on a layer, first line up the two main connectors with those on the foundation. They will only fit one way. Once the connectors are lined up, press down firmly on the layer to attach it to the foundation *(FigureA.12)*:

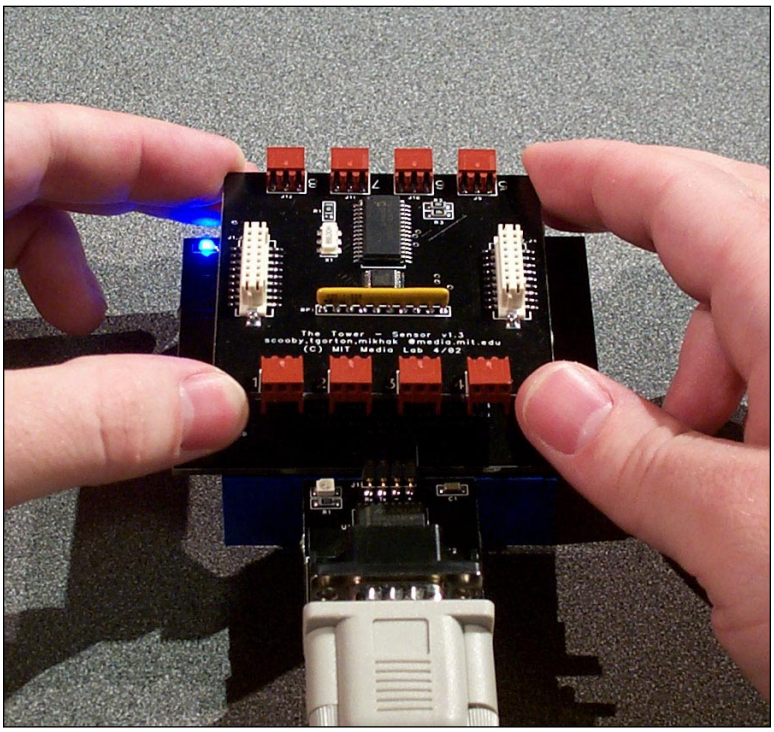


Figure A.12 - Attaching a Sensor layer to the foundation.

In this example, we're installing a Sensor layer, but attaching any other layer is a similar process. Now, we'll see if the foundation knows that it now has a layer attached.

Let's add a line to the "test.pl" file, to also include the "address.inc" procedures file. The new line should look like this:

```
include pic/include/address.inc
```

The address include file is needed for "scanning" the Tower to determine what layers are present, as

well as changing the addresses of any layer connected.

After downloading this modified test program, we can run the "scan-tower" procedure to see if the Sensor layer is found:

```
scan-tower
> Sensor layer located at address 10
> Done!
```

However, before using the Sensor layer (or any other layer), it is necessary to also include its respective procedures file, by adding this line to the program as well:

```
include pic/include/sensor.inc
```

Full documentation for the Sensor layer can be found in the "Sensor Layer Documentation" appendix, but for now, all we need to know is that the include file for the layer contains one function called "sensor." The function takes a single argument, the sensor port number (from 1 to 8), and returns the resulting sensor value.

Let's plug a light sensor into port "5" of the Sensor layer, as shown below *(Figure A.13)*:

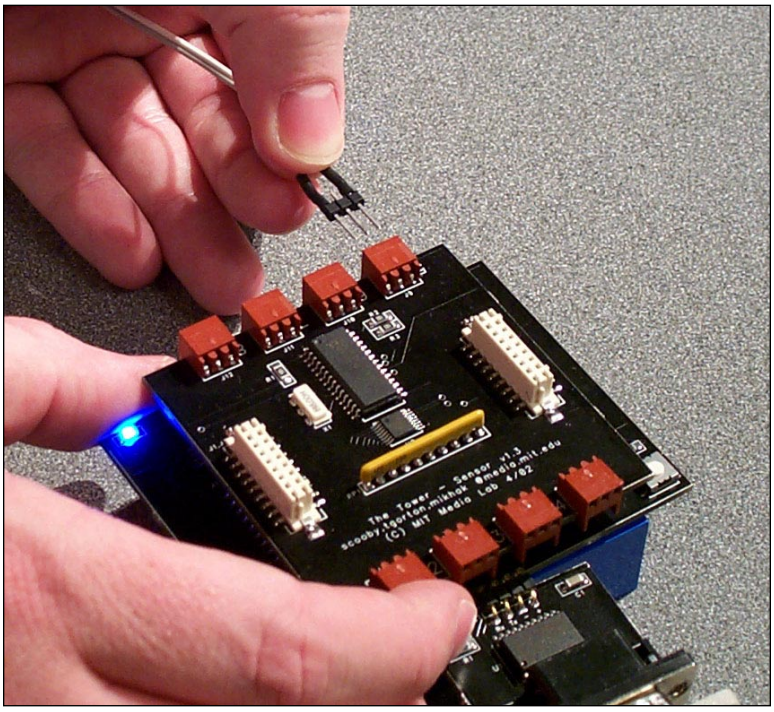


Figure A.13 - Plugging a light sensor into the Sensor layer.

Now that we have a sensor connected, we can read its value and print the result directly to the Command Center by typing the following command:

```
print sensor 5
> 1344
```

In this case, the sensor value happened to be 1344. Let's move the light sensor around and run it a few more times to see some different numbers:

```
print sensor 5
> 848

print sensor 5
> 1926

print sensor 5
> 1501
```

It's easy to make the command automatically repeat. By running the following line of code, three values will be printed to the Command Center immediately following each other.

```
repeat 3 [print sensor 5 wait 10]
> 473
> 2256
> 1012
```

The "wait 10" statement is put in the repeat loop to ensure that the foundation waits for one second (the argument is given in tenths of a second) in between sensor readings. While not at all necessary, waiting between readings ensures that more interesting values will be obtained, rather than three almost identical ones.

Graphing Data

Now that we're successfully reading sensor data, we can use the TDE's built-in graphing program to plot our sensor data. In the previous example, we were printing sensor values directly to the Command Center. If we want to plot them instead, it's as simple as checking the "Capture Data" box at the bottom of the window. Now, let's run the following code in the Command Center:

```
repeat 10 [print sensor 5 wait 10]
```

In this case, we're taking ten sensor readings, and waiting one second in between each. While the Tower is busy "thinking," the power LED will pulse on and off. When the LED is pulsing, the Tower will ignore all other communication from the computer. If you ever want to stop a program that is running, it can be done by just pressing the white button. The LED will stop pulsing, and the Tower will ready to listen to the TDE.

If the LED is done pulsing, it means that all ten values have been taken. Try moving the sensor around while recording values, to get some nice variety in the readings. It is important to note that when data is being captured, it will NOT be printed to the Command Center.

To plot the captured data, switch to the "Graph" panel by clicking on the appropriately named tab at the top of the window. The TDE should now look like this *(Figure A.14)*:

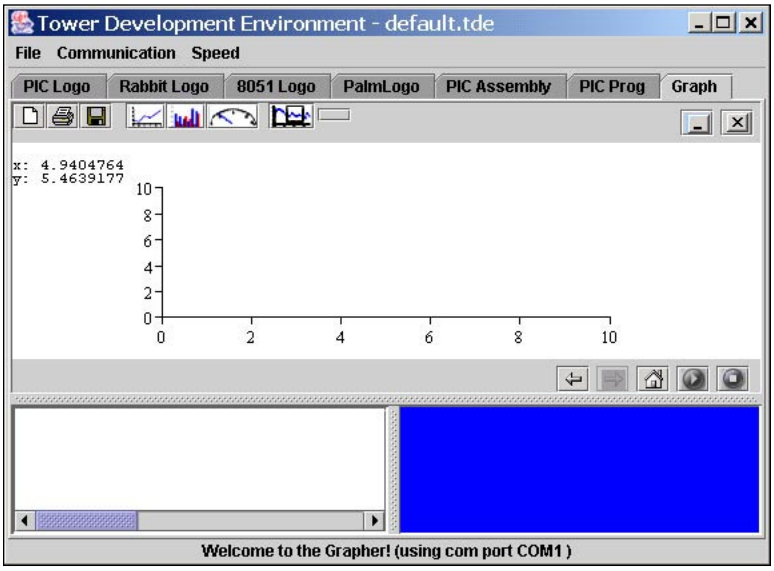


Figure A.14 - Ready to graph data.

Now, in the lower left text box in the window, type the following command and press the <Enter> key to plot the data we have just captured:

```
add-dataset "|Light Sensor| captured-data
```

In this command, the string "Light Sensor" will be used to mark the data on the plot, so that multiple sets of data can be plotted together. The value of the string can be anything you want.

After plotting the captured data, the TDE window should resemble the following *(Figure A.15)*:

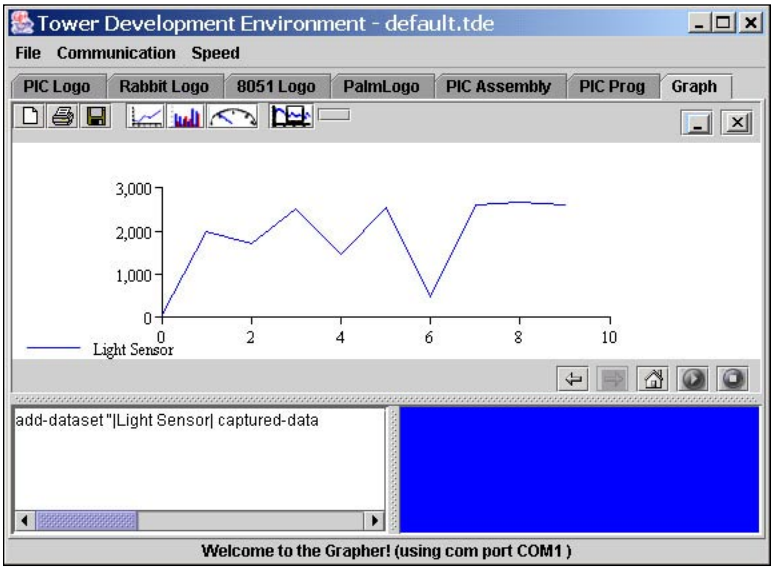


Figure A.15 - Plotting captured sensor data.

Now, let's capture more data and plot it alongside the first set of values. Switch back to the PIC Logo panel by clicking on the corresponding tab. After switching between panels, it is necessary to download the Logo program code to the Tower again before being able to use the Command Center. In order for the command center to work properly, it has to know the precise memory location of every procedure in the processor's memory. Requiring the code to be re-downloaded each time tabs are switched or the TDE is restarted ensures that there won't be any confusion between what the TDE thinks is on a Foundation, and what actually is. Also, the "Capture Data" box needs to be checked again. After the Logo program has been compiled and downloaded, run the following line of code in the Command Center again to take another ten sensor data values:

```
repeat 10 [print sensor 5 wait 10]
```

Just like before, the foundation's power LED will pulse until all data points have been taken. Once this process is complete, we can switch back to the "Graph" tab to plot the newly captured data. In the lower left text of the graphing window, type the following command and pressing the "Enter" key:

```
add-dataset "|More Light Sensor| captured-data
```

This time, we're going to name our data "More Light Sensor" to distinguish it from that taken before, but any other name could be used.

After plotting the second set of data, the TDE window should look like this *(Figure A.16)*:

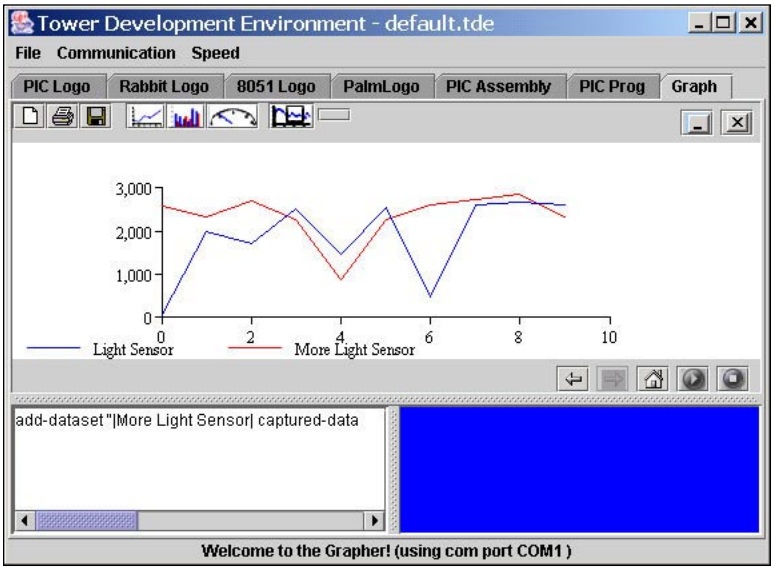


Figure A.16 - Plotting a second set of captured data.

Of course, your data will probably be quite different from that shown here, but notice that there are two separate data lines and that each is associated with a name label directly underneath the graph.

Now What?

This tutorial has covered just the basics of setting up and using the Tower. There is a wealth of information on programming Tower system devices in both Logo and Assembly that can be found in other appendices in this document. Full technical documentation is also provided for every layer currently available in the system.

As new modules for the Tower system are constantly being developed, the Tower website is a valuable resource for obtaining documentation for new modules as they are released:

- http://gig.media.mit.edu/projects/tower

Additionally, there is an online Tower support forum frequented by the Tower system developers and users:

- http://tower-support.media.mit.edu

The forum is a valuable resource for anyone using the system and wishing to obtain help or share ideas with other users around the world.

Appendix B: PIC Logo Language Reference

PIC Logo is the programming environment for the PIC foundation and the PIC LogoChips based on a PIC16F87x microcontroller. PIC Logo supports:

- The ability to directly write and read all microcontroller registers
- Control structures like if, repeat, wait, waituntil and loop
- Global and local variables
- Procedure definition with inputs and return values
- A multitasking **when** primitive
- A 16-bit number system (addition, subtraction, multiplication, division, comparison);
- Timing functions and a random number generator

When using PIC Logo, user programs are entered on a desktop computer and compiled into tokens that are transferred to the Tower through the serial port of the host computer. Logo commands can be executed by typing a line in the "command center" under the PIC Logo tab in the Tower Development Environment and pressing the <ENTER> key.

PIC Logo is a procedural language; procedures are defined using Logo **to** and **end** syntax:

```
to <procedure-name>
    <procedure-body>
end
```

User defined procedures are downloaded to a PIC foundation or Logo Chip by clicking the "Select File" under the PIC Logo tab of the Tower Development Environment, choosing the text file where these procedures are defined, and then clicking on "Compile & Download File".

Procedures can also be brought in from another file with "include". For example, in the PIC Logo test.pl file, the procedures from the standard.inc file are included with the following line:

```
include pic/include/standard.inc
```

Note that the full path name relative to the folder in which the TDE resides. Using several includes, procedures from multiple text files can be incorporated into one document.

What follows, is a list of all primitives currently supported in the PIC Logo Virtual Machine. The primitives are organized by their functionality, for ease of reference when the exact name is not known. The list below represents the order in which their detailed descriptions and examples of use can be found on the subsequent pages:

System Commands

Control Structures

Pin Control

Communication

Arithmetic and Logic

constants

System Commands

Description

Declares constant variables that are replaced with their values by the compiler at compile time.

Usage Format

```
constants [constant-list]
```

Note: The declarative **constants** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
constants [[a 2][b 3]]

print a
> 2
```

globals

System Commands

Description

Declares global variables that can be used anywhere in the scope of the user program. Globals can be set and read either directly or by their memory location.

Usage Format

```
globals [global-list]
```

Up to 96 global variables can be defined.

Note: The declarative **globals** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
globals [foo [bar 4]]

setfoo 5

print foo
> 5
```

In the case above, five global variables are defined. In the argument to the **globals** statement, two items are given. The first item simply creates a global variable called "foo". The second item creates four global variables, with "bar" representing the name associated with the first of the four. Defining globals in this manner can be useful when arrays are being created. Although the additional global variables are not directly named, they can be accessed by their memory locations using the **set-global** and **get-global** primitives.

set-global

System Commands

Description

Sets the value of a global variable when given a pointer to its memory location.

Usage Format

```
set-global pointer new-value
```

The argument **pointer** is the memory pointer to the global and the argument **new-value** is the new value to be assigned to the global variable.

This function is very useful for manipulating global variables as arrays, using memory pointers to dynamically access different locations in them.

In most cases, it is easier to use **set** and the name of the global to set the value of the global, as explained in the definition of **globals**.

The primitive **set-global** itself is only useful when you use pointers to globals. This is done using macros automatically defined when you create a global variable. For example, if you define a global using **globals [foo]** then the macro ***foo** is defined as a pointer to the global variable.

Globals are stored sequentially as they are declared, two bytes apiece. Thus the pointers are also sequential, increasing by two each time.

Sample Code and Output

```
globals [foo bar]

set-global *foo 14

print foo
> 14
```

get-global

System Commands

Description

Obtains the value of a global variable when given a pointer to its memory location.

Usage Format

```
get-global pointer
```

The argument **pointer** is the memory pointer to the global to be read.

This function is very useful for manipulating global variables as arrays, using memory pointers to dynamically access different locations in them.

In most cases, it is easier to use just the name of the global itself to obtain its value, as explained in the definition of **globals**.

The primitive **get-global** itself is only useful when you use pointers to globals. This is done using macros automatically defined when you create a global variable. For example, if you define a global using **globals [foo]** then the macro ***foo** is defined as a pointer to the global variable. Globals are stored sequentially as they are declared, two bytes apiece. Thus the pointers are also sequential, increasing by two each time.

Sample Code and Output

```
globals [foo bar]

set-global *foo + 2 1234

print get-global *foo + 2
> 1234
```

set-global-byte

System Commands

Description

Sets the value of a single byte of a global variable when given a pointer to its memory location.

Usage Format

```
set-global-byte pointer new-value
```

The argument **pointer** is the memory pointer to the global and the argument **new-value** is the new value to be assigned to the global variable.

This primitive gives low-level access to each byte of a two-byte global variable. This function is very useful for when you want to create a variable string, with values ranging only from 0 to 255. If an actual array of larger numbers is desired, **set-global** should be used.

The primitive **set-global-byte** itself is only useful when you use pointers to globals. This is done using macros automatically defined when you create a global variable. For example, if you define a global using **globals [foo]** then the macro ***foo** is defined as a pointer to the global variable. Globals are stored sequentially as they are declared, two bytes apiece. Thus the pointers are also sequential, increasing by two each time.

Sample Code and Output

```
globals [foo bar]

set-global-byte *foo 14

print foo
> 14
```

get-global-byte

System Commands

Description

Obtains the value of a single byte of a global variable when given a pointer to its memory location.

Usage Format

```
get-global-byte pointer
```

The argument **pointer** is the memory pointer to the global to be read.

This primitive gives low-level access to each byte of a two-byte global variable. This function is very useful for when you want to create a variable string, with values ranging only from 0 to 255. If an actual array of larger numbers is desired, **get-global** should be used.

The primitive **get-global-byte** itself is only useful when you use pointers to globals. This is done using macros automatically defined when you create a global variable. For example, if you define a global using **globals [foo]** then the macro ***foo** is defined as a pointer to the global variable. Globals are stored sequentially as they are declared, two bytes apiece. Thus the pointers are also sequential, increasing by two each time.

Sample Code and Output

```
globals [foo bar]

set-global-byte *foo + 2 123

print get-global-byte *foo + 2
> 123
```

table

System Commands

Description

Defines a table of values which can be indexed and looked up for various applications.

Usage Format

```
table name [item-list]
```

Independent table item values can be read using the **table-item** function defined in the standard includ file.

Note: The declarative **table** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
table foo [3 7 4 "cat 2]

print table-item foo 0
> 3

print table-item foo 2
> 4

print-string table-item foo 4
> cat
```

read-prog-mem

System Commands

Description

Reads and returns a byte from the program Flash EEPROM.

Usage Format

```
read-prog-mem address
```

The argument **address** is the program memory location from which we want to read.

Sample Code and Output

```
to print-string :n
    setnn :n
    loop
    [
        if (read-prog-mem nn) = 0 [stop]
        put-serial read-prog-mem nn
        setnn nn + 1
    ]
end
```

write-reg

System Commands

Description

Short for "write-register", writes to an internal PIC register.

Usage Format

```
write-reg address value
```

The argument **address** is the address of a PIC register to which we want to write and **value** is the value to write.

Sample Code and Output

```
to ad :chan
    write-reg adcon1 $80
    write-reg adcon0 ((:chan - 1) * 8) + $81
    set-bit adgo adcon0
    waituntil [not test-bit adgo adcon0]
    output ((read-reg adresh) * 256) +
             read-reg adresl
end
```

read-reg

System Commands

Description

Short for "read-register", examines and returns the content of one of the internal PIC registers.

Usage Format

```
read-reg address
```

The argument **address** is the address of the PIC register we want to read.

Sample Code and Output

```
print read-reg 5
> (the conent of register with address 5)
```

resett

System Commands

Description

Resets the value of the internal PIC timer.

Usage Format

```
resett
```

Note: Affects only the value that the timer primitive reports.

Sample Code and Output

```
resett wait 10 print timer
> 1000
```

timer

System Commands

Description

Returns the current value of the timer.

Usage Format

```
timer
```

Note: The timer overflows after 32767 milliseconds and resets to zero. (It never has a negative value.)

Sample Code and Output

```
resett wait 10 print timer
> 1000
```

on-white-button

System Commands

Description

Declares what commands should run when the white button on the Foundation is pressed.

Usage Format

```
on-white-button [command-list]
```

Note: The declarative **on-white-button** *should only be used along with procedure definitions in the source code and cannot be used in the command-center. If the Tower is already running a program (indicated by a pulsating blue LED), pressing the white button stops the program. When the white button is pressed for a second time, the Tower runs the list of commands declared by the* **on-white-button** *declarative.*

Sample Code and Output

```
on-white-button [your-favorite-procedure]
```

on-startup

System Commands

Description

Declares what commands should run when the Tower is first turned on or power-cycled.

Usage Format

```
on-startup [command-list]
```

Note: The declarative **on-startup** *should only be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
on-startup [your-favorite-startup-procedure]
```

ignore

System Commands

Description

Catches a value and throws it away.

Usage Format

```
ignore value
```

The ignore primitive is used to ignore a data byte in a case where a value being returned from a function must be caught by something, but is not actually needed.

Sample Code and Output

```
ignore read-reg 30
```

nop

System Commands

Description

Short for "no-operation," essentially used to create short processor delays.

Usage Format

```
nop
```

Sample Code and Output

```
repeat 100 [nop]
```

if

Control Stuctures

Description

Executes a block of code if a condition expression is non-zero.

Usage Format

```
if (conditional-expression)
   [code-block]
```

The **conditional-expression** is the expression whose value determines whether the code block is evaluated and **code-block** is the block of code to be evaluated.

Note: This primitive only actually checks the low byte of the result of the conditional expression (See the second sample code below.)

Sample Code and Output

```
if 42 [print-string "Hi! cr]
> Hi!

if $4000 [print-string "Hi! cr]
>

if 0 [print-string "Hi! Cr]
>
```

ifelse

Control Stuctures

Description

Executes one of two blocks of code, depending on an expression's results. If the expression is non-zero, the first block of code is evaluated. If not, the second block of code is executed.

Usage Format

```
ifelse (conditional-expression)
       [code-block-ture]
       [code-block-false]
```

The **conditional-expression** is the expression whose value determines which code block is evaluated. The **code-block-true** is the block of code to be evaluated if the condition is true and the **code-block-false** is the block of code to be evaluated if the condition is false.

Note: As for the **if** *primitive, this primitive only checks the low byte of its conditional expression.*

Sample Code and Output

```
ifelse (0 = 1) [print 1][print 0]
> 0

ifelse (1 = 1) [print 1][print 0]
> 1

ifelse ($4000) [print 1][print 0]
> 0

ifelse (42) [print 1][print 0]
> 1

ifelse (0) [print 1][print 0]
> 0
```

loop

Control Stuctures

Description

Repeats a block of code forever.

Usage Format

```
loop [code-block]
```

The **code-block** is the block of code to be repeated forever.

Note: The loop can be broken by a **stop**, **stop!**, *or* **output** *primitive.*

Sample Code and Output

```
to mwait :msecs
    resett
    loop
    [
        if (timer > :msecs) [stop]
    ]
end
```

output

Control Stuctures

Description

Exits the currently running procedure and returns a value.

Usage Format

```
output value
```

The **value** argument is the desired return value.

Sample Code and Output

```
to add-numbers :x :y
    output :x + :y
end
```

repeat

Control Stuctures

Description

Repeats a block of code for a given number of times.

Usage Format

```
repeat count [code-block]
```

The **code-block** is the block of code to be repeated and **count** is the number of times to repeat.

Sample Code and Output

```
repeat 2 + 2 [print-string "Hi! send 10]
> Hi!
> Hi!
> Hi!
> Hi!
```

stop!

Control Stuctures

Description

Halts the virtual machine and whatever program is running.

Usage Format

```
stop!
```

Sample Code and Output

```
if (test-bit 3 porta) [stop!]
```

stop

Control Stuctures

Description

Immediately exits the currently running procedure.

Usage Format

```
stop
```

Sample Code and Output

```
to mwait :msecs
    resett
    loop
    [
        if (timer > :msecs) [stop]
    ]
end
```

wait

Control Stuctures

Description

Waits for a specified number of tenths of a second. The process started by the when primitive continues to be checked during the wait.

Usage Format

```
wait duration
```

The **duration** argument is the number of tenths of seconds to wait.

Sample Code and Output

```
wait 10
```

waituntil

Control Stuctures

Description

Repeatedly checks a condition until it becomes true, at which point it continues with the subsequent commands in the program.

Usage Format

```
waituntil [condition]
```

Sample Code and Output

```
to mwait :msecs
    resett
    waituntil [timer > :msecs]
end
```

when

Control Stuctures

Description

Starts a background process which checks a condition between each line of logo code running in the main process. If the condition becomes true, a specific block of code is executed.

Usage Format

```
when [condition] [code-block]
```

The **condition** is what is checked in the background and **code-block** is what is executed every time the condition becomes true.

Note: the **when** *condition is edge-triggered. That is, if the condition becomes true and remains true after the code is executed, the code will not be executed again until the condition becomes false and then true again.*

Sample Code and Output

```
when [test-bit 3 porta]
    [print-string "|Switch pressed!|]
```

when-off

Control Stuctures

Description

Stops a previously started when process.

Usage Format

```
when-off
```

Sample Code and Output

```
when-off
```

set-bit

Pin Control

Description

Sets a bit on an internal PIC register.

Usage Format

```
set-bit bit-number register
```

The argument **bit-number** is the bit number (0-7) of the PIC register, with address **register**, that you want to set.

Note: If the register corresponds to a PIC I/O port, the bit-number would correspond to the pin number on the port.

Sample Code and Output

```
to set :chan :port
    clear-bit :chan (:port + $80)
    set-bit :chan :port
end
```

clear-bit

Pin Control

Description

Clears a bit on an internal PIC register.

Usage Format

```
clear-bit bit-number register
```

The argument **bit-number** is the bit number (0-7) of the PIC register, with address **register**, that you want to clear.

Note: If the register corresponds to a PIC I/O port, the bit-number would correspond to the pin number on the port.

Sample Code and Output

```
to clear :chan :port
    clear-bit :chan (:port + $80)
    clear-bit :chan :port
end
```

test-bit

Pin Control

Description

Tests a bit on an internal PIC register.

Usage Format

```
test-bit bit-number register
```

The argument **bit-number** is the bit number (0-7) of the PIC register, with address **register**, that you want to test.

Note: If the register corresponds to a PIC I/O port, the bit-number would correspond to the pin number on the port.

Sample Code and Output

```
print test-bit 1 5
> (prints a 1 if bit 1 of register 5 is set)
> (prints a 0 if bit 1 of register 5 is clear)
```

flip-bit

Pin Control

Description

Toggles a bit on an internal PIC register between set and clear states.

Usage Format

```
flip-bit bit-number register
```

The argument **bit-number** is the bit number (0-7) of the PIC register, with address **register**, that you want to toggle.

Note: If the register corresponds to a PIC I/O port, the bit-number would correspond to the pin number on the port.

Sample Code and Output

```
to toggle :chan :port
    clear-bit :chan (:port + $80)
    flip-bit :chan :port
end
```

pulse-out

Pin Control

Description

Generates a pulsed waveform on a given I/O pin of the PIC.

Usage Format

```
pulse-out pin-number port duration
```

The argument **pin-number** is the pin number (0-7) of the PIC I/O port, with address **port**, that you want to pulse. The **duration** argument is the length of the pulse, in tens of micro-seconds.

Sample Code and Output

```
to pulse-train :n
    loop
    [
        pulse-out 0 portb 100
        wait 1
    ]
end
```

new-serial?

Communication

Description

Returns true if a byte has been received on the serial port since the last time the **get-serial** command was issued.

Usage Format

```
new-serial?
```

Sample Code and Output

```
waituntil [new-serial?] print get-serial
```

get-serial

Communication

Description

Returns the last byte received on the serial port.

Usage Format

```
get-serial
```

Sample Code and Output

```
waituntil [new-serial?] print get-serial
```

put-serial

Communication

Description

Sends a byte over the serial port.

Usage Format

```
put-serial value
```

The argument **value** is the byte to send over serial.

Note: When the byte is sent back to the TDE, it will print the ASCII character corresponding to the byte sent.

Sample Code and Output

```
put-serial 65
> A
```

set-baud-2400

Communication

Description

Sets the serial communication rate to 2400 bps.

Usage Format

```
set-baud-2400
```

Note: This is the default baud rate for serial communication.

Sample Code and Output

```
set-baud-2400
```

set-baud-9600

Communication

Description

Sets the serial communication rate to 9600 bps.

Usage Format

```
set-baud-9600
```

Note: The default baud rate for serial communication is 2400 bps.

Sample Code and Output

```
set-baud-9600
```

i2c-start

Communication

Description

Starts an I²C communication sequence.

Usage Format

```
i2c-start
```

Note: This primitive works only on I²C master devices, such as the PIC Foundation.

Sample Code and Output

```
to turn-servo :number :angle
    i2c-start
    i2c-write-byte $04
    i2c-write-byte 2
    i2c-write-byte (:number % 9) - 1
    i2c-write-byte :angle
    i2c-stop
end
```

i2c-write-byte

Communication

Description

Writes a byte to the I²C bus.

Usage Format

```
i2c-write-byte value
```

The **value** argument is the value to be sent over the bus. The value itself represents different things based on its location after the **i2c-start** command. Following the example code below, the first **i2c-write-byte** command takes the address of the I²C slave device as an argument. The next argument to the **i2c-write-byte** command represents the number of arguments to follow, while each subsequent **i2c-write-byte** command, prior to an **i2c-stop** command, sends an argument itself, and depends on the functionality of the I2C slave device being communicated with. All I2C slave addresses are even numbers. Adding "!" to the address, making it odd, indicates that a data-read operation is being initiated, as opposed to a data-write.

Note: This primitive works only on I²C master devices, such as the PIC Foundation.

Sample Code and Output

```
to turn-servo :number :angle
    i2c-start
    i2c-write-byte $04
    i2c-write-byte 2
    i2c-write-byte (:number % 9) - 1
    i2c-write-byte :angle
    i2c-stop
end
```

i2c-read-byte

Communication

Description

Reads a byte from the I^2C bus.

Usage Format

```
i2c-read-byte last-byte
```

The **last-byte** argument is used to indicate whether or not a byte being read from the slave device will be the last in the given read sequence. A "0" should be used when the last byte is being requested, and a "1" for all other read operations.

Note: This primitive works only on I^2C master devices, such as the PIC Foundation.

Sample Code and Output

```
to sensor :n
    i2c-start
    i2c-write-byte $0a
    i2c-write-byte 2
    i2c-write-byte 0
    i2c-write-byte (:n - 1)
    i2c-stop
    i2c-start
    i2c-write-byte $0b
    ignore i2c-read-byte 1
    seti2c-byte (lsh i2c-read-byte 1 8)
    seti2c-byte i2c-byte or i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

i2c-check

Communication

Description

Checks to see if a board is present at a given address.

Usage Format

```
i2c-check address
```

The **address** argument is the I²C slave device address to be checked. It should be an even value from 2 to 126. The function returns a "1" if the device is present, and a "0" if it is not.

Note: This primitive works only on I²C master devices, such as the PIC Foundation.

Sample Code and Output

```
to test-for-board :address
    i2c-start
    seti2c-byte i2c-check :address
    i2c-stop
    output i2c-byte
end
```

i2c-stop

Communication

Description

Terminates I²C communication.

Usage Format

```
i2c-stop
```

In addition to being a use-available primitive, **i2c-stop** is automatically called after every program terminates, to ensure that no slave devices are left in a bad state if a program ends abnormally.

Note: This primitive works only on I²C master devices, such as the PIC Foundation.

Sample Code and Output

```
to turn-servo :number :angle
    i2c-start
    i2c-write-byte $04
    i2c-write-byte 2
    i2c-write-byte (:number % 9) - 1
    i2c-write-byte :angle
    i2c-stop
end
```

new-i2c?

Communication

Description

Returns true if a byte has been received on the serial port since the last time the **get-serial** command was issued.

Usage Format

```
new-i2c?
```

Note: This primitive works only on I²C slave devices, such as a Tower Layer.

Sample Code and Output

```
waituntil [new-i2c?] print get-i2c 0
```

get-i2c

Communication

Description

Returns the specified byte received in the most recent data packet received over the I²C bus.

Usage Format

```
get-i2c index
```

The **index** argument corresponds to the buffer location to be read. The contents of the "0" location represents the number of arguments to follow in the remainder of the buffer.

Note: This primitive works only on I²C slave devices, such as a Tower Layer.

Sample Code and Output

```
waituntil [new-i2c?] print get-i2c 0
```

put-i2c

Communication

Description

Deposits a byte in the buffer for transmit over the I²C bus.

Usage Format

```
put-i2c index value
```

The **index** argument corresponds to the buffer location to be written to, and the **value** argument is the value to be written. The contents of the "0" location represents the number of arguments to follow in the remainder of the buffer. The **i2c-ready** function must be called to indicate that transmit data is stable and ready before it will actually be sent back to the master device.

Note: This primitive works only on I²C slave devices, such as a Tower Layer.

Sample Code and Output

```
to respond
    waituntil [new-i2c?]
    if ((get-i2c 1) = 1)
    [
        put-i2c 1 23
        put-i2c 0 1
        ready-i2c
    ]
end
```

ready-i2c

Communication

Description

Indicates that I²C transmit data is stable and ready to be returned to the master device..

Usage Format

```
ready-i2c
```

This function should be used as soon as all of the desired data values in the I²C trasmit buffer have been stored, including the "0" location indicating the total number of bytes to follow.

Note: This primitive works only on I²C slave devices, such as a Tower Layer.

Sample Code and Output

```
to respond
    waituntil [new-i2c?]
    if ((get-i2c 1) = 1)
    [
        put-i2c 1 23
        put-i2c 0 1
        ready-i2c
    ]
end
```

arithmetic (+, -, *, /, %)

Arithmetic and Logic

Description

Returns the result of arithmetical operators (+, -, *, /, %) applied to 16-bit numerical operands.

Usage Format

```
num1 (+, -, *, /, %) num2
```

Note: The "/" operator returns the integer part of the result if dividing num1 by num2. The "%" operator returns the remainder of a dividing num1 by num2.

Sample Code and Output

```
print 4 + -10
> -6

print 42 - 8
> 34

print 4 * -10
> -40

print 35 / 10
> 3

print 35 % 10
> 5
```

comparisons (=, >, <)

Arithmetic and Logic

Description

Returns a Boolean depending on the result of arithmetical comparison operators (=, <, >) applied to 16-bit numerical operands.

Usage Format

```
num1 (=, >, <) num2
```

Sample Code and Output

```
print 10 = 10
> 1

print 4 > 2
> 1

print 4 < 2
> 0
```

bit-wise logic (and, or, xor)

Arithmetic and Logic

Description

Returns the result of bit-wise logical operators (and, or, xor) applied to 16-bit numerical operands.

Usage Format

```
num1 (and, or, xor) num2
```

Sample Code and Output

```
print 20 and 4
> 4

print 16 or 4
> 20

print 7 xor 3
> 4

print (1 = 1) and (1 = 0)
> 0

print (1 = 1) or (1 = 0)
> 1

print (1 = 1) or (1 = 0)
> 1
```

not

Arithmetic and Logic

Description

Returns a Boolean not of a number.

Usage Format

```
not num
```

Note: This operation is a Boolean not and not a bit-wise not.

Sample Code and Output

```
print not 42
> 0

print not 0
> 1
```

left-shift

Arithmetic and Logic

Description

Used to left shift or right shift a number by a specified number of bits.

Usage Format

```
left-shift num dist
```

The argument **num** is the number to shift and **dist** is the number of bits by which to shift num.

Note: A negative value for **dist** *represents a right shift, while a positive value for* **dist** *represents a left shift.*

Sample Code and Output

```
print lsh 4 1
> 8

print lsh 64 -2
> 16
```

high-byte

Arithmetic and Logic

Description

Returns the high byte of a 16 bit value.

Usage Format

```
high-byte num
```

Sample Code and Output

```
print high-byte $4000
> 64
```

low-byte

Arithmetic and Logic

Description

Returns the low byte of a 16 bit value.

Usage Format

```
low-byte num
```

Sample Code and Output

```
print low-byte $6560
> 96
```

random

Arithmetic and Logic

Description

Returns a random number between 0 and 32767, inclusive.

Usage Format

```
random
```

Sample Code and Output

```
print random
> 5622

print random % 256
> 142
```

set-random-seed

Arithmetic and Logic

Description

Sets the seed for the random number generator with a given 16-bit value

Usage Format

```
set-random-seed value
```

The random number generator is normally seeded automatically off of the internal timer. However, if **on-starup** is used, the timer will always be the same value when seeded. Usually, if setting the seed manually, one would want to set it to a relatively unpredictable value, such as a floating sensor value.

Sample Code and Output

```
set-random-seed ad 1

print random
> 9460
```

The **ad** function is used to return a value corresponding to the analog voltage on a processor pin. If the pin is left unconnected, its voltage will float, providing a good random number seed. The function is defined in the standard include file.

Appendix C: Rabbit Logo Language Reference

Rabbit Logo is the programming environment for the Rabbit foundation based on a Rabbit 2200 microcontroller. Rabbit Logo supports:

- The ability to directly write and read all microcontroller registers
- Control structures like if, repeat, wait, waituntil and loop
- Global and local variables
- Array and string allocations
- Procedure definition with inputs and return values
- Up to 20 simultaneous processor threads with multitasking
- A 32-bit number system (addition, subtraction, multiplication, division, comparison);
- Timing functions and a random number generator

When using Rabbit Logo, user programs are entered on a desktop computer and compiled into tokens that are transferred to the Tower through the serial port of the host computer. Logo commands can be executed by typing a line in the "command center" under the Rabbit Logo tab in the Tower Development Environment and pressing the <ENTER> key.

Rabbit Logo is a procedural language; procedures are defined using Logo **to** and **end** syntax:

```
to <procedure-name>
    <procedure-body>
end
```

User defined procedures are downloaded to a Rabbit foundation by clicking the "Select File" under the Rabbit Logo tab of the Tower Development Environment, choosing the text file where these procedures are defined, and then clicking on "Compile & Download File".

Procedures can also be brought in from another file with "include". For example, in the Rabbit Logo test.pl file, the procedures from the standard.inc file are included with the following line:

```
include rabbit/include/standard.inc
```

Note that the full path name relative to the folder in which the TDE resides. Using several includes, procedures from multiple text files can be incorporated into one document.

What follows, is a list of all primitives currently supported in the Rabbit Logo Virtual Machine. The primitives are organized by their functionality, for ease of reference when the exact name is not known. The list below represents the order in which their detailed descriptions and examples of use can be found on the subsequent pages:

System Commands

Control Structures

Strings and Arrays

Pin Control

Communication

TCP/IP and Sockets

Arithmetic and Logic

constants

System Commands

Description

Declares constant variables that are replaced with their values by the compiler at compile time.

Usage Format

```
constants [constant-list]
```

Note: The declarative **constants** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
constants [[a 2][b 3]]

print a
> 2
```

globals

System Commands

Description

Declares global variables that can be used anywhere in the scope of the user program. Globals can be set and read either directly or by their memory location.

Usage Format

```
globals [global-list]
```

Up to 128 global variables can be defined.

Note: The declarative **globals** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
globals [foo bar]

setfoo 5

print foo
> 5
```

In the case above, two global variables are defined. Any global variable can be accessed directly using "set" and its name, or by its memory location using the **set-global** and **get-global** primitives.

set-global

System Commands

Description

Sets the value of a global variable when given a pointer to its memory location.

Usage Format

```
set-global pointer new-value
```

The argument **pointer** is the memory pointer to the global and the argument **new-value** is the new value to be assigned to the global variable.

This function is very useful for allowing procedures to dynamically change the values of different global variables.

In most cases, it is easier to use **set** and the name of the global to set the value of the global, as explained in the definition of **globals**.

The primitive **set-global** itself is only useful when you use pointers to globals. This is done using macros automatically defined when you create a global variable. For example, if you define a global using **globals [foo]** then the macro ***foo** is defined as a pointer to the global variable. Globals are stored sequentially as they are declared, 32-bits apiece. Thus the pointers are also sequential, increasing by one each time.

Sample Code and Output

```
globals [foo bar]

set-global *foo 14

print foo
> 14
```

get-global

System Commands

Description

Obtains the value of a global variable when given a pointer to its memory location.

Usage Format

```
get-global pointer
```

The argument **pointer** is the memory pointer to the global to be read.

This function is very useful for allowing procedures to dynamically change the values of different global variables.

In most cases, it is easier to use just the name of the global itself to obtain its value, as explained in the definition of **globals**.

The primitive **get-global** itself is only useful when you use pointers to globals. This is done using macros automatically defined when you create a global variable. For example, if you define a global using **globals [foo]** then the macro ***foo** is defined as a pointer to the global variable. Globals are stored sequentially as they are declared, 32-bits apiece. Thus the pointers are also sequential, increasing by one each time.

Sample Code and Output

```
globals [foo bar]

set-global *foo + 1 1234

print get-global *foo + 1
> 1234
```

table

System Commands

Description

Defines a table of values which can be indexed and looked up for various applications.

Usage Format

```
table name [item-list]
```

The **aget** primitive is used to read individual table item values.

Note: The declarative **table** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
table foo [3 7 4 "cat 2]

print aget foo 0
> 3

print aget foo 2
> 4

print-string aget foo 4
> cat
```

strings

System Commands

Description

Declares string variables that can be used anywhere in the scope of the user program. Any element in a string can be set or read independently, and there are a variety of primitives available for common string operations such as copy and compare. Each element in a string is an 8-bit value.

Usage Format

```
strings [string-list]
```

Note: The declarative **strings** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
strings [[foo 10][bar 5]]

strcpy foo "hello

print-string foo
> hello
```

When a string is allocated, it is specified with a maximum size. In the above example, the string "foo" is ten bytes long, and the string "bar" is five bytes.

flash-strings

System Commands

Description

Declares string variables in flash memory that can be used anywhere in the scope of the user program, and are permanently stored during power-off. Elements in a flash string cannot be set or read independently, but there are a variety of primitives available for common string operations such as copy and compare. Each element in a string is an 8-bit value.

Usage Format

```
flash-strings [string-list]
```

Note: The declarative **flash-strings** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
flash-strings [[flash-foo 10][flash-bar 5]]

strcpy flash-foo "hello

print-string flash-foo
> hello
```

When a string is allocated, it is specified with a maximum size. In the above example, the string "foo" is ten bytes long, and the string "bar" is five bytes.

arrays

System Commands

Description

Declares array variables that can be used anywhere in the scope of the user program. Any element in an array can be set or read independently. Each element in an array is a 32-bit value.

Usage Format

```
arrays [array-list]
```

Note: The declarative **arrays** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
arrays [[foo 10][bar 5]]

aset foo 1 23

print aget foo 1
> 23
```

When an array is allocated, it is specified with a maximum size. In the above example, the array "foo" is ten bytes long, and the array "bar" is five bytes.

flash-arrays

System Commands

Description

Declares array variables in flash memory that can be used anywhere in the scope of the user program, and are permanently stored during power-off. Elements in flash arrays cannot be set or read independently. Thei value should be stored by copying the contents of a non-flash array using the **strcpy** function. Each element in an array is a 32-bit value.

Usage Format

```
flash-arrays [array-list]
```

Note: The declarative **flash-arrays** *should be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
arrays [[foo 10][bar 5]]
flash-arrays [[flash-foo 10][flash-bar 5]]

aset foo 1 23

strcpy flash-foo foo

strcpy foo flash-foo

print aget foo 1
> 23
```

When an array is allocated, it is specified with a maximum size. In the above example, the arrays "foo" and "flash-foo" are ten bytes long, and the arrays "bar" and "flash-bar" are five bytes.

let

System Commands

Description

Declares local variables that can be used only within the scope of a procedure.

Usage Format

```
let [local-variable-list]
```

Note: The primitive **let** *can only be used within a procedure definition in the source code and cannot be used in the command center.*

Sample Code and Output

```
to sample
    let [my-n 0 my-m 23]
    make "my-n 15
    output :my-n
end
```

Each local variable must be set to an initial value. In the above example, the variable "my-n" is set to 0, and the variable "my-m" is set to 23. Changing the value of a local variable can only be accomplished by using the make primitive, followed by a double quoation mark and the name of the variable, and then the new value to set it to. Reading the value of a local variable is accomplished by placing a colon before the name of the variable, as shown.

make

System Commands

Description

Sets the value of a local variable defined within a procedure.

Usage Format

```
make "variable-name value
```

The **variable-name** argument is the name of the target local variable, and the **value** is the number to set it to.

Note: The primitive **make** *can only be used within a procedure definition in the source code and cannot be used in the command center.*

Sample Code and Output

```
to sample
    let [my-n 0 my-m 23]
    make "my-n :my-m + 1
    output :my-n
end
```

The double quotation mark is necessary before the variable name, since the **make** primtive treats the supplied name as a string.

write-rtc

System Commands

Description

Writes a value to the Rabbit processor module's built-in 128-day real-time-clock.

Usage Format

```
write-rtc value
```

The **value** argument is the number to write into the clock register.

Sample Code and Output

```
write-rtc 10  wait 20  print read-rtc
> 12
```

read-rtc

System Commands

Description

Reads the value of the Rabbit processor module's built-in 128-day real-time-clock.

Usage Format

```
read-rtc
```

The **value** argument is the number to write into the clock register.

Sample Code and Output

```
write-rtc 10  wait 20  print read-rtc
> 12
```

resett

System Commands

Description

Resets the value of the internal Rabbit timer.

Usage Format

```
resett
```

Note: Affects only the value that the timer primitive reports.

Sample Code and Output

```
resett wait 10 print timer
> 1000
```

timer

System Commands

Description

Returns the current value of the timer.

Usage Format

```
timer
```

Note: The timer overflows after about 24 days and resets to zero. (It never has a negative value.)

Sample Code and Output

```
resett wait 10 print timer
> 1000
```

on-white-button

System Commands

Description

Declares what commands should run when the white button on the foundation is pressed.

Usage Format

```
on-white-button [command-list]
```

Note: The declarative **on-white-button** *should only be used along with procedure definitions in the source code and cannot be used in the command-center. If the Tower is already running a program (indicated by a pulsating blue LED), pressing the white button stops the program. When the white button is pressed for a second time, the Tower runs the list of commands declared by the* **on-white-button** *declarative.*

Sample Code and Output

```
on-white-button [your-favorite-procedure]
```

on-startup

System Commands

Description

Declares what commands should run when the Tower is first turned on or power-cycled.

Usage Format

```
on-startup [command-list]
```

Note: The declarative **on-startup** *should only be used along with procedure definitions in the source code and cannot be used in the command-center.*

Sample Code and Output

```
on-startup [your-favorite-startup-procedure]
```

ignore

System Commands

Description

Catches a value and throws it away.

Usage Format

```
ignore value
```

The ignore primitive is used to ignore a data byte in a case where a value being returned from a function must be caught by something, but is not actually needed.

Sample Code and Output

```
ignore read-reg 30
```

if

Control Stuctures

Description

Executes a block of code if a condition expression is non-zero.

Usage Format

```
if (conditional-expression)
   [code-block]
```

The **conditional-expression** is the expression whose value determines whether the code block is evaluated and **code-block** is the block of code to be evaluated.

Sample Code and Output

```
if 42 [print-string "Hi! cr]
> Hi!

if $4000 [print-string "Hi! cr]
> Hi!

if 0 [print-string "Hi! Cr]
>
```

ifelse

Control Stuctures

Description

Executes one of two blocks of code, depending on an expression's results. If the expression is non-zero, the first block of code is evaluated. If not, the second block of code is executed.

Usage Format

```
ifelse (conditional-expression)
       [code-block-ture]
       [code-block-false]
```

The **conditional-expression** is the expression whose value determines which code block is evaluated. The **code-block-true** is the block of code to be evaluated if the condition is true and the **code-block-false** is the block of code to be evaluated if the condition is false.

Sample Code and Output

```
ifelse (0 = 1) [print 1][print 0]
> 0

ifelse (1 = 1) [print 1][print 0]
> 1

ifelse ($4000) [print 1][print 0]
> 1

ifelse (42) [print 1][print 0]
> 1

ifelse (0) [print 1][print 0]
> 0
```

loop

Control Stuctures

Description

Repeats a block of code forever.

Usage Format

```
loop [code-block]
```

The **code-block** is the block of code to be repeated forever.

Note: The loop can be broken by a **stop**, **stop!**, *or* **output** *primitive.*

Sample Code and Output

```
to mwait :msecs
    resett
    loop
    [
        if (timer > :msecs) [stop]
    ]
end
```

output

Control Stuctures

Description

Exits the currently running procedure and returns a value.

Usage Format

```
output value
```

The **value** argument is the desired return value.

Sample Code and Output

```
to add-numbers :x :y
    output :x + :y
end
```

repeat

Control Stuctures

Description

Repeats a block of code for a given number of times.

Usage Format

```
repeat count [code-block]
```

The **code-block** is the block of code to be repeated and **count** is the number of times to repeat.

Sample Code and Output

```
repeat 2 + 2 [print-string "Hi! send 10]
> Hi!
> Hi!
> Hi!
> Hi!
```

stop

Control Stuctures

Description

Immediately exits the currently running procedure.

Usage Format

```
stop
```

Sample Code and Output

```
to mwait :msecs
    resett
    loop
    [
        if (timer > :msecs) [stop]
    ]
end
```

wait

Control Stuctures

Description

Waits for a specified number of tenths of a second. Background processes continue to be checked during the wait.

Usage Format

```
wait duration
```

The **duration** argument is the number of tenths of seconds to wait.

Sample Code and Output

```
wait 10
```

mwait

Control Stuctures

Description

Waits for a specified number of milliseconds. Background processes continue to be checked during the wait.

Usage Format

```
mwait duration
```

The **duration** argument is the number of milliseconds to wait.

Sample Code and Output

```
mwait 100
```

waituntil

Control Stuctures

Description

Repeatedly checks a condition until it becomes true, at which point it continues with the subsequent commands in the program.

Usage Format

```
waituntil [condition]
```

Sample Code and Output

```
to mwait :msecs
    resett
    waituntil [timer > :msecs]
end
```

launch

Control Stuctures

Description

Launches a new thread to run on the processor. The primitive returns a pointer to thead itself, which can later be used to terminate the thread.

Usage Format

```
launch [code-block]
```

The **code-block** argument is the code that should run in the new thread.

Sample Code and Output

```
ignore launch [loop [tcp-tick]]
```

every

Control Stuctures

Description

Launches a new thread that runs a block of code at regular specified timing intervals. The primitive returns a pointer to thead itself, which can later be used to terminate the thread.

Usage Format

```
every delay [code-block]
```

The **delay** argument is the time, in tenths of a second, between executions of the code specified in the **code-block** argument.

Sample Code and Output

```
ignore every 10 [print-string "hello cr]
> hello
> hello
> hello
...
```

when

Control Stuctures

Description

Starts a background thread which repeatedly checks a condition. If the condition becomes true, a specific block of code is executed. The primitive returns a pointer to the thread itself,which can later be used to terminate the thread.

Usage Format

```
when [condition] [code-block]
```

The **condition** is what is checked in the background and **code-block** is what is executed every time the condition becomes true.

Note: the **when** *condition is edge-triggered. That is, if the condition becomes true and remains true after the code is executed, the code will not be executed again until the condition becomes false and then true again.*

Sample Code and Output

```
ignore when [test-bit 3 porta]
    [print-string "|Switch pressed!|]
```

kill-thread

Control Stuctures

Description

Stops a previously started thread.

Usage Format

```
kill-thread thread-number
```

The argument **thread-number** is a pointer to the thread that is to be stopped. The value of the pointer is obtained when a thread is invoked.

Sample Code and Output

```
to read-sensor
    setn launch [loop [print sensor 1]]
    wait 10
    kill-thread n
end
```

stopme

Control Stuctures

Description

Stops the current thread.

Usage Format

```
stopme
```

Sample Code and Output

```
to read-sensor
    resett
    ignore launch
    [
        loop [print sensor 1
              if (timer > 10) [stopme]]
    ]
end
```

no-multi

Control Stuctures

Description

Turns off multitasking to prevent other threads from interrupting the current process. This is very useful for functions that are dependent on hardware-level control, such as those involving I²C communication.

Usage Format

```
no-multi [code-block]
```

The **code-block** argument is the code that should run while multitasking is turned off.

Sample Code and Output

```
to display-clear
    no-multi
    [
        i2c-start
        i2c-write-byte $16
        i2c-write-byte 1
        i2c-write-byte 0
        i2c-stop
        i2c-start
        i2c-write-byte $17
        waituntil [i2c-read-byte = 1]
        ignore i2c-read-byte-last
        i2c-stop
    ]
end
```

aset

Strings and Arrays

Description

Sets an element of an array to a specified value.

Usage Format

```
aset name index value
```

The **name** argument is the name of the array, the **index** argument is the location in the array to modify, and **value** is the number to set the array element to.

Sample Code and Output

```
arrays [[foo 10][bar 5]]

aset foo 1 23

print aget foo 1
> 23
```

aget

Strings and Arrays

Description

Obtains the value of a specified element of an array.

Usage Format

```
aget name index
```

The **name** argument is the name of the array, and the **index** argument is the location in the array to read.

Sample Code and Output

```
arrays [[foo 10][bar 5]]

aset foo 1 23

print aget foo 1
> 23
```

set-char

Strings and Arrays

Description

Sets a character in a string to a specified value.

Usage Format

```
set-char name index value
```

The **name** argument is the name of the string, the **index** argument is the location in the string to modify, and **value** is the character to set the string element to.

Sample Code and Output

```
strings [[foo 10][bar 5]]

strcpy foo "cat

print-string foo
> cat

set-char foo 0 104

print-string foo
> hat
```

The number 104 used as an argument to **set-char** is the ASCII value for the lower-case "h" character.

get-char

Strings and Arrays

Description

Obtains the value of a character in a string.

Usage Format

```
get-char name index
```

The **name** argument is the name of the string, and the **index** argument is the location in the string to modify.

Sample Code and Output

```
strings [[foo 10][bar 5]]

strcpy foo "cat

put-serial 3 get-char foo 0
> c
```

The **put-serial** function will print the character corresponding to a given ASCII value over a serial port. The number "3" indicates the third serial port, which is the one that will be connected to the host computer.

strcpy

Strings and Arrays

Description

Copies the contents of one string into another.

Usage Format

```
strcpy string1 string2
```

The contents of **string2** are copied into **string1**, starting at the beginning of the string.

Sample Code and Output

```
strings [[foo 10][bar 5]]

stcpy foo "hello

print-string foo
> hello
```

strcmp

Strings and Arrays

Description

Compares two strings to see if they match, case sensitive.

Usage Format

```
strcmp string1 string2
```

The **string1** and **string2** arguments are the strings to be compared.

Sample Code and Output

```
ifelse (strcmp "foo "foo) [print 1][print 0]
> 1

ifelse (strcmp "foo "Foo) [print 1][print 0]
> 0
```

strcmpi

Strings and Arrays

Description

Compares two strings to see if they match, case insensitive.

Usage Format

```
strcmpi string1 string2
```

The **string1** and **string2** arguments are the strings to be compared.

Sample Code and Output

```
ifelse (strcmpi "foo "foo) [print 1][print 0]
> 1

ifelse (strcmpi "foo "Foo) [print 1][print 0]
> 1
```

strcat

Strings and Arrays

Description

Concatenates two strings.

Usage Format

```
strcat string1 string2
```

The contents of **string2** are concatenated onto the end of **string1**.

Sample Code and Output

```
strings [[foo 10][bar 5]]

strcpy foo "123

strcat foo "456

print-string foo
> 123456
```

strlen

Strings and Arrays

Description

Determines the length of a string, counting only the characters that are being used.

Usage Format

```
strlen string
```

The **string** argument is the string to be measured.

Sample Code and Output

```
strings [[foo 10][bar 5]]

strcpy foo "example

print strlen foo
> 6
```

strmaxlen

Strings and Arrays

Description

Determines the maximum length of a string, based on the number of memory locations that were allocated for it when it was created.

Usage Format

```
strmaxlen string
```

The **string** argument is the string to be measured.

Sample Code and Output

```
strings [[foo 10][bar 5]]

stcpy foo "example

print strmaxlen foo
> 10
```

str-to-num

Strings and Arrays

Description

Converts a string to a number.

Usage Format

```
str-to-num string
```

The **string** argument is the string to be converted.

Sample Code and Output

```
strings [[foo 10][bar 5]]

strcpy foo "123

print str-to-num foo
> 123
```

num-to-str

Strings and Arrays

Description

Converts a number to a string. The fnuction takes the string as an argument, and fills it with the appropriate contents.

Usage Format

```
num-to-str string number
```

The **string** argument is the empty string to be filled with the resulting text, and the **number** argument is the number to be converted.

Sample Code and Output

```
strings [[foo 10][bar 5]]

num-to-str foo 123

print-string foo
> 123
```

set-pin

Pin Control

Description

Sets a pin on a Rabbit port register.

Usage Format

```
set-pin pin-number port
```

The argument **pin-number** is the pin (0-7) of the Rabbit port register with address **port**, that should be set.

Sample Code and Output

```
to flash :pin :port
    set-pin :pin :port
    wait 10
    clear-pin :ping :port
end
```

clear-pin

Pin Control

Description

Clears a pin on a Rabbit port register.

Usage Format

```
clear-pin pin-number port
```

The argument **pin-number** is the pin (0-7) of the Rabbit port register with address **port**, that should be cleared.

Sample Code and Output

```
to flash :pin :port
    set-pin :pin :port
    wait 10
    clear-bit :pin :port
end
```

test-pin

Pin Control

Description

Tests a pin on a Rabbit port register.

Usage Format

```
test-pin pin-number port
```

The argument **pin-number** is the pin (0-7) of the Rabbit port register with address **port**, that should be tested.

Sample Code and Output

```
print test-pin 1 porta
> (prints a 1 if pin 1 of port A is set)
> (prints a 0 if pin 1 of port A is clear)
```

write-port

Pin Control

Description

Sets the state of a Rabbit port register equal to a value. This is useful for changing every pin on a port at the same time.

Usage Format

```
write-port port value
```

The **port** argument is the Rabbit I/O register to be written to, and **value** is the value to write to the port.

Sample Code and Output

```
write-port porta $55
```

This line of code will set every other pin on port A, leaving the ones in between clear.

read-port

Pin Control

Description

Reads the state of an entire Rabbit port register. This is useful for reading every pin on a port at the same time.

Usage Format

```
read-port port
```

The **port** argument is the Rabbit I/O register to be read.

Sample Code and Output

```
print read-port porta
> 85
```

In this test case, every other pin in port A was set. Remember, the result printed is in decimal format, not hexadecimal. The decimal number 85 is 55 in hex, corresponding to every other bit being a "1".

new-serial?

Communication

Description

Returns the number of bytes that have been received on the serial port and are waiting in the buffer.

Usage Format

```
new-serial? port-number
```

The **port-number** argument is the number of the serial port to use (from 1 to 4). Port 3 is the one that can be connected directly to a computer using the RS-232 module.

Sample Code and Output

```
loop [if ((new-serial? 3) > 0) [print get-serial 3]
```

get-serial

Communication

Description

Returns a byte received on the serial port, and removes that byte from the receive buffer. As bytes are read out of the buffer, the result of a **new-serial?** operation will decrease accordingly, assuming that no new data is being received in the process.

Usage Format

```
get-serial port-number
```

The **port-number** argument is the number of the serial port to use (from 1 to 4). Port 3 is the one that can be connected directly to a computer using the RS-232 module.

Sample Code and Output

```
loop [if ((new-serial? 3) > 0) [print get-serial 3]
```

put-serial

Communication

Description

Sends a byte over the serial port.

Usage Format

```
put-serial port-number value
```

The **port-number** argument is the number of the serial port to use (from 1 to 4). Port 3 is the one that can be connected directly to a computer using the RS-232 module. The argument **value** is the byte to send over serial.

Note: When the byte is sent back to the TDE, it will print the ASCII character corresponding to the byte sent.

Sample Code and Output

```
put-serial 3 65
> A
```

put-serial-string

Communication

Description

Sends a string over the serial port.

Usage Format

```
put-serial port-number string
```

The **port-number** argument is the number of the serial port to use (from 1 to 4). Port 3 is the one that can be connected directly to a computer using the RS-232 module. The argument **string** is the string to send over serial.

Sample Code and Output

```
put-serial-string 3 "hello
> hello
```

put-serial-num

Communication

Description

Sends a number over the serial port.

Usage Format

```
put-serial port-number number
```

The **port-number** argument is the number of the serial port to use (from 1 to 4). Port 3 is

the one that can be connected directly to a computer using the RS-232 module. The argument **number** is the number to send over serial.

Sample Code and Output

```
put-serial-num 3 65
> 65
```

put-serial-float

Communication

Description

Sends a floating-point number over the serial port.

Usage Format

```
put-serial-float port-number float
```

The **port-number** argument is the number of the serial port to use (from 1 to 4). Port 3 is the one that can be connected directly to a computer using the RS-232 module. The argument **float** is the floating-point number to send over serial.

Sample Code and Output

```
put-serial-float 3 16.5f
> 16.5
```

serial-set-baud

Communication

Description

Sets the serial communication baud rate.

Usage Format

```
serial-set-baud port-number rate
```

The **port-number** argument is the number of the serial port to set the baud rate of (from 1 to 4). Port 3 is the one that can be connected directly to a computer using the RS-232 module. The argument **rate** is the new baud rate to use.

Note: The default baud rate for serial communication is 9600 bps.

Sample Code and Output

```
serial-set-baud 3 57600
```

i2c-start

Communication

Description

Starts an I²C communication sequence.

Usage Format

```
i2c-start
```

Sample Code and Output

```
to turn-servo :number :angle
    no-multi
    [
        i2c-start
        i2c-write-byte $04
        i2c-write-byte 2
        i2c-write-byte (:number % 9) - 1
        i2c-write-byte :angle
        i2c-stop
    ]
end
```

i2c-write-byte

Communication

Description

Writes a byte to the I²C bus.

Usage Format

```
i2c-write-byte value
```

The **value** argument is the value to be sent over the bus. The value itself represents different things based on its location after the **i2c-start** command. Following the example code below, the first **i2c-write-byte** command takes the address of the I²C slave device as an argument. The next argument to the **i2c-write-byte** command represents the number of arguments to follow, while each subsequent **i2c-write-byte** command, prior to an **i2c-stop** command, sends an argument itself, and depends on the functionality of the I2C slave device being communicated with. All I2C slave addresses are even numbers. Adding "!" to the address, making it odd, indicates that a data-read operation is being initiated, as opposed to a data-write.

Sample Code and Output

```
to turn-servo :number :angle
    no-multi
    [
        i2c-start
        i2c-write-byte $04
        i2c-write-byte 2
        i2c-write-byte (:number % 9) - 1
        i2c-write-byte :angle
        i2c-stop
    ]
end
```

i2c-read-byte

Communication

Description

Reads a byte from the I²C bus.

Usage Format

```
i2c-read-byte
```

This function should be used for reading all but the last byte in a sequence from an I²C slave device. When receiving the last byte, a special termination sequence must occur, so the **i2c-read-byte-last** primitive should be used instead.

Sample Code and Output

```
to sensor :n
    let [i2c-byte 0]
    no-multi
    [
        i2c-start
        i2c-write-byte $0a
        i2c-write-byte 2
        i2c-write-byte 0
        i2c-write-byte (:n - 1)
        i2c-stop
        i2c-start
        i2c-write-byte $0b
        ignore i2c-read-byte
        make "i2c-byte ( i2c-read-byte << 8)
        make "i2c-byte
             :i2c-byte bit-or i2c-read-byte-last
        i2c-stop
    ]
    output :i2c-byte
end
```

i2c-read-byte-last

Communication

Description

Reads the last byte in a sequence from the I²C bus.

Usage Format

```
i2c-read-byte-last
```

This function should be used for reading the last byte in a sequence from an I²C slave device. When receiving all but the last byte, the **i2c-read-byte** primitive should be used instead.

Sample Code and Output

```
to sensor :n
    let [i2c-byte 0]
    no-multi
    [
        i2c-start
        i2c-write-byte $0a
        i2c-write-byte 2
        i2c-write-byte 0
        i2c-write-byte (:n - 1)
        i2c-stop
        i2c-start
        i2c-write-byte $0b
        ignore i2c-read-byte
        make "i2c-byte ( i2c-read-byte << 8)
        make "i2c-byte
             :i2c-byte bit-or i2c-read-byte-last
        i2c-stop
    ]
    output :i2c-byte
end
```

i2c-check

Communication

Description

Checks to see if a board is present at a given address.

Usage Format

```
i2c-check address
```

The **address** argument is the I²C slave device address to be checked. It should be an even value from 2 to 126. The function returns a "1" if the device is present, and a "0" if it is not.

Sample Code and Output

```
to test-for-board :address
    let [i2c-byte 0]
    no-multi
    [
        i2c-start
        make "i2c-byte i2c-check :address
        i2c-stop
    ]
        output :i2c-byte
end
```

i2c-stop

Communication

Description

Terminates I²C communication.

Usage Format

```
i2c-stop
```

In addition to being a use-available primitive, **i2c-stop** is automatically called after every program terminates, to ensure that no slave devices are left in a bad state if a program ends abnormally.

Sample Code and Output

```
to turn-servo :number :angle
    no-multi
    [
        i2c-start
        i2c-write-byte $04
        i2c-write-byte 2
        i2c-write-byte (:number % 9) - 1
        i2c-write-byte :angle
        i2c-stop
    ]
end
```

tcp-init

TCP/IP and Sockets

Description

Sets up and nitializes TCP/IP information. This function or **tcp-init-dhcp** should be run each time the Rabbit Tower is powered on if socket communications will be used.

Usage Format

```
tcp-init ipaddress netmask gateway nameserver
```

The **ipaddress** argument is a string representing the IP address of the Rabbit network node, the **netmask** argument is a string representing the Rabbit's netmask, the **gateway** argument is the Rabbit's default gateway, and the **nameserver** argument is the Rabbit's default nameserver.

Sample Code and Output

```
tcp-init "18.85.44.213 "255.255.255.0
         "18.85.33.1 "18.85.2.171
```

tcp-init-dhcp

TCP/IP and Sockets

Description

Sets up and nitializes TCP/IP information automatically using DHCP. This function or **tcp-init** should be run each time the Rabbit Tower is powered on if socket communications will be used.

Usage Format

```
tcp-init-dhcp
```

Sample Code and Output

```
tcp-init-dhcp
```

acquire-dhcp

TCP/IP and Sockets

Description

Renews TCP/IP information by DHCP. The function returns a number representing whether or not the DHCP renewal request was successful.

Usage Format

```
acquire-dhcp
```

Sample Code and Output

```
if (acquire-dhcp) [print-string "Success!]
> Success!
```

tcp-tick

TCP/IP and Sockets

Description

Processes TCP packet communication at high speed. Usually launched in a background thread to ensure that all socket-based communications will function properly.

Usage Format

```
tcp-tick
```

Sample Code and Output

```
ignore launch [loop [tcp-tick]]
```

get-ip

TCP/IP and Sockets

Description

Obtains the IP address currently in use.

Usage Format

```
get-ip
```

The IP address is returned as a 32-bit number. Individual components of the address can be determined by only looking at the desired bits, as shown in the example below.

Sample Code and Output

```
print get-ip
> 307571925

print-ip-address get-ip
> 18.85.44.213
```

get-netmask

TCP/IP and Sockets

Description

Obtains the netmask currently in use.

Usage Format

```
get-netmask
```

The netmask is returned as a 32-bit number. Individual components of the address can be determined by only looking at the desired bits, as shown in the example below.

Sample Code and Output

```
print get-netmask
> 4294967040

print-ip-address get-netmask
> 255.255.255.0
```

resolve-dns

TCP/IP and Sockets

Description

Used to convert a host-name into an IP address.

Usage Format

```
resolve-dns host-name
```

The **host-name** argument is a string representing the name of the server to be looked up.

Sample Code and Output

```
print-ip-address resolve-dns "gig.media.mit.edu
> 18.85.45.11
```

fill-with-mac-address

TCP/IP and Sockets

Description

Used to obtain a device's unique MAC address. The address is stored in a string which is passed to the function as an argument.

Usage Format

```
fill-with-mac-address string
```

The **string** argument is an empty string to be filled with a devices MAC address.

Sample Code and Output

```
to print-my-mac-addr
    print-string "|my mac addr is: |
    let [index 0]
    fill-with-mac-address mac-addr-buf
    repeat 5 [print-hex-number
                  getchar mac-addr-buf :index
              make "index :index + 1
              print-string ":]
    print-hex getchar mac-addr-buf :index
end
```

sock-open-init

TCP/IP and Sockets

Description

Opens a socket connection to a given host ip address at any specified port. Returns a pointer to the socket that has been opened.

Usage Format

```
sock-open-init host port
```

The **host** argument is a string representing the target IP address. If a DNS server is available, the argument can also be specified as a string with the hostname of the target machine. The **port** argument is the port to open the connection on.

Note: This primitive should not be called directly. It is called by the **sock-open** *function in the net include file to ensure that a connection is successfully established.*

Sample Code and Output

```
to sock-open :host :port
    let [tsock sock-open-init :host :port]
    let [timeout 30 start read-rtc]
    loop
    [
        if (sock-established-loop :tsock)
           [output :tsock]
        if (read-rtc > (:start + :timeout))
           [output 0]
    ]
end
```

sock-accept-init

TCP/IP and Sockets

Description

Accepts a socket connection on any specified port. Returns a pointer to the socket that has been accepted.

Usage Format

```
sock-accept-init port
```

The **port** argument is the port number to open the socket on.

Note: This primitive should not be called directly. It is called by the **sock-accept** *function in the net include file to ensure that a connection is successfully established.*

Sample Code and Output

```
to sock-accept :port
    let [tsock sock-accept-init :port]
    loop
    [
        if (sock-established-loop :tsock)
           [output :tsock]
    ]
end
```

sock-established-loop

TCP/IP and Sockets

Description

Used to determine if a socket connection has been successfully established.

Usage Format

```
sock-established-loop socket
```

The **socket** argument is a pointer to the socket to be established. The socket pointer is obtained when a socket is opened or accepted.

Note: This primitive should not be called directly. It is called by the **sock-open** *and* **sock-accept** *functions in the net include file to ensure that a connection is successfully established.*

Sample Code and Output

```
to sock-open :host :port
    let [tsock sock-open-init :host :port]
    let [timeout 30 start read-rtc]
    loop
    [
        if (sock-established-loop :tsock)
           [output :tsock]
        if (read-rtc > (:start + :timeout))
           [output 0]
    ]
end
```

sock-close

TCP/IP and Sockets

Description

Closes a socket connection.

Usage Format

```
sock-established-loop socket
```

The **socket** argument is a pointer to the socket to be established. The socket pointer is obtained when a socket is opened or accepted.

Sample Code and Output

```
setn sock-open "gig.media.mit.edu 80

sock-close n
```

sock-write

TCP/IP and Sockets

Description

Writes a string to a specified socket.

Usage Format

```
sock-write socket string
```

The **socket** argument is a pointer to the socket to be written to. The socket pointer is obtained when a socket is opened or accepted. The **string** argument is the string to write to the socket.

Sample Code and Output

```
setn sock-open "gig.media.mit.edu 80

sock-write n "hello

sock-close n
```

sock-writec

TCP/IP and Sockets

Description

Writes a single character to a specified socket.

Usage Format

```
sock-writec socket value
```

The **socket** argument is a pointer to the socket to be written to. The socket pointer is obtained when a socket is opened or accepted. The **value** argument is the character to write to the socket.

Sample Code and Output

```
setn sock-open "gig.media.mit.edu 80

sock-writec n 75

sock-close n
```

sock-read

TCP/IP and Sockets

Description

Reads a string from a specified socket. An empty string is supplied to the function, which then fills it with the received data, and returns a number corresponding to the length of the string.

Usage Format

```
sock-read socket string
```

The **socket** argument is a pointer to the socket to be read from. The socket pointer is obtained when a socket is opened or accepted. The **string** argument is an empty string to be

filled with the received data.

Sample Code and Output

```
strings [[foo 10]]

setn sock-open "gig.media.mit.edu 80

print sock-read n foo
> 5

print-string foo
> hello

sock-close n
```

sock-readc

TCP/IP and Sockets

Description

Reads a single character from a specified socket.

Usage Format

```
sock-readc socket
```

The **socket** argument is a pointer to the socket to be read from. The socket pointer is obtained when a socket is opened or accepted.

Sample Code and Output

```
setn sock-open "gig.media.mit.edu 80

print sock-readc n
> 75

sock-close n
```

arithmetic (+, -, *, /, %)

Arithmetic and Logic

Description

Returns the result of arithmetical operators (+, -, *, /, %) applied to 32-bit numerical integer operands.

Usage Format

```
num1 (+, -, *, /, %) num2
```

Note: The "/" operator returns the integer part of the result if dividing num1 by num2. The "%" operator returns the remainder of a dividing num1 by num2.

Sample Code and Output

```
print 4 + -10
> -6

print 42 - 8
> 34

print 4 * -10
> -40

print 35 / 10
> 3

print 35 % 10
> 5
```

comparisons (=, >, <)

Arithmetic and Logic

Description

Returns a Boolean depending on the result of arithmetical comparison operators (=, <, >) applied to 32-bit numerical integer operands.

Usage Format

```
num1 (=, >, <) num2
```

Sample Code and Output

```
print 10 = 10
> 1

print 4 > 2
> 1

print 4 < 2
> 0
```

floating-point arithmetic (f+, f-, f*, f/)

Arithmetic and Logic

Description

Returns the result of arithmetical operators (+, -, *, /) applied to 32-bit numerical floating-point operands.

Usage Format

```
num1 (+, -, *, /) num2
```

Sample Code and Output

```
print 4.3f f+ 6.9f
> 11.2

print 42.1f f- 8.7f
> 33.4

print 5.2f f* -10.1f
> -52.52

print 35.8f f/ 10.0f
> 3.58
```

floating-point comparisons (f=, f>, f<)

Arithmetic and Logic

Description

Returns a Boolean depending on the result of arithmetical comparison operators (=, <, >) applied to 32-bit numerical floating-point operands.

Usage Format

```
num1 (=, >, <) num2
```

Sample Code and Output

```
print 10.0f f= 10.0f
> 1

print 4.1f f> 2.3f
> 1

print 4.1f f< 2.3f
> 0
```

int-to-float

Arithmetic and Logic

Description

Converts an integer number to a floating point one.

Usage Format

```
int-to-float integer
```

The **integer** argument is the number to be converted to floating-point format.

Sample Code and Output

```
print-float int-to-float 10
> 10
```

float-to-int

Arithmetic and Logic

Description

Converts a floating point number to an integer one.

Usage Format

```
float-to-int float
```

The **float** argument is the number to be converted to integer format.

Sample Code and Output

```
print-num float-to-int 10.4f
> 10
```

bit-wise logic (bit-and, bit-or, bit-xor)

Arithmetic and Logic

Description

Returns the result of bit-wise logical operators (and, or, xor) applied to 32-bit numerical integer operands.

Usage Format

```
num1 (bit-and, bit-or, bit-xor) num2
```

Sample Code and Output

```
print 20 bit-and 4
> 4

print 16 bit-or 4
> 20

print 7 bit-xor 3
> 4

print (1 = 1) and (1 = 0)
> 0

print (1 = 1) or (1 = 0)
> 1
```

logic (and, or)

Arithmetic and Logic

Description

Returns a Boolean of logibal operators (and, or) applied to 32-bit numerical integer operands.

Usage Format

```
num1 (and, or) num2
```

Sample Code and Output

```
print 0 and 20
> 0

print 20 and 3
> 1

print 7 or 0
> 1

print 0 or 0
> 0
```

not

Arithmetic and Logic

Description

Returns a Boolean not of a number.

Usage Format

```
not num
```

Note: This operation is a Boolean not and not a bit-wise not.

Sample Code and Output

```
print not 42
> 0

print not 0
> 1
```

shift (>>, <<)

Arithmetic and Logic

Description

Used to left shift or right shift a number by a specified number of bits.

Usage Format

```
num (>>, <<) dist
```

The argument **num** is the number to shift and **dist** is the number of bits by which to shift num left or right.

Sample Code and Output

```
print 4 << 1
> 8

print 64 >> 2
> 16
```

random

Arithmetic and Logic

Description

Returns a random number between 0 and 2147483648, inclusive.

Usage Format

```
random
```

Sample Code and Output

```
print random
> 5622684

print random % 256
> 142
```

set-random-seed

Arithmetic and Logic

Description

Sets the seed for the random number generator with a given 32-bit value

Usage Format

```
set-random-seed value
```

The random number generator is normally seeded automatically off of the internal timer.

However, if **on-starup** is used, the timer will always be the same value when seeded. Usually, if setting the seed manually, one would want to set it to a relatively unpredictable value, such as a floating sensor value.

Sample Code and Output

```
set-random-seed sensor 1

print random
> 9460
```

The **sensor** function is used to return a value corresponding to a voltage reading from the Sensor Layer. If the port is left unconnected, its voltage will float, providing a good random number seed. The function is defined in the Sensor layer include file.

Appendix D: PIC Assembly Language Reference

PIC Assembly is the lowest-level programming lanugage for Microchip PIC microcontrollers. These processors are used on the LogoChip and LogoChip modules, the LogoBoard, the PIC Foundation, and every layer in the Tower system.

When using PIC Assembly, user programs are entered on a desktop computer and assembled into byte codes that are transferred to the PIC processor through the serial port of the host computer.

Our assembly language differs in some ways from the commercially-available one. It is designed to be more human-readable than the standard instruction set, and can be easily modified to support other processors while retaining its user-friendly naming scheme.

As in all assembly-level programming, computation is performed in an "accumulator" scratch register, which is modified as needed by the different processor instructions.

The basic instruction format uses square brackets to frame each instruction:

```
[instruction argument]
```

Program location labels are defined by plain text, with no brackets:

```
program-label
```

Constants are used to define memory locations for both built-in registers and variable storage. A constant is defined like this:

```
[const name address]
```

Memory addresses can be denoted either in decimal or hexidecimal format, as indicated by placing a "$" character in front of the number.

The **status** register, at memory location "$03", is a special one on the processor. It has user-readable flags that represent different conditions, which can be triggered by the results of various instructions. In most cases, the two status flags that one would be most concerned with are:

- C - The carry flag, which is set when an instruction overflows a register.
- Z - The zero flag, which is set when the result of an instruction is "0".

What follows, is a list of all instructions currently supported in the PIC Assembler. The instructions are organized by their functionality, for ease of reference when the exact name is not known. The list below represents the order in which their detailed descriptions and examples of use can be found on the subsequent pages:

Program Flow

Register Manipulation

Bit Manipulation

Arithmetic

Logic

System Commands

bsr - branch to subroutine

Program Flow

Description

Branches to a subroutine located at a given memory location. The value of the program counter incremented by one is pushed onto the stack.

Usage Format

```
[bsr address]
```

The **address** argument is a number corresponding to a program memory location, usually specified by the name of a location label.

Status Flags Affected

None

rts - return from subroutine

Program Flow

Description

Returns from a subroutine. The top value on the stack is popped and loaded into the program counter, returning to the program location of the most recently called **bsr** instruction.

Usage Format

```
[rts]
```

Status Flags Affected

None

rtv - return from subroutine with value

Program Flow

Description

Returns from a subroutine with a value stored in the accumulator. The top value on the stack is popped and loaded into the program counter, returning to the program location of the most recently called **bsr** instruction.

Usage Format

```
[rtv number]
```

The **number** argument is the value to return.

Status Flags Affected

None

bra - unconditional branch

Program Flow

Description

Branches to a given memory location.

Usage Format

```
[bra address]
```

The **address** argument is a number corresponding to a program memory location, usually specified by the name of a location label.

Status Flags Affected

None

rti - return from interrupt

Program Flow

Description

Returns from an interrupt. The top value on the stack is popped and loaded into the program counter, returning to the program location at the time of interrupt.

Usage Format

```
[rti]
```

Status Flags Affected

None

lda - load accumulator

Register Manipulation

Description

Loads the accumulator with the contents of a register.

Usage Format

```
[lda register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

ldan - load accumulator with number

Register Manipulation

Description

Loads the accumulator with a number.

Usage Format

```
[ldan number]
```

The **number** argument is the value to be loaded into the accumulator.

Status Flags Affected

None

sta - store accumulator

Register Manipulation

Description

Stores the contents of the accumulator in a register.

Usage Format

```
[lda register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

tst - test

Register Manipulation

Description

Tests the value of a register and sets the zero flag if the contents of the register is equal to "0".

Usage Format

```
[tst register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

clr - clear

Register Manipulation

Description

Sets the contents of a register to "0".

Usage Format

```
[clr register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

clra - clear accumulator

Register Manipulation

Description

Sets the contents of the accumulator to "0".

Usage Format

```
[clra]
```

Status Flags Affected

Z

inc - increment

Register Manipulation

Description

Increments the contents of a register and stores the result in the register.

Usage Format

```
[inc register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

linc - load and increment

Register Manipulation

Description

Increments the contents of a register and stores the result in the accumulator.

Usage Format

```
[linc register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

incsz - increment skip if zero

Register Manipulation

Description

Increments the contents of a register and stores the result in the register. If the result of the operation is zero, the next program intsruction is skipped.

Usage Format

```
[incsz register]
```

The **register** argument is the address of any register.

Status Flags Affected

None

lincsz - load and increment skip if zero

Register Manipulation

Description

Increments the contents of a register and stores the result in the accumulator. If the result of the operation is zero, the next program intsruction is skipped.

Usage Format

```
[lincsz register]
```

The **register** argument is the address of any register.

Status Flags Affected

None

dec - decrement

Register Manipulation

Description

Decrements the contents of a register and stores the result in the register.

Usage Format

```
[dec register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

ldec - load and decrement

Register Manipulation

Description

Decrements the contents of a register and stores the result in the accumulator.

Usage Format

```
[ldec register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

decsz - decrement skip if zero

Register Manipulation

Description

Decrements the contents of a register and stores the result in the register. If the result of the operation is zero, the next program instruction is skipped.

Usage Format

```
[decsz register]
```

The **register** argument is the address of any register.

Status Flags Affected

None

ldecsz - load and decrement skip if zero

Register Manipulation

Description

Decrements the contents of a register and stores the result in the accumulator. If the result of the operation is zero, the next program intsruction is skipped.

Usage Format

```
[ldecsz register]
```

The **register** argument is the address of any register.

Status Flags Affected

None

rol - rotate left

Register Manipulation

Description

The contents of a register are rotated to the left through the carry flag and the result is stored in the register.

Usage Format

```
[rol register]
```

The **register** argument is the address of any register.

Status Flags Affected

C

lrol - load and rotate left

Register Manipulation

Description

The contents of a register are rotated to the left through the carry flag and the result is stored in the accumulator.

Usage Format

```
[lrol register]
```

The **register** argument is the address of any register.

Status Flags Affected

C

ror - rotate right

Register Manipulation

Description

The contents of a register are rotated to the right through the carry flag and the result is stored in the register.

Usage Format

```
[ror register]
```

The **register** argument is the address of any register.

Status Flags Affected

C

lror - load and rotate right

Register Manipulation

Description

The contents of a register are rotated to the right through the carry flag and the result is stored in the accumulator.

Usage Format

```
[lror register]
```

The **register** argument is the address of any register.

Status Flags Affected

C

com - complement

Register Manipulation

Description

Complements the contents of a register and stores the result in the register.

Usage Format

```
[com register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

lcom - load and complement

Register Manipulation

Description

Complements the contents of a register and stores the result in the accumulator.

Usage Format

```
[lcom register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

swap - swap

Register Manipulation

Description

The upper and lower halves of a register are swapped and the result is stored in the register.

Usage Format

```
[swap register]
```

The **register** argument is the address of any register.

Status Flags Affected

None

lswap - load and swap

Register Manipulation

Description

The upper and lower halves of a register are swapped and the result is stored in the accumulator.

Usage Format

```
[lswap register]
```

The **register** argument is the address of any register.

Status Flags Affected

None

bset - bit set

Bit Manipulation

Description

Makes the value of a given bit of a register equal to "1".

Usage Format

```
[bset bit register]
```

The **bit** argument is any bit number (from 0 to 7) and the **register** argument is the address of any register.

Status Flags Affected

None

bclr - bit clear

Bit Manipulation

Description

Makes the value of a given bit of a register equal to "0".

Usage Format

```
[bclr bit register]
```

The **bit** argument is any bit number (from 0 to 7) and the **register** argument is the address of any register.

Status Flags Affected

None

btss - bit test skip if set

Bit Manipulation

Description

Tests the value of a given bit of a register. If it is equal to "1," the next program instruction is skipped.

Usage Format

```
[btss bit register]
```

The **bit** argument is any bit number (from 0 to 7) and the **register** argument is the address of any register.

Status Flags Affected

None

btsc - bit test skip if clear

Bit Manipulation

Description

Tests the value of a given bit of a register. If it is equal to "0," the next program instruction is skipped.

Usage Format

```
[btsc bit register]
```

The **bit** argument is any bit number (from 0 to 7) and the **register** argument is the address of any register.

Status Flags Affected

None

add - add

Arithmetic

Description

Adds the contents of the accumulator to a reguster and stores the result in the accumulator.

Usage Format

```
[add register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C

addm - add to memory

Arithmetic

Description

Adds the contents of the accumulator to a register and stores the result in the register.

Usage Format

```
[addm register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C

addn - add number

Arithmetic

Description

Adds the conents of the accumulator to a number and stores the result in the accumulator.

Usage Format

```
[addn number]
```

The **number** argument is the value to be added to.

Status Flags Affected

Z, C

sub - subtract

Arithmetic

Description

Subtracts the contents of the accumulator from a register and stores the result in the accumulator.

Usage Format

```
[sub register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C (C=0 means result is negative)

subm - subtract from memory

Arithmetic

Description

Subtracts the contents of the accumulator from a register and stores the result in the register.

Usage Format

```
[subm register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C (C=0 means result is negative)

subn - subtract number

Arithmetic

Description

Subtracts the contents of the accumulator from a number and stores the result in the accumulator.

Usage Format

```
[subn number]
```

The **number** argument is the value to be subtracted from.

Status Flags Affected

Z, C (C=0 means result is negative)

and - and

Logic

Description

Performs a bit-wise "and" of the contents of the accumulator with a reguster and stores the result in the accumulator.

Usage Format

```
[and register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C

andm - and memory

Logic

Description

Performs a bit-wise "and" of the contents of the accumulator with a reguster and stores the result in the register.

Usage Format

```
[andm register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

andn - and number

Logic

Description

Performs a bit-wise "and" of the contents of the accumulator with a number and stores the result in the accumulator.

Usage Format

```
[andn number]
```

The **number** argument is the value to "and" the accumulator with.

Status Flags Affected

Z

or - or

Logic

Description

Performs a bit-wise "or" of the contents of the accumulator with a reguster and stores the result in the accumulator.

Usage Format

```
[or register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C

orm - or memory

Logic

Description

Performs a bit-wise "or" of the contents of the accumulator with a reguster and stores the result in the register.

Usage Format

```
[orm register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

orn - or number

Logic

Description

Performs a bit-wise "or" of the contents of the accumulator with a number and stores the result in the accumulator.

Usage Format

```
[orn number]
```

The **number** argument is the value to "or" the accumulator with.

Status Flags Affected

Z

xor - exclusive or

Logic

Description

Performs a bit-wise "xor" of the contents of the accumulator with a reguster and stores the result in the accumulator.

Usage Format

```
[xor register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z, C

xorm - exclusive or memory

Logic

Description

Performs a bit-wise "xor" of the contents of the accumulator with a reguster and stores the result in the register.

Usage Format

```
[xorm register]
```

The **register** argument is the address of any register.

Status Flags Affected

Z

xorn - exclusive or number

Logic

Description

Performs a bit-wise "or" of the contents of the accumulator with a number and stores the result in the accumulator.

Usage Format

```
[xorn number]
```

The **number** argument is the value to "xor" the accumulator with.

Status Flags Affected

Z

nop - no operation

System Commands

Description

Does nothing. Usually used to perform precise timing operations.

Usage Format

```
[nop]
```

Status Flags Affected

None

sleep - sleep

System Commands

Description

Puts the processor to sleep, a mode where its power consumption is significantly reduced. It cannot perform any computation while in this state, but can be awakened by interrupts.

Usage Format

```
[sleep]
```

Status Flags Affected

None

clrwdt - clear watchdog timer

System Commands

Description

Resets the value of the internal watchdog timer.

Usage Format

```
[clrwdt]
```

Status Flags Affected

None

Appendix E: PIC Foundation Documentation

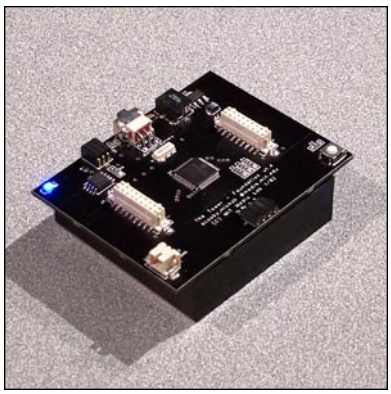


Figure E.1 - The PIC foundation.

Description

The PIC foundation *(Figure E.1)* is built around a Microchip PIC16F877 processor operating at 8 MHz. The foundation contains a serial programming header, as well as all of the power and support circuitry needed for basic operation. Every I/O pin from the processor is broken out to the main Tower connectors, and passed up and down the entire stack.

Hardware Detail

In addition to the pins used for I²C communication, each layer on the Tower stack has access to every other I/O pin from the foundation. To make this possible, all 33 I/O pins of the PIC16F877 are passed to every layer through the main Tower connectors. The connectors are 18-pin surface mount Hirose connectors. Each board has female connectors on the top, and male connectors on the bottom, so stacking layers is a simple process. In addition to the I/O pins, two power busses and a ground line are also available to every layer. The pin configuration for the connectors is detailed below *(Figure E.2)*:

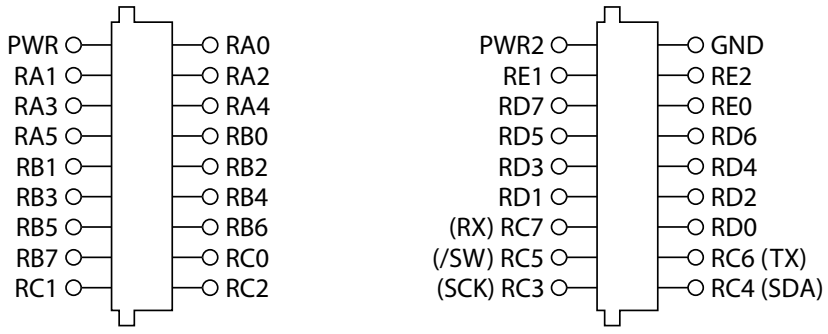


Figure E.2 - Tower connectors pin configuration.

The Foundation itself as well as many of the layers, obtain their power only off of the primary power bus. The primary bus can be powered by either four AA batteries, or through a 1mm wall jack also located on the foundation. The PIC foundation can accept either alkaline or rechargeable batteries. As an alternative to batteries, the wall jack can be used. It must have a 1mm negative-tip connector, providing at least 6 volts, since it gets regulated down to a tight 5 volts, and needs a bit of overhead to do so.

Some layers that consume more power have the option of using either the primary of secondary bus. To put power on the secondary bus, a cable is needed that plugs into the 2-pin power connector on the Foundation. This connector is a standard header, with the pinout shown below *(Figure E.3)*:

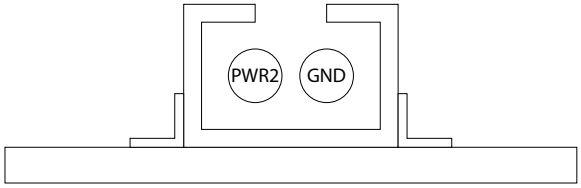


Figure E.3 - The foundation power connector.

Both power busses are switched on and off using the same power switch, located at the top edge of the Foundation. The switch is three-way, with the middle position being "off", and the two sides selecting between wall or battery power for the primary bus. Regardless of what source the primary bus is using, the secondary bus will always use whatever is provided to it through the connector, but can still be turned off by ensuring that the switch is in the middle position.

The Foundation itself can be programmed through the on-board serial-programming header when connected to a serial cable with an RS-232 module on the end. The pin configuration of the header on the board is shown below *(Figure E.4)*:

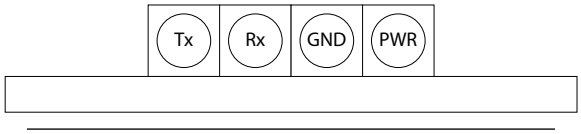


Figure E.4 - The foundation serial connector.

In addition to downloading Logo code, it is also possible to download Assembly code. To download assembly, the board must be powered on while the Button is depressed. At that point, Assembly code can be downloaded through the Tower Development Environment. After downloading new assembly code, it is necessary to power-cycle the layer before the new code will run.

In rare cases, it may be necessary to program the PIC on the Foundation with an actual hardware PIC programmer. If the serial-boot-loader program that handles assembly downloads becomes damaged, or if an assembly program needs to have precise control over the exact memory location of each instruction, it is necessary to program the chip directly through the programming header. The pin configuration of the 2mm programming header is shown here *(Figure E.5)*:

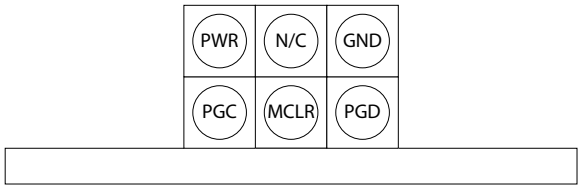


Figure E.5 - The foundation programming header.

A special cable is needed to connect this header to the hardware PIC programmer. Not that since the MCLR pin will be pulled to a high voltage during programming, it has a no-connect pin directly across

from it on the connector, to ensure that the high voltage will not damage the chip if the cable is accidentally connected upside down.

Foundation Code

For information on all of the primitives available in the virtual machine, please refer to the PIC Logo Language Reference document in Appendix B. There are however, three include files that pertain directly to functions on the foundation, which will be discussed here.

The standard include file contains the definitions for commonly used constants and global variables, as well as functions for driving output pins, performing A/D conversions, enabling Pulse-Width-Modulation, and looking up items in table entries.

The basic constant and global declarations are useful for standard arithmetic operations, as well as talking directly to frequently accessed registers such as the I/O ports. The declarations themselves are shown here:

```
constants
    [[porta 5][porta-ddr $85]
     [portb 6][portb-ddr $86]
     [portc 7][portc-ddr $87]
     [portd 8][portd-ddr $88]
     [porte 9][porte-ddr $89]
     [adcon0 $1f][adcon1 $9f]
       [adon 0][adgo 2]
     [adresh $1e][adresl $9e]
     [t2con $12][ccp1con $17]
     [ccpr1l $15][pr2 $92]]

globals [nn i2c-byte n]
```

For controlling output signals on PIC pins, three functions have been written to simplify the process. While at its basic level, controlling outputs is as simple as just setting and clearing bits, an important step that users often forget, is that they need to first ensure that the pins are configured as outputs. All three procedures first set the pin to be an output, before talking to the pin itself. The **set**, **clear**, and **toggle** functions can be used to respectively turn-on, turn-off, and switch the state of output pins.

```
to set :chan :port
    clear-bit :chan (:port + $80)
    set-bit :chan :port
end

to clear :chan :port
    clear-bit :chan (:port + $80)
    clear-bit :chan :port
end

to toggle :chan :port
    clear-bit :chan (:port + $80)
    flip-bit :chan :port
end
```

The **ad** function is used to perform a 10-bit A/D conversion using one of the eight built in A/D pins on the PIC. The function takes a single argument ranging from 1 to 8, that corresponds to a fixed pin on the chip as given in the following table *(Table E.1)*:

A/D Channel	Pin Number
0	A0
1	A1
2	A2
3	A3
4	A5
5	E0
6	E1
7	E2

Table E.1 - A/D Channel Mappings.

The function itself is as follows:

```
to ad :chan
    write-reg adcon1 $80
    write-reg adcon0 (:chan * 8) + $81
    set-bit adgo adcon0
    waituntil [not test-bit adgo adcon0]
    output ((read-reg adresh) * 256) +
             read-reg adresl
end
```

The **pwm** function is used to set up a Pulse-Width-Modulation signal on pin C2 of the chip. The function takes a single argument, a value from 0 to 255, and will create a fixed-frequency pulse-train with a

positive duty cycle directly proportional to the argument given.

```
to pwm :val
    clear-bit 2 portc-ddr
    write-reg t2con 6
    write-reg pr2 100
    write-reg ccp1con 12
    write-reg ccpr1l :val
end
```

The **table-item** function is used in conjunction with the table primitive, to read indexed values out of a previously defined table. The function takes two arguments, the name of the table itself as defined with the table primitive, and the index location that is to be read. The first item in the table will be at location 0, and will then count up from there. The function looks up the value in the corresponding memory location, and outputs it.

```
to table-item :table :index
    output (left-shift (read-prog-mem :table +
                (left-shift :index 1)) 8)
             or (read-prog-mem :table +
                (lsh :index 1) + 1)
end
```

The print include file contains the functions needed for formatting and printing numbers, characters, and strings directly to the terminal.

The most basic print functions are used to print numbers directly to the terminal. The procedures **print** and **type** both just call the **print-number** procedure, putting a negative sign in front if the number is below zero. The main difference between **print** and **type** is that print inserts a carriage return after the number, by calling the **cr** function, which just sends the ASCII sequence needed to jump to the next line in the terminal. The **print-number** function itself breaks a large number into individual digits, and then calls **print-digit** to send the ASCII code need to print each individual character.

```
to print :n
    if (:n < 0) [put-serial 45
                 print-number 0 - :n
                 cr
                 stop]
    print-number :n
    cr
end

to type :n
    if :n < 0 [put-serial 45
               print-number 0 - :n
               stop]
    print-number :n
end

to print-number :n
    if :n > 9999 [print-digit :n 10000]
    if :n > 999 [print-digit :n 1000]
    if :n > 99 [print-digit :n 100]
    if :n > 9 [print-digit :n 10]
    print-digit :n 1
end

to print-digit :n :d
    put-serial ((:n / :d) % 10) + 48
end

to cr
    put-serial 10 put-serial 13
end
```

Another print function has been defined which is useful for printing clock values, or any other case where a number below 10 should have a zero printed in front of it. The **print-clock** function does just that, by checking to see if the number is below the threshold, and if it is, printing a 0 before the actual number is printed.

```
to print-clock :n
    if :n < 0 [put-serial 45
               print-number 0 - :n
               stop]
    ifelse (:n > 9)
        [print-number :n]
        [print-number 0 print-number :n]
end
```

In some cases, we want to print numbers in hexadecimal format. .The **print-hex** function first prints a "$" character, and then splits the byte in half, and prints both the high and low nibbles using the **print-hex-digit** function, which simply prints the letters A through F corresponding to number 10 to 15.

```
to print-hex :n
    print-string "$
    print-hex-digit :n / 16
    print-hex-digit :n % 16
    cr
end

to print-hex-digit :n
    ifelse :n < 10
        [put-serial :n + 48]
        [put-serial :n + 55]
end
```

One last key function that is commonly used is **print-string**, which will print a character-based text string directly to the terminal. The procedure itself starts by pointing a variable to the beginning of the string, and then just sends out one character at a time until it reaches a value of "0" signifying the end of the string.

```
to print-string :n
    setnn :n
    loop
    [
        if (read-prog-mem nn) = 0 [stop]
        put-serial read-prog-mem nn
        setnn nn + 1
    ]
end
```

Finally, we have defined a **hi** function which just prints out a welcome string to the terminal. While not critically important for most programs, it serves as a good power-on test for the Tower, and can be called whenever you want to know if things are working properly.

```
to hi
    print-string
        "|    Welcome to Tower Development Kit|
    cr
end
```

The address include file contains the basic functions needed for scanning the Tower, and for changing

the address of any layer connected to it.

The **get-board-id** function is used to determine which specific board is located at a given address. The function itself takes a single argument, the address to query, which must be an even number between 0 and 254. The return value will be the hardware identifier, which can be looked up in the get-board-name function, also present in this file, to obtain the name of the actual board. The function sends two values to the actual layer, a "255" to enter address mode, and a "0" indicating that we want to obtain the hardware identifier. The identifier is then read back from the board, ignoring the first value back, which will just be a "1" indicating that a single byte is going to follow.

```
to get-board-id :address
    i2c-start
    i2c-write-byte :address
    i2c-write-byte 2
    i2c-write-byte 255
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte :address + 1
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **set-board-address** function is used to change the address of any layer plugged into the stack. The function takes two arguments, the old address, and the new address, both of which can be values from 0 to 254. However, it should be noted that "0" is the general-call-address, meaning that no layer should actually have it as its address. If a "0" is sent as the argument for the new address, the board will return to its default address, which is always defined to be exactly twice its hardware identifier. The function itself sends three values to the layer, a "255" to enter address mode, a "1" indicating that we want to write a new address, and then the new address itself, which must be an even number from 0 to 254.

```
to set-board-address :oldaddress :newaddress
      i2c-start
      i2c-write-byte :oldaddress
      i2c-write-byte 3
      i2c-write-byte 255
      i2c-write-byte 1
      i2c-write-byte :newaddress
      i2c-stop
end
```

The **scan-tower** function is used to check every address on the Tower, and when a board is found, it prints a string to the terminal indicating that fact. Starting at address 0, each address is individually checked to see if a board is present, by the i2c-check primitive. If a board is present, the hardware identifier is obtained, and then looked up by the get-board-name function. The resulting string is then printed to the terminal, followed by a statement indicating the address at which it was found. This process then repeats until the entire address space has been checked.

```
to scan-tower
    setn 0
    repeat 128
    [
        i2c-start
        setnn i2c-check n
        i2c-stop
        if (nn = 1)
        [
            print-string
                get-board-name get-board-id n
            print-string
                "| layer located at address |
            type n
            cr
        ]
        setn n + 2
    ]
    print-string "|Done!|
    cr
end
```

The **get-board-name** function is used to look up the name of a board, given its hardware identifier. The name is output directly as a string to the calling function. If no match is found, the phrase "Unknown" is returned.

```
to get-board-name :n
    if (:n = 1) [output "Bus stop]
    if (:n = 2) [output "Servo stop]
    if (:n = 3) [output "IR stop]
    if (:n = 4) [output "Voice stop]
    if (:n = 5) [output "Sensor stop]
    if (:n = 6) [output "EEPROM stop]
    if (:n = 7) [output "Clock stop]
    if (:n = 8) [output "CompactFlash stop]
    if (:n = 9) [output "I2C stop]
    if (:n = 10) [output "Tricolor stop]
    if (:n = 11) [output "Display stop]
    if (:n = 12) [output "RF stop]
    if (:n = 13) [output "Motor stop]
    if (:n = 15) [output "PICProto stop]
    if (:n = 16) [output "Serial stop]
    output "Unknown
end
```

Examples of Use

Let's start by saying "Hi" to our foundation:

```
hi
>     Welcome to Tower Development Kit
```

Let's do a little simple math, just to show that everything is in working order:

```
print 3 + 5
> 8
```

How about we try out some basic pin I/O. For this example, we'll say that we have an LED connected to pin B0. Let's turn it on:

```
set 0 portb
```

Now let's make the LED flash by writing a simple loop:

```
loop [set 0 portb wait 10 clear 0 portb wait 10]
```

This code turns the pin on, waits a second, turns it off, waits another second, and then keeps repeating. the process.

Before we do anything else, make sure to press the white button to stop the program, or we won't be able to communicate with the Foundation.

Let's play around with addressing. We can start by scanning the Tower like this:

```
scan-tower
> Bus layer located at address 2
> IR layer located at address 6
> Sensor layer located at address 10
> EEPROM layer located at address 12
> Clock layer located at address 14
> CompactFlash layer located at address 16
> I2C layer located at address 18
> Tricolor layer located at address 20
> Display layer located at address 22
> Motor layer located at address 26
> PicProto layer located at address 30
> Servo layer located at address 58
> Done!
```

If we want to change the address of the Servo layer to "4", we can say the following:

```
set-board-address 58 4
```

Now let's scan the Tower again, just to make see that it worked:

```
scan-tower
> Bus layer located at address 2
> Servo layer located at address 4
> IR layer located at address 6
> Sensor layer located at address 10
> EEPROM layer located at address 12
> Clock layer located at address 14
> CompactFlash layer located at address 16
> I2C layer located at address 18
> Tricolor layer located at address 20
> Display layer located at address 22
> Motor layer located at address 26
> PicProto layer located at address 30
> Done!
```

Appendix F: Rabbit Foundation Documentation

Figure F.1 - The Rabbit foundation.

Description

The Rabbit foundation *(Figure F.1)* is built around a Rabbit 2200 processor module operating at 22.1 MHz. The foundation contains a serial programming header, as well as all of the power and support circuitry needed for basic operation. Every I/O pin from the processor is broken out to the main Tower connectors, and passed up and down the entire stack.

Hardware Detail

In addition to the pins used for I²C communication, each layer on the Tower stack has access to all of the I/O pins from the foundation. When the Rabbit's Ethernet module is in use, there are 14 free I/O pins, 6 input-only pins, 2 output-only pins, and 2 serial-port pins which are passed to every layer through the main Tower connectors. The connectors are 18-pin surface mount Hirose connectors. Each board has female connectors on the top, and male connectors on the bottom, so stacking layers is a simple process. In addition to the I/O pins, two power busses and a ground line are also available to every layer. The pin configuration for the connectors is detailed below *(Figure F.2)*:

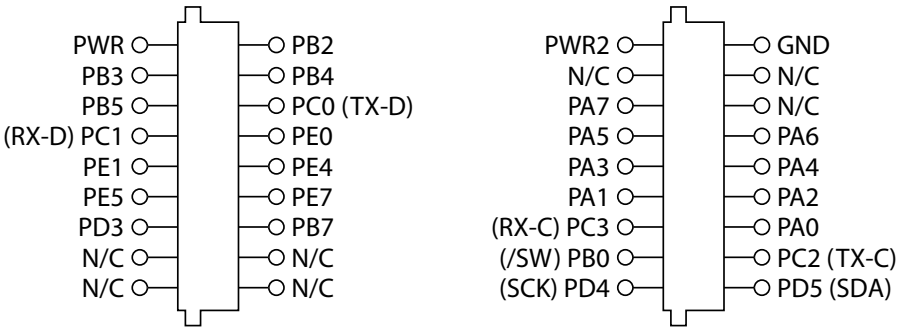


Figure F.2 - Tower connectors pin configuration.

**Note: Pins PD4 and PD5 are actually serial port B, which is translated to SCK and SDA by an on-board PIC.*

The foundation itself as well as many of the layers, obtain their power only off of the primary power bus. The primary bus can be powered by either four AA batteries, or through a 1mm wall jack also located on the foundation. The Rabbit foundation can accept only rechargeable batteries, due to their slightly lower voltage. As an alternative to batteries, the wall jack can be used. It must have a 1mm negative-tip connector, providing at least 6 volts, since it gets regulated down to a tight 5 volts, and needs a bit of overhead to do so.

Some layers that consume more power have the option of using either the primary of secondary bus. To put power on the secondary bus, a cable is needed that plugs into the 2-pin power connector on the Foundation. This connector is a standard header, with the pinout shown below *(Figure F.3)*:

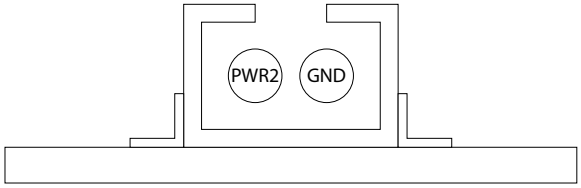


Figure F.3 - The foundation power connector.

Both power busses are switched on and off using the same power switch, located at the top edge of the Foundation. The switch is three-way, with the middle position being "off", and the two sides selecting between wall or battery power for the primary bus. Regardless of what source the primary bus is using, the secondary bus will always use whatever is provided to it through the connector, but can still be turned off by ensuring that the switch is in the middle position.

The foundation can be programmed through the on-board serial-programming header when connected to a serial cable with an RS-232 module on the end. The pin configuration of the header on the board is shown below *(Figure F.4)*:

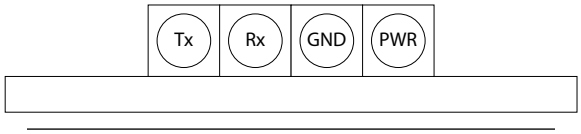


Figure F.4 - The foundation serial connector.

In addition to downloading Logo code, it is also possible to download Assembly code. To download assembly, the board must be powered on while the Button is depressed. At that point, Assembly code can be downloaded through the Tower Development Environment. After downloading new assembly code, it is necessary to power-cycle the layer before the new code will run.

In rare cases, when new low-level machine code must be downloaded to the Rabbit foundation, the commercially-available Rabbit programming kit is required. A programming header for this connection is located directly on the Rabbit processor module itself.

Foundation Code

For information on all of the primitives available in the virtual machine, please refer to the Rabbit Logo Language Reference document in Appendix C. There are however, four include files that pertain directly to functions on the foundation, which will be discussed here.

The standard include file contains the definitions for commonly used constants, as well as functions for switching processor pins between input and output states, determining min/max relationships, clearing an array, and viewing the entire contents of strings and arrays.

The basic constant declarations are useful for talking directly to frequently accessed registers such as the I/O ports. The declarations themselves are shown here:

```
constants
    [[porta 0x30]
     [portb 0x40]
     [portc 0x50]
     [portd 0x60]
     [porte 0x70]
     [spcr 0x24]
     [pcfr 0x55]
     [pdfr 0x65]
     [pefr 0x75]
     [pddcr 0x66]
     [pdddr 0x67]
     [peddr 0x77]]
```

While controlling I/O pins is as simple as just setting and clearing bits, an important step that users often forget, is that they need to first ensure that the pins are configured as outputs. Functions have been defined to simplify the process of switching various Rabbit I/O pins between input and output status.

```
to set-porta-output
    write-port spcr 0x84
end

to set-porta-input
    write-port spcr 0x80
end

to set-portd-bit-output :bit
    set-bit :bit pdddr
end

to set-portd-bit-input :bit
    clear-bit :bit pdddr
end

to set-porte-bit-output :bit
    set-bit :bit peddr
end

to set-porte-bit-input :bit
    clear-bit :bit peddr
end
```

Additionally, a **flash-pin** function has been defined to turn a pin on and off several times. This is useful for debugging complex operations, by simply connecting an LED to a pin and making at flash at various points within the program.

```
to flash-pin :pin :port
    repeat 3 [set-pin :pin :port wait 1
              clear-pin :pin :port wait 1]
end
```

Two functions, **min** and **max**, have also been defined to return the lower or higher of two numerical input values, to simplify some of the more complex math routines that may be written:

```
to min :a :b
    if :a > :b [output :b]
    output :a
end

to max :a :b
    if :a < :b [output :b]
    output :a
end
```

When working with arrays, it is often useful to set every element to zero. The **clear-array** function works by obtaining the maximum defined length of the array, and then setting that number of elements to zero.

```
to clear-array :str
    let [pos 0]
    loop
    [
        if not (:pos < strmaxlen :str) [stop]
        setchar :str :pos 0
        make "pos :pos + 1
    ]
end
```

Another important operation that can be performed on strings and arrays, is printing out a list of the values stored in them. The **show-string-bytes** function will stop when it reaches a "0" character, indicating the end of a string. There are two functions for showing the contents of arrays, **show-array-bytes** to print a specified number of elements, and **show-all-array-bytes** to print the entire contents of the array.

```
to show-string-bytes :str
    let [pos 0]
    loop
    [
        print-number get-char :str :pos
        if (((get-char :str :pos) = 0) or
            (:pos > strmaxlen :str))
            [cr stop]
        print-string ",
        make "pos :pos + 1
    ]
end

to show-array-bytes :str :start :len
    let [pos :start]
    loop
    [
        if not (:pos < (:start + :len)) [cr stop]
        print-number get-char :str :pos
        print-string ",
        make "pos :pos + 1
    ]
end

to show-all-array-bytes :str
    show-array-bytes :str 0 strmaxlen :str
end
```

In addition to the other functions defined in the standard include file, a **nop** (no operation) function has also been defined, to allow for fine tuning of timing critical operations.

```
to nop
end
```

The print include file contains the functions needed for formatting and printing numbers, characters, and strings directly to the terminal.

The most basic print functions are used to print numbers directly to the terminal. The procedures **print** and **print-num** both just call the **put-serial-num** primitive. The main difference between **print** and **print-num** is that **print** inserts a carriage return after the number, by calling the **cr** function which just sends the ASCII sequence needed to jump to the next line in the terminal. Functions are also defined for printing a floating point number, printing a single ASCII character, and printing a text string.

```
to print :n
    put-serial-num 3 :n cr
end

to print-num :n
    put-serial-num 3 :n
end

to print-float :n
    put-serial-float 3 :n
end

to print-char :n
    put-serial 3 :n
end

to print-string :n
    put-serial-string 3 :n
end

to cr
    print-char 10 print-char 13
end
```

In some cases, we want to print numbers in hexadecimal format. .The **print-hex** function first prints a "$" character, and then splits the byte in half, and prints both the high and low nibbles using the **print-hex-digit** function, which simply prints the letters A through F corresponding to number 10 to 15.

```
to print-hex :n
    print-string "$
    print-hex-digit :n / 16
    print-hex-digit :n % 16
    cr
end

to print-hex-digit :n
    ifelse :n < 10
        [send :n + 48]
        [send :n + 55]
end
```

Finally, we have defined a **hi** function which just prints out a welcome string to the terminal. While not critically important for most programs, it serves as a good power-on test for the Tower, and can be called whenever you want to know if things are working properly.

```
to hi
    print-string
        "|  Welcome to Tower Development Kit|
    cr
end
```

The address include file contains the basic functions needed for scanning the Tower, and for changing the address of any layer connected to it.

The **get-board-id** function is used to determine which specific board is located at a given address. The function itself takes a single argument, the address to query, which must be an even number between 0 and 254. The return value will be the hardware identifier, which can be looked up in the get-board-name function, also present in this file, to obtain the name of the actual board. The function sends two values to the actual layer, a "255" to enter address mode, and a "0" indicating that we want to obtain the hardware identifier. The identifier is then read back from the board, ignoring the first value back, which will just be a "1" indicating that a single byte is going to follow.

```
to get-board-id :address
    no-multi
    [
        i2c-start
        i2c-write-byte :address
        i2c-write-byte 2
        i2c-write-byte 255
        i2c-write-byte 0
        i2c-stop
        i2c-start
        i2c-write-byte :address + 1
        ignore i2c-read-byte
        let [i2c-byte i2c-read-byte-last]
        i2c-stop
    ]
    output :i2c-byte
end
```

The **set-board-address** function is used to change the address of any layer plugged into the stack. The function takes two arguments, the old address, and the new address, both of which can be values from 0 to 254. However, it should be noted that "0" is the general-call-address, meaning that no layer should actually have it as its address. If a "0" is sent as the argument for the new address, the board will return to its default address, which is always defined to be exactly twice its hardware identifier. The function itself sends three values to the layer, a "255" to enter address mode, a "1" indicating that we want to write a new address, and then the new address itself, which must be an even number from 0 to 254.

```
to set-board-address :oldaddress :newaddress
    no-multi
    [
        i2c-start
        i2c-write-byte :oldaddress
        i2c-write-byte 3
        i2c-write-byte 255
        i2c-write-byte 1
        i2c-write-byte :newaddress
        i2c-stop
    ]
end
```

The **scan-tower** function is used to check every address on the Tower, and when a board is found, it prints a string to the terminal indicating that fact. Starting at address 0, each address is individually checked to see if a board is present, by the i2c-check primitive. If a board is present, the hardware identifier is obtained, and then looked up by the get-board-name function. The resulting string is then printed to the terminal, followed by a statement indicating the address at which it was found. This process then repeats until the entire address space has been checked.

```
to scan-tower
    let [n 0 board-id 0 nn 0]
    repeat 128
    [
        no-multi
        [
            i2c-start
            make "nn i2c-check :n
            i2c-stop
        ]
        if (:nn = 1)
        [
            make "board-id get-board-id :n
              print-string get-board-name :board-id
              print-string
                  "| layer located at address |
              print :n
        ]
        make "n :n + 2
    ]
    print-string "|Done!|
    cr
end
```

The **get-board-name** function is used to look up the name of a board, given its hardware identifier. The name is output directly as a string to the calling function. If no match is found, the phrase "Unknown" is returned.

```
to get-board-name :n
    if (:n = 1) [output "Bus stop]
    if (:n = 2) [output "Servo stop]
    if (:n = 3) [output "IR stop]
    if (:n = 4) [output "Voice stop]
    if (:n = 5) [output "Sensor stop]
    if (:n = 6) [output "EEPROM stop]
    if (:n = 7) [output "Clock stop]
    if (:n = 8) [output "CompactFlash stop]
    if (:n = 9) [output "I2C stop]
    if (:n = 10) [output "Tricolor stop]
    if (:n = 11) [output "Display stop]
    if (:n = 12) [output "RF stop]
    if (:n = 13) [output "Motor stop]
    if (:n = 15) [output "PICProto stop]
    if (:n = 16) [output "Serial stop]
    output "Unknown
end
```

The net include file contains the basic functions needed for establishing socket based network communications over TCP/IP, and sending and receiving data over those channels.

First, we have two functions for printing IP addresses. The **print-ip-address** function takes a 32-bit numerical representation and prints it as a string, and the **print-my-ip** function simply calls the first with the local IP address.

```
to print-ip-address :ip
    print-number (:ip >> 24)
                 bit-and 0xff print-string ".
    print-number (:ip >> 16)
                 bit-and 0xff print-string ".
    print-number (:ip >> 8)
                 bit-and 0xff print-string ".
    print-number :ip bit-and 0xff
end

to print-my-ip
    print-string "|my ip address is: |
    print-ip-address get-ip
    cr
end
```

Every device that uses standard networking hardware has a unique MAC address, which can be used to identify the device on a network. The **print-my-mac-addr** function is defined to print the local node's MAC address. A short string variable is also defined to hold the address while it is being printed.

```
strings [[mac-addr-buf 6]]

to print-my-mac-addr
    print-string "|my mac addr is: |
    let [index 0]
    fill-with-mac-address mac-addr-buf
    repeat 5
    [
        print-hex-number
            get-char mac-addr-buf :index
        make "index :index + 1 print-string ":
    ]
    print-hex get-char mac-addr-buf :index
end
```

Two functions, **sock-open** and **sock-accept**, have been written for opening and accepting socket connections. These functions use the lower-level primitives for opening and accepting socket connections, but also ensure that connections are full established before returning.

```
to sock-open :host :port
    let [tsock sock-open-init :host :port]
    let [timeout 30 start read-rtc]
    loop
    [
        if (sock-established-loop :tsock)
           [output :tsock]
        if read-rtc > (:start + :timeout)
           [output 0]
    ]
end

to sock-accept :port
    let[tsock sock-accept-init :port]
    loop
    [
        if (sock-established-loop :tsock)
           [output :tsock]
    ]
end
```

Finally, a **send-mail** function has been written to send an email message over a socket connection. The function takes arguments for the socket itself, to and from information, the message itself, and the subject line for it. Short delays are required at various points in the function to allow the mail data to be processed accordingly.

```
to send-mail :sock :to :from :subject :message
    wait 10
    sock-write :sock "|MAIL FROM:<|
    sock-write :sock :from
    sock-write :sock "|> |
    sock-writec :sock 13 sock-writec :sock 10
    wait 10
    sock-write :sock "|RCPT TO:<|
    sock-write :sock :to
    sock-write :sock "|> |
    sock-writec :sock 13 sock-writec :sock 10
    wait 10
    sock-write :sock "|DATA |
    sock-writec :sock 13 sock-writec :sock 10
    wait 10
    sock-write :sock "|From: |
    sock-write :sock :from
    sock-writec :sock 13 sock-writec :sock 10
    sock-write :sock "|To: |
    sock-write :sock :to
    sock-writec :sock 13 sock-writec :sock 10
    sock-write :sock "|Subject: |
    sock-write :sock :subject
    sock-writec :sock 13 sock-writec :sock 10
    sock-writec :sock 13 sock-writec :sock 10
    sock-write :sock :message
    sock-writec :sock 13 sock-writec :sock 10
    sock-writec :sock 46
    sock-writec :sock 13 sock-writec :sock 10
    sock-write :sock "|QUIT |
    sock-writec :sock 13 sock-writec :sock 10
end
```

Examples of Use

Let's start by saying "Hi" to our foundation:

```
hi
>     Welcome to Tower Development Kit
```

Now let's do a little simple math, just to show that everything is in working order:

```
print 3 + 5
> 8
```

How about we try out some basic pin I/O. For this example, we'll say that we have an LED connected to pin B0. Let's first set the pin to be an output:

```
set-porte-bit-output 0
```

Now we'll make the LED flash using the defined function:

```
flash-pin 0 porte
```

Before doing any network communication, we need to set up the IP information:

```
to tcp-init-garden-rabbit-1
      tcp-init
            "18.85.45.41      ;; ip address
            "255.255.255.0    ;; netmask
            "18.85.45.1       ;; gateway
            "18.85.2.138      ;; dns
end
```

Note that the IP address, netmask, gateway, and DNS specified above must be specified based on the local network the Rabbit is operating on.

Now that TCP/IP communication has been set up, let's open a socket and send a piece of email:

```
setn sock-open "outgoing.mit.edu 25
send-mail n "tgorton@mit.edu
            "scooby@mit.edu
            "|New Code|
            "|The new code is working great!|
sock-close n
```

We can also write a simple web server to return a text-string with a hit counter to any web browser that queries it:

```
strings [[hit-string 10]]

to web-server
    tcp-init-dhcp
    ignore launch [loop [tcp-tick]]
    let [hit-count 0]
    loop
    [
        let [sock sock-accept 80]
        make "hit-count :hit-count + 1
        sock-write :sock
            "|thank you for visiting!<br>|
        sock-write :sock
            "|you are visitor number |
        num-to-str hit-string :hit-count
        sock-write :sock hit-string
        sock-close :sock
    ]
end
```

Note that the string "hit-string" must be defined to hold the string representation of the hit-counter, since it is not possible for the **num-to-str** function to directly return a string datatype.

We can also play around with Tower addressing. Let's start by scanning the Tower like this:

```
scan-tower
> Bus layer located at address 2
> IR layer located at address 6
> Sensor layer located at address 10
> EEPROM layer located at address 12
> Clock layer located at address 14
> CompactFlash layer located at address 16
> I2C layer located at address 18
> Tricolor layer located at address 20
> Display layer located at address 22
> Motor layer located at address 26
> PicProto layer located at address 30
> Servo layer located at address 58
> Done!
```

If we want to change the address of the Servo layer to "4", we can say the following:

```
set-board-address 58 4
```

Now let's scan the Tower again, just to make see that it worked:

```
scan-tower
> Bus layer located at address 2
> Servo layer located at address 4
> IR layer located at address 6
> Sensor layer located at address 10
> EEPROM layer located at address 12
> Clock layer located at address 14
> CompactFlash layer located at address 16
> I2C layer located at address 18\
> Tricolor layer located at address 20
> Display layer located at address 22
> Motor layer located at address 26
> PicProto layer located at address 30
> Done!
```

Appendix G: Sensor Layer Documentation

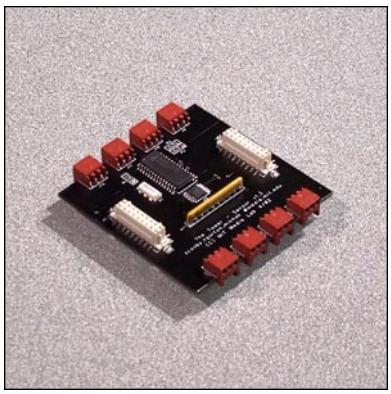


Figure G.1 - The Sensor layer.

Description

The Sensor layer *(Figure G.1)* has ports for eight resistive sensor inputs. By the use of a voltage divider, measurement values are converted by a 12-bit A/D converter. It is also possible to directly access power, ground, and the analog voltage-sensing lines directly from the ports as different sensing needs require.

Hardware Detail

The sensor ports are designed to allow for either resistive, or direct voltage-level readings. A schematic of the 3-terminal port is shown below *(Figure G.2)*:

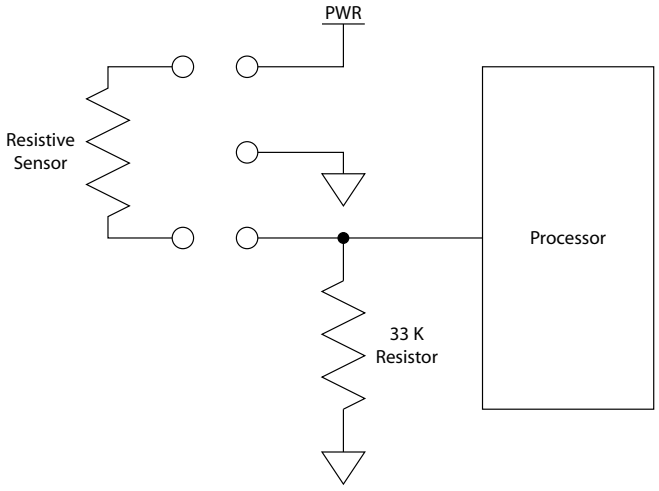


Figure G.2 - A schematic of a sensor port.

When looking at the port from the board edge, the pin configuration is as follows *(Figure G.3)*:

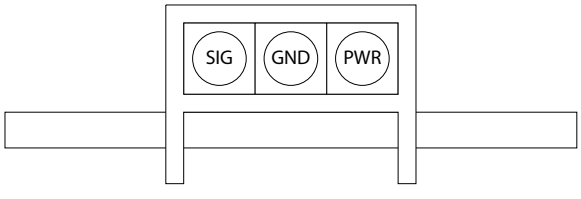


Figure G.3 - A sensor port connector.

To make a resistive measurement, the sensor is connected between the outer two pins, as shown in the diagram above. In conjunction with an on-board pull-down resistor, this creates a voltage divider, which is then read by a 12-bit A/D conversion chip, also located on the layer.

For direct voltage measurements, a ground-referenced voltage can be input into the SIG socket on the connector, the value of which will be returned normalized by the value of the power supply rail, a nominal 5 volts when using regulated wall power, however a bit higher when batteries are being used. It is important to note that this method of voltage measurement is not differential, and is therefore limited to the range between power and ground, and may not be ideal for some applications.

Layer Code

The include file for the sensor layer contains one function, which is used to return a 12-bit sensor value. *(This function is written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **sensor** function is called with one argument, the sensor port number to read, which is a value between 1 and 8. The function first sends two values to the Sensor layer, a "0" indicating that a Sensor reading is being requested (as opposed to an address command), and then the sensor number to be read, minus one, since the layer firmware is actually expecting a number between 0 and 7. After the request is sent to perform the conversion, the result is read out of the layer's transmit buffer in the form of two bytes. Those two bytes are reassembled into a single result by shifting the high byte to the left by 8 bits, and then performing a logical or with the low-byte result. Actually three bytes are read out, but the first is ignored, as it is known to be a "2", representing the number of arguments to follow.

```
to sensor :n
    i2c-start
    i2c-write-byte $0a
    i2c-write-byte 2
    i2c-write-byte 0
    i2c-write-byte (:n - 1)
    i2c-stop
    i2c-start
    i2c-write-byte $0b
    ignore i2c-read-byte 1
    seti2c-byte (left-shift i2c-read-byte 1 8)
    seti2c-byte i2c-byte or i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

Examples of Use

Using the Sensor layer is as simple as calling the "sensor" function, with a single argument representing the sensor port to be read. A simple example of how to print a sensor value to the console is shown as follows:

```
print sensor 1
> 1527
```

This code will first call the "sensor" function with an argument of "1". The result of that operation, a value between 0 and 4095, will then be passed to the print function, which then prints the value to the console. In the above sample, the value happened to be 1527.

If you want to do a conditional test of a sensor value, and depending on the value, do different things, a sample of how to do so is given here:

```
ifelse ((sensor 1) > 1000)
    [set red-led led-port]
    [set green-led led-port]
```

In this example, the value of the sensor in port 1 is read, and if its result is greater than 1000, a red LED is turned on. If the result is found to be less than 1000, a green LED is turned on.

Appendix H: DC Motor Layer Documentation

Figure H.1 - The DC Motor layer.

Description

The DC Motor layer *(Figure H.1)* can drive up to four DC motors at high currents. Each motor can be individually controlled to drive in either direction, at a variable speed, and can be easily accelerated or decelerated over a desired period of time.

Hardware Detail

A full H-Bridge circuit drives each motor, in order to allow bi-directional high-powered control. Two L293D integrated circuits are used to drive two motors each. The on-board PIC controls the enable pins on each H-Bridge, to allow for high-voltage velocity control. The chips themselves contain the flyback diodes needed to prevent damage to the chip due to the field collapse when the motor's inductive load is switched off rapidly.

The motor connectors themselves are 3-pin 0.1" female headers. The pin configuration of a port is shown below *(Figure H.2)*:

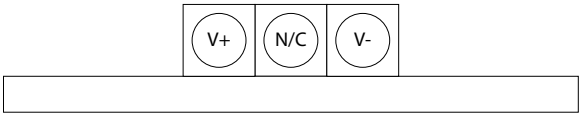


Figure H.2 - A DC motor port connector.

The motors can be powered off of either the primary or secondary bus as selected by the switch on the layer, and can be driven up to 36 volts if desired. A bicolor LED near each motor port will light up red or green depending on the power and direction that the motor is being driven.

Layer Code

The include file for the DC Motor layer contains four functions, one each for turning the motor forwards and backwards, one for stopping abruptly, and one for just cutting power and letting the motor coast to a stop. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **motor-on-forward** function is used to turn on a motor in the forward direction. The function

takes three arguments from the user, the motor number (a value from 1 to 4), the desired speed (from 0 to 255), and the time in hundredths of a second (0 to 255) that it should take to ramp up or down to the new speed setting. A total of four arguments are being sent to the layer itself. First a "0" is sent, indicating that motor forward drive function is being called. Then, the three arguments are sent in order, and the motor will assume its new state.

```
to motor-on-forward :n :speed :time
    i2c-start
    i2c-write-byte 26
    i2c-write-byte 4
    i2c-write-byte 0
    i2c-write-byte (:n - 1)
    i2c-write-byte :speed
    i2c-write-byte :time
    i2c-stop
end
```

The **motor-on-reverse** function is used to turn on a motor in the reverse direction. The function takes three arguments from the user, the motor number (a value from 1 to 4), the desired speed (from 0 to 255), and the time in hundredths of a second (0 to 255) that it should take to ramp up or down to the new speed setting. A total of four arguments are being sent to the layer itself. First a "1" is sent, indicating that motor reverse drive function is being called. Then, the three arguments are sent in order, and the motor will assume its new state.

```
to motor-on-reverse :n :speed :time
    i2c-start
    i2c-write-byte 26
    i2c-write-byte 4
    i2c-write-byte 1
    i2c-write-byte (:n - 1)
    i2c-write-byte :speed
    i2c-write-byte :time
    i2c-stop
end
```

The **motor-stop** function is used to cut power to the motor, but will still allow it to coast to a stop. The function takes a single argument, the motor number (a value from 1 to 4), and sends it to the layer, right after a "2" is sent, signifying a motor-stop call.

```
to motor-stop :n
    i2c-start
    i2c-write-byte 26
    i2c-write-byte 2
    i2c-write-byte 2
    i2c-write-byte (:n - 1)
    i2c-stop
end
```

The **motor-brake** function is used to immediately stop the motor from turning. By turning on both forward and reverse drive simultaneously, the voltage across the motor becomes fixed, resisting further motion by fighting against the voltage created by its turning. The function takes a single argument, the motor number (a value from 1 to 4), and sends it to the layer, right after a "3" is sent, signifying a motor-brake call.

```
to motor-brake :n
    i2c-start
    i2c-write-byte 26
    i2c-write-byte 2
    i2c-write-byte 3
    i2c-write-byte (:n - 1)
    i2c-stop
end
```

Examples of Use

Using the DC Motor layer is as simple as just turning a motor on and off. Let's say we want to turn on the motor in port number 1 going forwards. We could say something like this:

```
motor-on-forward 1 255 0
```

The first argument to the function is saying that we want to turn on motor 1. The 255 being sent is indicating that we want it to go to full power, and the 0 says that we want it to happen instantly. Now, let's cut the speed by half:

```
motor-on-forward 1 128 100
```

This time, we sent a value of 128 for the speed setting, which is about half of the 255 we sent before. For the third argument, we sent a value of 100, indicating that we wanted to it to ramp down to this new speed over the course of one second. It is important to note that the motor may actually be

spinning faster than half of its original speed. This is because the speed argument is actually a power setting, so if there is not a significant load on the motor, you won't see a huge speed change even if the power is cut in half.

Let's now tell the motor to go the other direction:

```
motor-on-reverse 1 255 200
```

When this code is run, the motor should switch directions, and begin ramping to full power in the opposite direction. The 200 argument at the end is telling it to take 2 seconds to reach that new speed. Let's cut power to the motor and let it coast to a stop:

```
motor-stop 1
```

The motor should slow down, and eventually stop. If you try moving the axle with your hand, you should find it easy to spin. For a comparison, let's turn the motor back on:

```
motor-on-forward 1 255 0
```

And now have it stop abruptly:

```
motor-brake 1
```

This time, the motor should stop spinning almost immediately. If you try to spin the axle by hand, you should notice that there is much more resistance to motion than there was in the previous case.

Appendix I: Servo Motor Layer Documentation

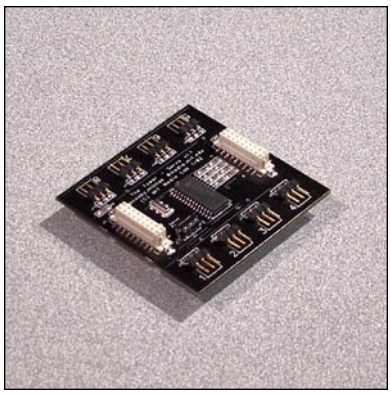


Figure I.1 - The Servo Motor layer.

Description

The Servo Motor layer *(Figure I.1)* has ports for eight standard servo-motor connections. Each servo can be independently turned on or off and can be set to any angle within a 100 degree range.

Hardware Detail

The Servo layer uses a PIC to generate the PWM (Pulse Width Modulation) signals needed to communicate with standard hobby servos. The connector is a 3-pin 0.1" header, compatible with Futaba J-Type servo connectors. The pin configuration of a port is shown below *(Figure I.2)*:

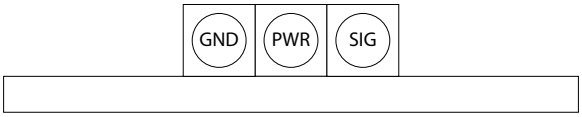


Figure I.2 - A servo motor port connector.

When powered on, there will be a pulse train present on the signal line, with a positive pulse width between 1ms and 2ms long. The width of the pulse corresponds directly to the servo angle, with a 1.5ms pulse aligning the servo to its center point.

Even though most servo-motors are capable of 180 degrees of rotation, they tend to have different offsets built in, so the range has been limited to 100 degrees to eliminate the possibility of unintentional damage to servos by over or under-driving them, and there is no need to calibrate a neutral position.

Layer Code

The include file for the Servo layer contains three functions, two for turning a servo on and off, and one for rotating the servo to the desired angle. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **servo-on** function takes one argument, the servo number to turn on, a value from 1 to 8. Within the function, a total of two values are actually sent to the servo layer. First an "8" is sent, indicating that we want to turn a servo on. Then, the actual servo number is sent. It should be noted that the servo

number has the number one subtracted from it before it is sent to the layer, since the layer is actually expecting a value from 0 to 7. The reason for the modulo 9, is that we want to ensure that the user cannot inadvertently send a value greater than 7 in that spot, to avoid possibly confusing the layer itself.

```
to servo-on :number
    i2c-start
    i2c-write-byte $04
    i2c-write-byte 2
    i2c-write-byte 8
    i2c-write-byte (:number % 9) - 1
    i2c-stop
end
```

The **servo-off** function operates similarly to the write function, taking one argument, the servo number to turn on, which is a value from 1 to 8. Within the function, a total of two values are actually sent to the servo layer. First a "9" is sent, indicating that we want to turn a servo off. Then, the actual servo number is sent. Just as in the previous function, the servo number is scaled down to the 0 to 7 range, and is forced to be kept in the desired range by the use of the modulus operator.

```
to servo-off :number
    i2c-start
    i2c-write-byte $04
    i2c-write-byte 2
    i2c-write-byte 9
    i2c-write-byte (:number % 9) - 1
    i2c-stop
end
```

The **turn-servo** function is what actually makes the servo move by changing the positive pulse width of the signal pin. The function takes two arguments, the servo number to be turned, and the angle that the servo should be turned to. The angle argument is represented in degrees, with a maximum swing from 0 to 100 degrees. If a value higher than 100 is sent, it will be interpreted just as 100 by the pulse modulation code on the layer. By default, all servos will start at the 0 degree point when they are turned on until they are changed.

```
to turn-servo :number :angle
    i2c-start
    i2c-write-byte $04
    i2c-write-byte 2
    i2c-write-byte (:number % 9) - 1
    i2c-write-byte :angle
    i2c-stop
end
```

Examples of Use

Probably the most common thing to do with the servo layer is to just turn on a servo and start moving it around. Let's start by turning on the servo plugged into port 3:

```
servo-on 3
```

Pretty simple, huh? Well, it's almost as easy to move it:

```
turn-servo 3 50
```

The servo motor should now turn to its midpoint, since 50 is halfway between 0 and 100. We can write a simple loop to make the servo swing back and forth between its endpoints:

```
loop [turn-servo 3 0 wait 5
      turn-servo 3 100 wait 5]
```

The servo motor will now start moving between its two extremes, switching positions every half second. The speed at which it turns is not user controllable, and is set by the actual servo itself. It is possible to buy servos of a variety of powers and speeds, with the tradeoff being that the faster servos are generally much weaker. If a servo is too fast, it's always possible to slow it down manually, similar to this:

```
setn 0
repeat 101 [turn-servo 3 n wait 5 setn n + 1]
```

This code first sets the global variable "n" to 0. Then, it loops 101 times, sending a turn-servo command with n as its angle argument, incrementing n each time. This would cause the servo to turn one

degree every half second, giving the illusion of a much slower movement. We looped 101 times in order to have both the 0 and 100 positions included.

If you remember, we said that every servo is at the "0" angle position when it is turned on. In some applications, that can cause a problem, depending on how things are set up in the mechanical system. If you want to have a servo be at a different angle when it is turned on, its as simple as just calling the turn-servo function before the servo-on function:

```
turn-servo 3 75
servo-on 3
```

In this case, the servo will be at a 75 degree angle when it is turned on. The last thing we might want to do, is turn the servo off:

```
servo-off 3
```

Turning off the servo will not cause the servo to move, it will simply stop driving power into it. If nothing is pushing on the servo, it will stay exactly at the angle it had been at. However, it's easy to rotate by hand when it is turned off. For a comparison, try turning a servo by hand while it is turned on. You should find it very difficult, and feel the motor actively pushing back against your efforts.

Appendix J: EEPROM Layer Documentation

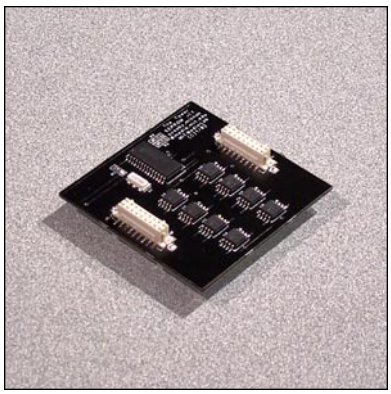


Figure J.1 - The EEPROM layer.

Description

The EEPROM layer *(Figure J.1)* contains a bank of eight, 256-kilobit EEPROM chips, for a total memory of 256 kilobytes. The memory is useful storing values in data-collection applications.

Hardware Detail

There are eight, individually addressable EEPROM chips on the layer. While all chips communicate via the I2C serial protocol, it is important to note that they are not on the main Tower serial bus. To avoid address conflicts with the rest of the system, the PIC processor on the layer independently communicates with them using two of its standard I/O lines.

Layer Code

The include file for the EEPROM layer contains two functions, one for writing a byte to EEPROM, and one for reading a byte. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **ee-write** function takes three arguments, the chip number, memory location, and the data value to be written. The chip number is given as a value from 0 to 7, and the memory location can be anywhere from 0 to 32768. A total of five arguments are being sent to the EEPROM layer. First a "0" is sent, indicating that a write function is being performed. Then, the chip number, memory location, and data values are sent, with the memory location argument being split into high and low byte portions to allow addressing of all memory locations. The two-millisecond wait at the end of the function is to ensure that the internal memory write cycle properly terminates. Without the wait, back-to-back function calls could potentially cause memory write failures.

```
to ee-write :chipnumber :addr :data
    i2c-start
    i2c-write-byte $0c
    i2c-write-byte 5
    i2c-write-byte 0
    i2c-write-byte :chipnumber
    i2c-write-byte highbyte :addr
    i2c-write-byte lowbyte :addr
    i2c-write-byte :data
    i2c-stop
    mwait 2
end
```

The **ee-read** function operates similarly to the write function, sending the chip number and memory location as arguments to the layer. The first argument in this case however, is a "1", signifying a memory read. After the request is sent to perform the memory read, the result is read out of the layer's transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to ee-read :chipnumber :addr
    i2c-start
    i2c-write-byte $0c
    i2c-write-byte 4
    i2c-write-byte 1
    i2c-write-byte :chipnumber
    i2c-write-byte highbyte :addr
    i2c-write-byte lowbyte :addr
    i2c-stop
    i2c-start
    i2c-write-byte $0d
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

Examples of Use

To use the EEPROM layer, a user would first want to write a value or string of values to memory, and then read them out at a later time. A simple example of how to write a number to memory is shown below:

```
ee-write 7 23 80
```

This code will write the value 80 to memory location 23, on chip 7. To test that the write worked correctly, one could do something like this to read the value back out:

```
print ee-read 7 23
> 80
```

The result of the read operation, which in this case should be 80, will then be passed to the print function, which then prints the value to the console. If you want to take a number of sensor readings, write them to memory, and then later read them out, a sample of how to do so is given here:

```
setn 0
repeat 5 [ee-write 0 n ((sensor 1) / 16)
          wait 10 setn n + 1]

setn 0
repeat 5 [print ee-read 0 n setn n + 1]
> 24
> 81
> 126
> 187
> 244
```

For this sample code to work, a Sensor layer must also be present on the Tower. In this example, the global variable "n" is first set to zero, then 5 values of the sensor in port 1 are read, and stored to memory, with "n" representing the memory location being written to, which is incremented after each iteration. Since the sensor reading is a 12-bit number, we are dividing by 16 in order to store meaningful data in the 8-bit memory locations. The wait in the loop causes a half-second delay between sensor readings, to allow for a more interesting variety of sensor values. With the current delay, it will take 5 seconds to capture the 5 sensor values. After the data has been stored, we again reset the value of the global "n". With another repeat loop of 5 iterations, we print out the stored values to the console one at a time, using n as our memory location counter as in the write loop.

Appendix K: CompactFlash Layer Documentation

Figure K.1 - The CompactFlash layer.

Description

The CompactFlash layer *(Figure K.1)* is used to communicate with CompactFlash I and II cards, as well as Microdrive units, to provide a vast amount of data storage when needed for storing significant sensor data, binary data files, or even providing full filesystem support to the Tower.

Hardware Detail

A reduced-IDE interface is used to communicate with the cards, minimizing the complexity required for hardware connectivity. Data can be written to and read from the card on a sector-by-sector basis. It is necessary to read and write full sectors, as partial operations will leave the card itself in a bad state. When using the higher capacity Microdrives, it may be necessary to switch the card power to the secondary power bus, as spinning up a Microdrive uses a significant amount of power that may be unavailable from the primary battery power source.

Layer Code

The include file for the CompactFlash layer contains eight functions, one checking to see if a card is in the slot, one each for turning the card power on and off, one for initializing a card to communicate with, one each for starting a sector write or a sector read, and one each for actually writing or reading a byte within the sector. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **cf-is-there** function does not take any arguments, and will return a value of either 1 or 0 depending on whether or not a card is currently inserted in the slot. The function since a single value to the layer, a "0" indicating that a query is being made. The result is read out and returned, ignoring the first byte back, which is simply the number of arguments to follow, since it is known to be one.

```
to cf-is-there
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 1
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte $11
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **cf-power-on** function takes no arguments, and is used to power on the CompactFlash card itself. Since the cards can draw a lot of power, it is advantageous to keep them powered down when not actively accessing them. The function sends a single value of "1" to the layer, indicating the desired command.

```
to cf-power-on
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 1
    i2c-write-byte 1
    i2c-stop
end
```

The **cf-power-off** function takes no arguments, and is used to power off the CompactFlash card itself. Since the cards can draw a lot of power, it is advantageous to keep them powered down when not actively accessing them. The function sends a single value of "2" to the layer, indicating the desired command.

```
to cf-power-off
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 1
    i2c-write-byte 2
    i2c-stop
end
```

The **cf-init** function takes no arguments, and is used to initialize the compact CompactFlash card after it has been powered on. The initialization routine sets up internal buffering and cylinder information, and prepares the card to be accessed. The function sends a single value of "3" to the layer, indicating the desired command.

```
to cf-init
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 1
    i2c-write-byte 3
    i2c-stop
end
```

The **cf-start-write-sector** function initializes a sector-write operation, taking one argument, the sector number to begin at. The first argument sent to the actual layer in this case however, is a "4", signifying a sector write operation. When the layer receives the request to begin the write, it sets the cylinder and head positions as needed to point to the desired sector. It is important to note that sectors must be written in their entirety, so 512 bytes must be written following this command, or the card itself will be left in a bad state.

```
to cf-start-write-sector :sector
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 4
    i2c-write-byte 4
    i2c-write-byte 0
    i2c-write-byte :sector / 256
    i2c-write-byte :sector % 256
    i2c-stop
end
```

The **cf-start-read-sector** function initializes a sector-read operation, taking one argument, the sector number to begin at. The first argument sent to the actual layer in this case however, is a "5", signifying a sector read operation. When the layer receives the request to begin the read, it sets the cylinder and head positions as needed to point to the desired sector. It is important to note that sectors must be read in their entirety, so 512 bytes must be read out following this command, or the card itself will be left in a bad state If only a specific byte within the sector is needed, it's still necessary to issue a fixed number of dummy-reads to increment the read pointer to the desired position, read that byte, and then finish off the sector with the necessary number of dummy-reads to bring the total to 512 bytes.

```
to cf-start-read-sector :sector
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 4
    i2c-write-byte 5
    i2c-write-byte 0
    i2c-write-byte :sector / 256
    i2c-write-byte :sector % 256
    i2c-stop
end
```

The **cf-write** function is used following a cf-start-write-sector command to actually write data to the CompactFlash card. The function takes a single argument, the data byte to be written, and then sends a "6", signifying a data write operation, followed by the byte itself.

```
to cf-write-byte :data
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 2
    i2c-write-byte 6
    i2c-write-byte :data
    i2c-stop
end
```

The **cf-read** function is used following a cf-start-read-sector command to actually read data from the CompactFlash card. The function takes no arguments, and just sends a "7" to the layer itself, signifying a data read operation. After the request is sent to perform the read, the result is read out of the layer's transmit buffer as a single byte, and returned to the calling function. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to cf-read-byte
    i2c-start
    i2c-write-byte $10
    i2c-write-byte 1
    i2c-write-byte 7
    i2c-stop
    i2c-start
    i2c-write-byte $11
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

Examples of Use

To use the CompactFlash layer, a good place to start it to make sure that a card is present in the slot. We can check for a card by using the cf-is-there function like this:

```
waituntil [cf-is-there]
          print-string "|Card is present.|
> Card is present.
```

In the code above, we simply wait until the cf-is-there function returns true, and then print the desired string to the terminal, letting us know that a card has been inserted. Before we can write to or read from the card, we need to power it on and initialize it. That's as simple as:

```
cf-power-on
cf-init
```

Don't forget to power on the card before you try to initialize it, or it won't work properly. As soon as the card is initialized, lets write some data to it:

```
cf-start-write-sector
setn 0
repeat 512 [cf-write n setn n + 1]
```

We've just written an entire sector worth of data. All we had to do was call the cf-start-write-sector function, and then write 512 bytes. The data we wrote is an incrementing sequence from 0 to 255 repeated twice. Even though the variable n will continue past 255 to 511, cf-write only writes a single byte, which by default will use just the lowest byte of the number given to it, which ranges from 0 to 255. Let's try reading that data back out to make sure it looks okay:

```
cf-start-read-sector
repeat 512 [print cf-read]
> 0
> 1
> 2
(lots more data)
> 255
> 0
> 1
(lots more data)
> 254
> 255
```

The code to read out is pretty straightforward. Almost like writing a sector, we just call cf-start-read-sector, and then read out 512 bytes. Finally, let's power down the card since we're not going to be using it for awhile, and don't want to waste those precious batteries:

```
cf-power-off
```

Currently, only low-level reads and writes as shown above have been implemented. We are in the process of implementing a full FAT-32 filesystem on the layer, to make it even easier to transfer large amounts of data between Towers, desktop computers, and other CompactFlash-enabled devices.

Appendix L: IR Layer Documentation

Figure L.1 - The IR layer.

Description

The IR layer *(Figure L.1)* can be used for short-range, wireless communications. The hardware is fully IRDA compliant and thus can be used to communicate with any device speaking over this standard hardware protocol.

Hardware Detail

The IR layer uses an Agilent HSDL-3610 IRDA emitter/detector pair. IR data is can be sent and received at any baud rate from 2400 to 57600 bps through the UART port on the on-board PIC, and is translated to the necessary IR waveforms by an Agilent HSDL-7001. At full strength, signals can be transmitted up to about 8 feet. There are user-settable power settings, which allow anyone to reduce the strength of the signal transmission to either 2/3 or 1/3 of full power if desired, or turned off entirely, to avoid unintentional communication with other nearby compliant devices.

Layer Code

The include file for the IR layer contains five functions, one each for sending and receiving bytes, one to see if new data has been received, one for adjusting the transmission power, and a fifth for setting the baud rate. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **new-ir?** function is used when you want to find out if a new data byte has been received over IR. It is almost always desired to check first before reading from the register, to avoid unintentional double-reads of the same data. The function takes no arguments, and will return either a "1" or a "0", with a 1 signifying that new data is present, and a zero meaning otherwise. The function itself first sends a single value to the layer, a "0" indicating that it wants to perform the query. The result of "1" or "0" is then read back from the layer and output.

```
to new-ir?
    i2c-start
    i2c-write-byte $06
    i2c-write-byte 1
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte $07
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

Once we know that new data is available, we can use the **get-ir** function to obtain the actual value that was received. The function takes no arguments, and sends a single value of "1" to the layer to indicate the desired operation. The result is then read back from the layer and output, again ignoring the first byte back, which is simply the number of bytes to follow, which in this case we know is one.

```
to get-ir
    i2c-start
    i2c-write-byte $06
    i2c-write-byte 1
    i2c-write-byte 1
    i2c-stop
    i2c-start
    i2c-write-byte $07
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The other key thing that we will probably want to do with the IR layer is to send data out. This can be done using the **put-ir** function. The function takes a single argument, the byte to be sent. Two values are sent to the layer, first a "2" indicating that data is to be sent out, and then the value itself. As soon as the layer receives the value, it will immediately be sent out over the IR channel.

```
to put-ir :n
    i2c-start
    i2c-write-byte $06
    i2c-write-byte 2
    i2c-write-byte 2
    i2c-write-byte :n
    i2c-stop
end
```

If we want to adjust the IR transmission power, we can use the **ir-setpower** function. The function takes a single argument, the desired power. The power setting can range from 0 to 3, with 0 corresponding to a disabled transmitter, and 3 setting the transmitter to full power in order to obtain maximum range. The function sends two values to the layer, a "3" to enter the power adjust mode, and then the actual power setting itself.

```
to ir-setpower :power
    i2c-start
    i2c-write-byte $06
    i2c-write-byte 2
    i2c-write-byte 3
    i2c-write-byte :power
    i2c-stop
end
```

Finally, the **ir-setbaud** function can be used to adjust the baud rate of the transmitter and receiver. The function takes a single argument, a value from 0 to 5 corresponding to the desired baud rate. The available baud rates are shown in the table below *(Table L.1)*:

Control Value	Baud Rate
0	2400
1	4800
2	9600
3	19200
4	38400
5	57600

Table L.1 - IR baud rate control values.

Two arguments are sent to the actual layer, a "4" signifying a baud rate set command, and then the desired value itself. At power-on, the layer will be operating at 38400 baud.

```
to ir-setbaud :baud
    i2c-start
    i2c-write-byte $06
    i2c-write-byte 2
    i2c-write-byte 4
    i2c-write-byte :baud
    i2c-stop
end
```

Examples of Use

The two most basic things to do with the IR layer, are sending and receiving bytes. Sending a byte is very simple:

```
put-ir 23
```

We've just sent the data byte 23 out over the IR channel. If we want do read that value with another Tower, we could do something like this:

```
print get-ir
> 23
```

This method works well, assuming that we know to read the byte on the receiver after we find out that it's been sent. But what if we don't know that? We probably want to make our receiver program wait until new data has been received, and then read it out. Remember the new-ir? function? We can use that to do just what we want to:

```
waituntil [new-ir?] print get-ir
> 23
```

In this case, we wait until the new-ir? function returns a true value of 1, meaning that new data has been received. As soon as that happens, we print out the result. We've now got a reliable way of sending and receiving data through the IR channel.

If we're operating in a room full of IR devices, and want to shorten the transmission range to ensure that we're only talking to the one we want, we can reduce the transmission power like this:

```
ir-setpower 1
```

We've just reduced the transmission power to 1/3 of it's full value. We can just as easily set it to any of the available power settings or turn it off entirely with a power argument of "0".
Finally, let's try changing the transmission baud rate to 19200 bps:

```
ir-setbaud 3
```

According to the table, an argument of "3" corresponds to 19200 bps, so we should now be able to communicate with any device operating at the same speed.

Appendix M: Clock Layer Documentation

Figure M.1 - The Clock layer.

Description

The Clock layer *(Figure M.1)* contains a real-time-clock integrated circuit, as well as batteries to back up stored time and date information. The clock is capable of remembering year, month, date, day, hour, minute, and second, and can easily be used as a timer event-trigger for time-driven processes.

Hardware Detail

The Maxim MAX6900 real-time-clock chip communicates with the on-board PIC via the I^2C serial protocol, but it is important to note that it is not on the main Tower serial bus. To avoid address conflicts with the rest of the system, the PIC processor on the layer independently communicates with the clock chip using two of the PIC's standard I/O lines. Two 20mm 3-volt coin cell batteries are also on-board to provide clock backup for an estimated 10-15 years, so that the time data will not have to be re-entered every time the Tower is powered on. These batteries are type CR2032. Please note that the layer will not function without these batteries installed, even when the Tower is powered.

Layer Code

The include file for the Clock layer contains eight functions, two low-level read/write functions for directly accessing registers on the clock chip, two higher-level functions for setting and getting the current date and time, a function for starting a timer, a function for getting the timer value, and two lookup table functions, one for the day of the week, and another for the month of the year. In addition, three global variables are defined within the file, which are used exclusively by the timer functions to track the amount of elapsed time. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

For reading and writing data, it is important to know what is contained in each clock memory location. A table of the register assignments is shown here *(Table M.1)*:

Memory Location	Description
$00	Seconds
$01	Minutes
$02	Hours
$03	Date
$04	Month
$05	Day
$06	Year
$07	Control
$08	*(Unused)*
$09	Century

Table M.1 - The Clock chip memory map.

The **clock-write** function takes two arguments, the clock memory location, and the data value to be written. The clock memory location contains a specific field corresponding to the current time, and can be a value from 0 to 9. A total of three arguments are being sent to the Clock layer. First a "0" is sent, indicating that a write function is being performed. Then, the memory location and data value are sent. It is important to note that all data values are stored in BCD (Binary Coded Decimal) format, where the high four bits represents the tens digit of the value, and the low four bits represent the ones digit. The conversion between decimal and BCD is done in the following two functions as the data value is sent or received.

```
to clock-write :addr :data
    i2c-start
    i2c-write-byte $0e
    i2c-write-byte 3
    i2c-write-byte 0
    i2c-write-byte :addr
    i2c-write-byte ((:data / 10) * 16) +
                    (:data % 10)
    i2c-stop
end
```

The **clock-read** function operates similarly to the write function, sending the clock memory location as an argument to the layer. The first argument in this case however, is a "1", signifying a memory read. After the request is sent to perform the clock memory read, the result is read out of the layer's transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to clock-read :addr
    i2c-start
    i2c-write-byte $0e
    i2c-write-byte 2
    i2c-write-byte 1
    i2c-write-byte :addr
    i2c-stop
    i2c-start
    i2c-write-byte $0f
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output ((i2c-byte / 16) * 10) +
            (i2c-byte % 16)
end
```

The **set-time** function is used to set the entire time and date memory at the same time. It takes seven total arguments, corresponding to each data field for the time registers, and then simply calls the clock-write function with these arguments and the corresponding clock memory locations. The arguments are, in order, the year (which will be internally split into separate year and century values), the day of the week (Sunday=1 to Saturday=7), the month (January=1 to December=12), the day of the month, the hours (24-hour time), the minutes, and the seconds.

```
to set-time :year :day :month :date
            :hour :minute :second
    clock-write 0 :second
    clock-write 1 :minute
    clock-write 2 :hour
    clock-write 3 :date
    clock-write 4 :month
    clock-write 5 :day
    clock-write 6 (:year % 100)
    clock-write 9 (:year / 100)
end
```

The **get-time** function is used to print out all of the day and time information to the console in a single, easy to read sentence. The function essentially reads out all of the values from the timekeeping registers, and formats them nicely using a number of print and print-string function calls. The type function is similar to print, only it does not put a carriage return at the end of the number when it prints it. Look-up functions are used to get the string values for the day of the week and month names. Those functions are defined further down in this document.

```
to get-time
    print-string "|Today is |
    print-string get-day clock-read 5
    print-string "|, |
    print-string get-month clock-read 4
    print-string "| |
    type clock-read 3
    print-string "| |
    type ((clock-read 9) * 100) + clock-read 6
    print-string "|, |
    setn clock-read 2
    ifelse (n = 0)
        [type 12]
        [
            ifelse (n > 12)
                [type n - 12]
                [type n]
        ]
    print-string "|:|
    print-clock clock-read 1
    print-string "|:|
    print-clock clock-read 0
    print-string "| |
    ifelse (n > 11)
        [print-string "PM]
        [print-string "AM]
    cr
end
```

The **get-day** function is the look-up function for the name of the day of the week. The function takes a single argument, the number corresponding to the day, which ranges from 1-7. A series of if-statements are used to check the argument value and when a match is found, the text string is output.

```
to get-day :n
    if (:n = 1) [output "Sunday stop]
    if (:n = 2) [output "Monday stop]
    if (:n = 3) [output "Tuesday stop]
    if (:n = 4) [output "Wednesday stop]
    if (:n = 5) [output "Thursday stop]
    if (:n = 6) [output "Friday stop]
    if (:n = 7) [output "Saturday stop]
    output "Unknown
end
```

The **get-month** function is the look-up function for the name of the month. The function takes a single argument, the number corresponding to the month, which ranges from 1-12. A series of if-statements are used to check the argument value and when a match is found, the text string is output.

```
to get-month :n
    if (:n = 1) [output "January stop]
    if (:n = 2) [output "February stop]
    if (:n = 3) [output "March stop]
    if (:n = 4) [output "April stop]
    if (:n = 5) [output "May stop]
    if (:n = 6) [output "June stop]
    if (:n = 7) [output "July stop]
    if (:n = 8) [output "August stop]
    if (:n = 9) [output "September stop]
    if (:n = 10) [output "October stop]
    if (:n = 11) [output "November stop]
    if (:n = 12) [output "December stop]
    output "Unknown
end
```

The following two functions are used to set and read values from a long-term timer. While the built-in timer on the foundation is great for timing things on the sub-second level, this timer implementation uses the Clock layer to provide timer values anywhere from seconds to hours of elapsed time. To use the timer, you must first reset it, by calling the **start-timer** function. This function takes no arguments, and simply sets the values of three global variables to the current clock values to use as a reference point.

```
globals [timer-hour timer-min timer-sec]

to start-timer
    settimer-hour clock-read 2
    settimer-min clock-read 1
    settimer-sec clock-read 0
end
```

The current timer value can be obtained by calling the **get-timer** function. This function takes no arguments, and returns the elapsed time (in seconds).

```
to get-timer
    output (((clock-read 2) - timer-hour) * 3600)
           + (((clock-read 1) - timer-min) * 60)
           + ((clock-read 0) - timer-sec)
end
```

Examples of Use

To use the Clock layer, a user would first want to check what time is currently stored on the chip. If the layer has been used before, a value may already be saved in the clock registers, which would still be present due to the on-board battery backups. Calling get-time will print out the current time data.

```
get-time
> Today is Sunday, January 1 1970, 12:07:43 AM
```

Obviously, this time data is probably not accurate, and the clock was probably reset recently. To set the clock, we use the set-time function as shown below:

```
set-time 2002 6 7 23 19 43 15
get-time
> Today is Friday, July 23 2002, 7:43:15 PM
```

This code will set the time registers to correspond to the desired date and time, and then read it back out to ensure that it was stored properly. The order of arguments is important, and is explained in detail in the function definition earlier in this document.

To use the timer functions, we must first reset it by calling start-timer.

```
start-timer
```

At any point in time after the timer has been started, get-timer can be used to see how much time has passed:

```
print get-timer
> 153
```

In this case, 153 seconds (that is, 2 minutes and 33 seconds) has passed since the time at which we started the timer.

The timer can easily be used to wait a fixed amount of time, as shown in the following example:

```
start-timer waituntil [get-timer > 92]
            print-string "|It's been 93 seconds.|
> It's been 93 seconds.
```

In this code, we reset the timer and wait for it to become the desired value. As soon as it reaches that point, the program continues. In the sample above, a string is then printed which tells us that the desired time interval is over.

Appendix N: Display Layer Documentation

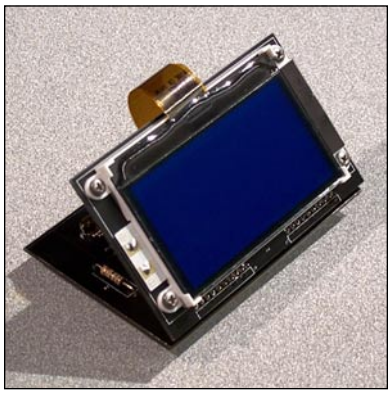


Figure N.1 - The Display layer.

Description

The Display layer *(Figure N.1)* uses a 128x64 pixel white-on-blue graphical LCD display module to provide powerful visual output capabilities for the Tower. Capable of text or graphics modes, users have the ability to send text strings, draw graphical objects, and even interact with individual pixels.

Hardware Detail

The Display layer is built around an Optrex 51320 LCD screen. The PIC processor interfaces directly with the EPSON SED-1565 LCD controller located directly on-glass via a 30-pin ribbon cable. The LCD backlight is powered through the metal connectors used to hold the two circuit boards together, and can be powered off of either the primary or secondary bus, depending on power constraints for a given application.

Layer Code

The include file for the Display layer contains ten functions, one for blanking the display, two each for reading and writing data directly to the display RAM, three for interfacing with pixels, one for printing text strings, and one for drawing a graphical line on the screen. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **display-clear** function takes no arguments, and is used to completely clear every pixel on the display simultaneously. The function simply sends a "0" to the layer, requesting that the operation be performed. The function then waits for a "1" to be sent back form the layer, indicating that the operation has been completed. Even though this is not necessary, it is there to ensure that the display is in a good state if we try to write to it immediately following the clear command.

```
to display-clear
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end
```

The **display-write-command**-byte function is used to send a command to the display. Commands are used to set the contrast, toggle display power on and off, enter and exit write-mode, and adjust other low-level system parameters. The function takes a single argument, the value to be sent, and sends the layer a "1" indicating that a command-byte write is being performed, followed by the value to be written.

```
to display-write-command-byte :data
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 1
    i2c-write-byte :data
    i2c-stop
end
```

The **display-write-data-byte** function is used to directly write data into the display RAM. This function is usually used following a display-write-command-byte, to send data after a write command has been initiated. The function takes a single argument, the value to be sent, and sends the layer a "2" indicating that a data-byte write is being performed, followed by the value to be written.

```
to display-write-data-byte :data
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 2
    i2c-write-byte :data
    i2c-stop
end
```

The **display-read-command-byte** function is used to read the status of the display. Information can easily be obtained regarding current power and contrast settings at any time during operation. The first argument in this case however, is a "3", signifying a status read. After the request is sent, the result is read out of the layer's transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to display-read-command-byte
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 3
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **display-read-data-byte** function is used to directly read data from the display RAM. This function is usually used following a display-write-command-byte, to determine what pixels are currently turned on in a specific region of the screen. The function takes no arguments, and sends the layer a "2" indicating that a data-byte read is being performed. This function is normally performed in conjunction with a display-write-data-byte call, to read out the contents of a RAM location, modify it, and write it back to its original place.

```
to display-read-data-byte
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 4
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **display-clear-line** function is used to erase one of the eight possible lines of text on the screen. The function takes a single argument, the line number (a value from 0 to 7). When the line is erased,

its counter will also be reset, so that the next characters printed to it will begin at the left side of the screen. After the line is erased, A value of "5" is sent to the layer, followed by the line number to be cleared. Since the Display layer itself takes a bit of time to complete this operation, the function waits for the layer to return a value of "1" indicating its successful completion.

```
to display-clear-line :line
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 5
    i2c-write-byte :line
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end
```

The next set of functions are used for printing strings, numbers, and graphics. For these functions to work correctly, four global variables must be defined within the file:

```
globals [disp-n disp-nn xdiff ydiff]
```

The first two variables are used as line and string counters, while the second two are used in the line-draw function to help with complicated math operations.

The **display-print-string** function is used to print a string of text on one of the eight available character lines on the display. The function takes two arguments, the line number (a value from 0 to 7), and the string itself, up to 21 characters in length. The string will begin printing at the location of the current line counter, so that the text will immediately follow anything that has already been printed on the line, and then increment the counter accordingly. The function first sends a "6" to the layer, indicating that a string-write is about to take place. The line number is then sent, followed by the string itself. The display-get-string-length function is used to determine the length of the string before it is sent. Since the Display layer itself takes a bit of time to complete this operation, the function waits for the layer to return a value of "1" indicating its successful completion.

```
to display-print-string :line :string
    setdisp-nn display-get-string-length :string
    i2c-start
    i2c-write-byte $16
    i2c-write-byte disp-nn + 2
    i2c-write-byte 6
    i2c-write-byte :line
    setdisp-n :string
    repeat disp-nn
    [
        i2c-write-byte read-prog-mem disp-n
        setdisp-n disp-n + 1
    ]
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end

to display-get-string-length :string
    setdisp-n :string
    loop
    [
        ifelse ((read-prog-mem disp-n) = 0)
            [output disp-n - :string]
            [setdisp-n disp-n + 1]
    ]
end
```

The **display-print-char** function is used to print a single character to the screen. The function takes two arguments, the line number (a value from 0 to 7), and the ASCII code of the character to print. The function sends three values to the layer itself, a "7" indicating that a character is to be printed, followed by the line and character arguments. The character will print at the current location of the line counter, directly after the last thing that was printed, and increment the counter accordingly.

```
to display-print-char :line :char
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 7
    i2c-write-byte :line
    i2c-write-byte :char
    i2c-stop
end
```

The **display-print-num** function is used to print a number to the screen. The function takes two arguments, the line number (a value from 0 to 7), and the number itself. The function divides the number into its individual digits, and calls display-print-digit as many times as needed to print the full number. The display-print-digit function actually chops out the desired digit, adds the number 48 to convert it to an ASCII code, and then calls display-print-char to actually print the digit.

```
to display-print-num :line :n
    if :n > 9999
       [display-print-digit :line :n 10000]
    if :n > 999 [display-print-digit :line :n 1000]
    if :n > 99 [display-print-digit :line :n 100]
    if :n > 9 [display-print-digit :line :n 10]
    display-print-digit :line :n 1
end

to display-print-digit :line :n :d
    display-print-char :line ((:n / :d) % 10) + 48
end
```

The **display-set-line-index** function is used to change the location of the line counter for a given line. This makes it possible to overwrite portions of a line without erasing the entire line itself. The function takes two arguments, the line number (a value from 0 to 7), and the desired character index (a value from 0 to 20). A total of three values are sent to the layer itself, beginning with an "8" indicating that an index set is being performed, and followed by the line and index arguments themselves.

```
to display-set-line-index :line :index
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 8
    i2c-write-byte :line
    i2c-write-byte :index
    i2c-stop
end
```

The **display-set-pixel** function is used to turn on a specific pixel on the screen. The function takes two arguments, the x (from 0 to 127) and y (from 0 to 63) coordinates of the pixel to turn on. The 0,0 point on the screen is the upper-left corner, and both x and y increment as we move towards the lower right. The function sends a "9" to the layer, indicating that a set-pixel command is being issued, followed by the x and y values themselves.

```
to display-set-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 9
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
end
```

The **display-clear-pixel** function is used to turn off a specific pixel on the screen. The function takes two arguments, the x (from 0 to 127) and y (from 0 to 63) coordinates of the pixel to turn on. The 0-0 point on the screen is the upper-left corner, and both x and y increment as we move towards the lower right. The function sends a "10" to the layer, indicating that a clear-pixel command is being issued, followed by the x and y values themselves.

```
to display-clear-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 10
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
end
```

The **display-test-pixel** function is used to find out if a specific pixel on the screen is currently turned on. The function takes two arguments, the x (from 0 to 127) and y (from 0 to 63) coordinates of the pixel to turn on. The 0,0 point on the screen is the upper-left corner, and both x and y increment as we move towards the lower right. The function sends an "11" to the layer, indicating that a set-pixel command is being issued, followed by the x and y values themselves. It then reads back the result, ignoring the first byte back, which is known to be a "1", indicating that one data value is being sent. The function returns a "0" if the pixel is currently turned off, and a "1" if it is on.

```
to display-test-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 11
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **display-draw-line** function is used to draw a line on the screen from any x-y coordinate, to any other x-y coordinate. The function takes four arguments, the starting x and y coordinates, and the ending x and y coordinates. The x-values can range from 0 to 127, and the y values from 0 to 63. The function uses a complex stack of if-statements to determine exactly how to draw the line based on its slope and direction. The details of the function are too complex to explain here, but you can try to follow it yourself and figure it out.

```
to display-draw-line :x1 :y1 :x2 :y2
    if (:x1 = :x2)
        [
            ifelse (:y2 > :y1)
                [
                    setdisp-n 0
                    repeat (:y2 - :y1) + 1
                        [display-set-pixel
                            :x1 :y1 + disp-n
                         setdisp-n disp-n + 1]
                    stop
                ]
                [
                    setdisp-n 0
                    repeat (:y1 - :y2) + 1
                        [display-set-pixel
                            :x1 :y2 + disp-n
                         setdisp-n disp-n + 1]
                    stop
                ]
        ]
```

```
        if (:y1 = :y2)
            [
                ifelse (:x2 > :x1)
                    [
                        setdisp-n 0
                        repeat (:x2 - :x1) + 1
                            [display-set-pixel
                                 :x1 + disp-n :y1
                             setdisp-n disp-n + 1]
                        stop
                    ]
                    [
                        setdisp-n 0
                        repeat (:x1 - :x2) + 1
                            [display-set-pixel
                                 :x2 + disp-n :y1
                             setdisp-n disp-n + 1]
                        stop
                    ]
                ]
        setxdiff (:x2 - :x1)
        setydiff (:y2 - :y1)
        if (xdiff < 0)
            [setxdiff xdiff - (2 * xdiff)]
        if (ydiff < 0)
            [setydiff ydiff - (2 * ydiff)]
        setdisp-n 0
        ifelse (xdiff > ydiff)
            [
                ifelse ((:x2 - :x1) > 0)
                    [
                        ifelse (:y2 > :y1)
                            [
                                repeat xdiff + 1
                                    [display-set-pixel
                                     (:x1 + disp-n)
                                     (:y1 + (disp-n *
                                       ydiff / xdiff))
                                     setdisp-n
                                       disp-n + 1]
                            ]
                            [
                                repeat xdiff + 1
                                    [display-set-pixel
                                     (:x1 + disp-n)
```

```
                                    (:y1 - (disp-n *
                                      ydiff / xdiff))
                                    setdisp-n
                                      disp-n + 1]
                            ]
                        ]
                        [
                            ifelse (:y2 > :y1)
                                [
                                    repeat xdiff + 1
                                        [display-set-pixel
                                         (:x1 - disp-n)
                                         (:y1 + (disp-n *
                                           ydiff / xdiff))
                                         setdisp-n
                                           disp-n + 1]
                                ]
                                [
                                    repeat xdiff + 1
                                        [display-set-pixel
                                         (:x1 - disp-n)
                                         (:y1 - (disp-n *
                                           ydiff / xdiff))
                                         setdisp-n
                                           disp-n + 1]
                                ]
                        ]
                ]
                [
                    ifelse ((:y2 - :y1) > 0)
                        [
                            ifelse (:x2 > :x1)
                                [
                                    repeat ydiff + 1
                                        [display-set-pixel
                                         (:x1 + (disp-n *
                                           xdiff / ydiff))
                                         (:y1 + disp-n)
                                         setdisp-n
                                           disp-n + 1]
                                ]
                                [
                                    repeat ydiff + 1
                                        [display-set-pixel
                                         (:x1 - (disp-n *
                                           xdiff / ydiff))
```

```
                                    (:y1 + disp-n)
                                    setdisp-n
                                      disp-n + 1]
                            ]
                        ]
                        [
                            ifelse (:x2 > :x1)
                                [
                                    repeat ydiff + 1
                                        [display-set-pixel
                                         (:x1 + (disp-n *
                                           xdiff / ydiff))
                                         (:y1 - disp-n)
                                         setdisp-n
                                           disp-n + 1]
                                ]
                                [
                                    repeat ydiff + 1
                                        [display-set-pixel
                                         (:x1 - (disp-n *
                                           xdiff / ydiff))
                                         (:y1 - disp-n)
                                         setdisp-n
                                           disp-n + 1]
                                ]
                        ]
                ]
        end
```

Examples of Use

To use the Display layer, we probably want to start by clearing it like this:

```
display-clear
```

In most cases, users will not directly be using the write-byte and read-byte procedures, so let's skip ahead to the fun stuff and make things light up. We'll start by just turning on a pixel in about the middle of the screen:

```
display-set-pixel 63 31
```

How about we now turn on a random pixel on the screen, just for kicks.

```
display-set-pixel (random % 128) (random % 64)
```

To see something really cool, let's write a loop that just keeps setting and clearing random pixels:

```
loop [display-set-pixel (random % 128)
                        (random % 64)
      display-clear-pixel (random % 128)
                          (random % 64)]
```

Now, we need to stop this program from running before we can talk to the Foundation again. Just pushing the button won't do it though, because if that stops the program in the middle of an I2C communication, the layer will be in a bad state. The best way to make sure everybody's happy is to turn the power off and then back on.

Now, why don't we go ahead and print some text in the middle of the screen:

```
display-print-string 3 "|LCD Displays are cool|
```

Let's say "Hello" on another line:

```
display-print-string 1 "|Hello|
```

Since the line pointer for line 1 now points to the next blank space after the word "Hello", the next thing that we print on that line will appear right after it. Let's print a number:

```
display-print-num 1 123
```

The line should now read "Hello123". Let's say that we want it to say "Hello456" instead. We could always clear the line and start over, but we can also move the line index back to the point we want the new number to be printed at. If you start counting at 0, the number "4" needs to be printed at position 5. We'll set the index like this:

```
display-set-line-index 1 5
```

Now, if we print the number 456 like this:

```
display-print-num 1 456
```

The line should read "Hello456". Let's now clear the line:

```
display-clear-line
```

Finally, let's draw two lines on the screen, forming a big "X" from corner to corner:

```
display-draw-line 0 0 127 63
display-draw-line 127 0 0 63
```

With these basic text and drawing primitives, it's possible to write many complex graphics routines capable of displaying anything the user desires.

Appendix O: Cricket Bus Layer Documentation

Figure O.1 - The Cricket Bus layer.

Description

The Cricket Bus layer *(Figure O.1)* has eight bus-ports, and is capable of communicating via a proprietary one-wire protocol with any device in the Media Lab's Cricket System, as well as with Crickets themselves.

Hardware Detail

The Cricket Bus layer uses a PIC to generate the tightly timed one-wire bus signals. The connector is a 3-pin JST connector, as found on all devices in the Cricket system. The pin configuration of the port is shown below *(Figure O.2)*:

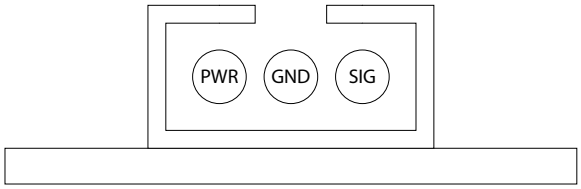


Figure O.2 - A bus port connector.

The protocol is master-driven, and devices on the bus will only respond when directly spoken to. The bus protocol itself consists of eight, 10-microsecond long bits, preceded by a start bit and a 100-microsecond synchronization time to give all devices on the bus a chance to respond if they are in the middle of an operation. A diagram of the Cricket bus protocol is shown below *(Figure O.3)*:

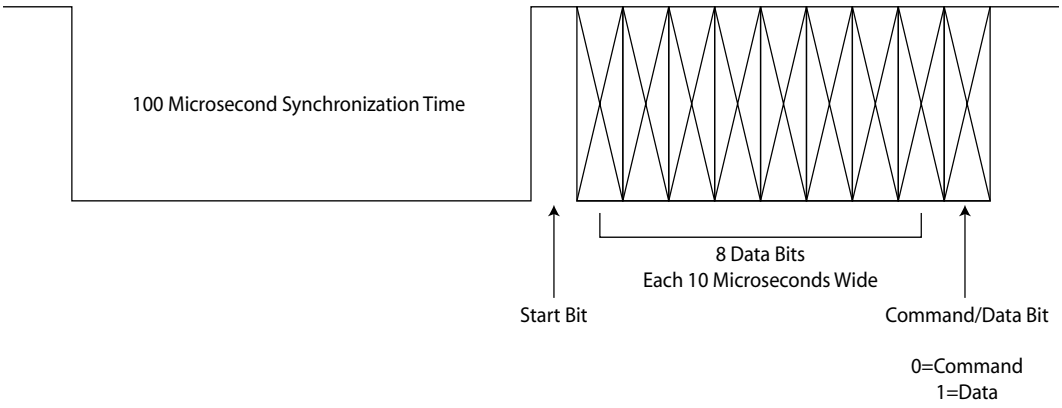


Figure O.3 - The Cricket bus protocol.

It should be noted that all eight bus-ports on the layer are tied together. The presence of multiple ports is just to allow for ease of wiring. Since some bus devices consume a great deal of power, it is possible to power the Bus lines off of either the primary or secondary power bus, as selected by the switch on the layer.

Layer Code

The include file for the Cricket Bus layer contains a variety of functions for talking to some of the more commonly used Cricket bus devices. Currently, functions have been written for talking to the Hexadecimal Display, Optical Distance Sensor, Clap Sensor, and the Voice Recorder. Additionally, there is a function that allows users to set a delay time between bytes being sent out over the bus, since some bus devices are unable to keep up if data is sent at full speed. Even though functions have only been written for a handful of bus devices, upon inspection you can see how easy it is to write functions to talk to any device in the Cricket system should the need arise. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **hex-display** function is used to display a value on a Hexidecimal Display bus device. It takes one argument, the number to display, which can be a value from 0 to 9999. Within the function, a total of four values are actually sent to the Cricket Bus layer. First, a "0" is sent, indicating that we wish to talk to a bus device, but are not expecting a response back from it. After that, we send the address of the bus device, in this case $70. Then, we send the value to display, split into high and low bytes, since the communication is only eight bits at a time.

```
to hex-display :n
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 4
    i2c-write-byte 0
    i2c-write-byte $70
    i2c-write-byte :n / 256
    i2c-write-byte :n % 256
    i2c-stop
end
```

The **distance** function is used to take a reading from an Optical Distance Sensor bus device. It takes no, arguments, but sends a total of two values to the layer. First a "1" is sent, indicating that we want to talk to a bus device, and that we do expect a response back from it. Then the address is sent, which for the Optical Distance Sensor is $55. After the request is sent to perform the sensor reading, the result is read out of the layer's transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to distance
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 2
    i2c-write-byte 1
    i2c-write-byte $55
    i2c-stop
    i2c-start
    i2c-write-byte $03
    waituntil [255 > i2c-read-byte 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **clap** function is used to take a reading from an Optical Distance Sensor bus device. It takes no, arguments, but sends a total of three values to the layer. First a "1" is sent, indicating that we want to talk to a bus device, and that we do expect a response back from it. Then the address is sent, which for the Clap Sensor is $6a. Next, we send a $01, because that code is needed to ask the device if it's heard a clap (or other loud noise), since the last time we asked it. After the request is sent to perform the sensor reading, the result is read out of the layer's transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to clap
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 3
    i2c-write-byte 1
    i2c-write-byte $6a
    i2c-write-byte $01
    i2c-stop
    i2c-start
    i2c-write-byte $03
    waituntil [255 > i2c-read-byte 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **voicerec-play** function is used to play a voice sample that has been previously recorded on the Voice Recorder bus device. It takes one argument, the sample number to play, which can be a value from 0 to 255. Within the function, a total of three values are actually sent to the Cricket Bus layer.

First, a "0" is sent, indicating that we wish to talk to a bus device, but are not expecting a response back from it. After that, we send the address of the bus device, in this case $54. Then, we send the number of the sample to play, and playback should begin immediately.

```
to voicerec-play :n
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 3
    i2c-write-byte 0
    i2c-write-byte $54
    i2c-write-byte :n
    i2c-stop
end
```

The **voicerec-clear** function is used to erase all of the voice samples that have been previously recorded on the Voice Recorder bus device. It takes no arguments, and within the function, a total of three values are actually sent to the Cricket Bus layer. First, a "0" is sent, indicating that we wish to talk to a bus device, but are not expecting a response back from it. After that, we send the address of the bus device, in this case $54. Then, we send a 255, which tells the bus device itself to reset its memory. The green LED on the Voice Recorder will flash to indicate that the erase command was successful. It is important to note that it is necessary to power-cycle the Voice Recorder itself after this command is sent, or the memory change will not actually take effect.

```
to voicerec-clear
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 3
    i2c-write-byte 0
    i2c-write-byte $54
    i2c-write-byte 255
    i2c-stop
end
```

The **set-bus-delay** function is used to increase the delay between bytes sent out over the bus signal line. While not needed for any of the devices that functions have currently been written for, there are some devices in the system which cannot take data as quickly, and need to have their delay increased. By default, the delay between bytes is about 10 microseconds. This function takes a single argument, the desired time delay (in milliseconds), which can be a value from 0 to 255. Two bytes are sent to the actual layer, a "2" indicating that the delay time is being changed, immediately followed the new desired timing delay.

```
to set-bus-delay :delay
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 2
    i2c-write-byte 2
    i2c-write-byte :delay
    i2c-stop
end
```

Examples of Use

To use the Cricket Bus layer, lets say that we've got all of the above-mentioned bus devices plugged into it. While that probably won't be true for the vast majority of the people using it, we'll look at examples of how to talk to each different device.

Talking to the Hexadecimal Display device is as simple as saying:

```
hex-display 1234
```

The number 1234 should now appear on the display itself. Let's now use the display to display a reading from the Optical Distance Sensor:

```
hex-display distance
```

You should see a single value appear on the display. That probably wasn't all that interesting, since it only happened once. Let's make it loop:

```
loop [hex-display distance wait 10]
```

Now, the Hexadecimal Display should be updating every second with a new value from the Optical Distance Sensor. While the wait statement in the loop is not necessary, it's in there so that we have a chance to actually read the number before it changes too quickly.

Now let's try using the Clap Sensor. We can use it just like we used the Optical Distance Sensor, and display its output on the Hexadecimal Display device like this:

```
loop [hex-display clap wait 10]
```

Like before, the display will update every second. In this case however, it will always be a "0" or a "1". A value of "0" means that it hasn't heard anything in the last second, and a "1" means that it has.

Let's add some sound to our project. For the sake of this example, assume that we have two voice samples recorded onto the Voice Recorder device. To record samples, you hold down the "Rec" button on the device itself and speak into the microphone. Let's say that the first sample is "Hello", and the second is "Goodbye". Here's how we tell the board to say "Hello":

```
voicerec-play 0
```

Now, let's do something a little more complicated. Let's check the distance sensor every 5 seconds, and if someone is close, we'll say "Hello", and if nobody is around, we'll say "Goodbye."

```
loop [if (distance > 100)
      [voicerec-play 0][voicerec-play 1] wait 50]
```

With the Optical Distance Sensor, higher numbers mean that the object is closer. In this case, we set the threshold on the Optical Distance Sensor to be 100, but that is something that should usually be determined experimentally given the particular application.

Finally, just to show how to use it, let's change the delay between bytes being sent on the bus signal line:

```
set-bus-delay 10
```

There should now be a 10 ms delay between each byte sent over the bus. Everything that worked before should still work, it just might take a little longer to happen.

Appendix P: I²C Layer Documentation

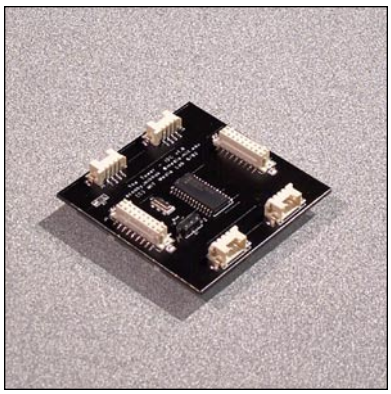


Figure P.1 - The DC Motor layer.

Description

The I²C layer *(Figure P.1)* has four ports to which any off-board Tower layer can be connected via a 4-wire I²C cable. The clock and data pins are connected directly to the Tower's main I²C bus, so boards connected through cables will appear to be directly on the Tower-stack when it is scanned. These layers are also useful for large-scale systems, because cables can be run at lengths up to 60 feet with no signal loss.

Hardware Detail

The I²C layer uses 4-pin Hirose surface mount connectors, providing power, ground, clock and data signals. A pin diagram of the connector is shown below *(Figure P.2)*:

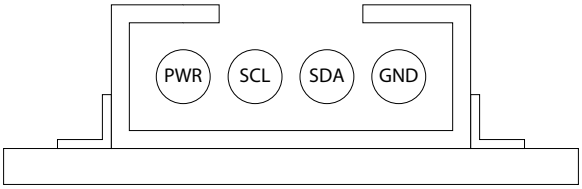


Figure P.2 - An I²C port connector.

Power to off-stack I²C devices can be tapped from the primary or secondary bus, as selected by the switch located on the board.

While not actually needed for operation, there is a PIC processor on this layer. The PIC is present for the sole purpose of letting the foundation know when this layer is on the stack. While this may seem unnecessary, it builds upon our goal of a fully modular system, and gives the user program full knowledge and control over everything connected to the Tower.

Layer Code

Since this board provides no functionality on its own, there are no procedures available that communicate directly with the board.

Examples of Use

Being primarily a connectivity point, there are no examples of how to use this board on its own. For using devices connected to it, please refer to their respective documentation files.

Appendix Q: Tricolor Layer Documentation

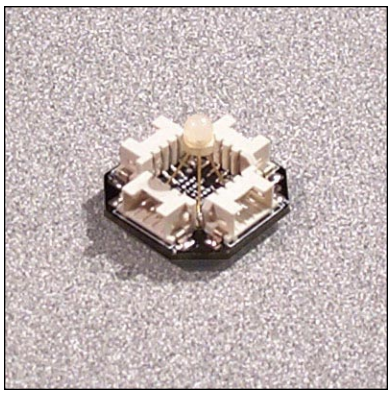


Figure Q.1 - The Tricolor layer.

Description

The Tricolor layer *(Figure Q.1)* contains a full-spectrum red-green-blue LED, capable of displaying millions of colors. Individual color change and fade commands can be sent to the board through the main Tower bus when connected through the I2C layer.

Hardware Detail

The Tricolor layer is an off-board Tower layer. It does not stack on top of the Tower as other layers do, but rather connects via bus cables to the I2C breakout layer. The cables are four wires, and each tricolor layer has four sockets to allow for the simple creation of complex spatial topologies. The Ledtronics DIS-1024 LED itself has integrated red, green, and blue diodes all inside of it, to allow for the easiest color mixing. One important thing to note about this layer, is that it has a button located underneath it. The button allows the easy creation of complex topologies, by making is simple to dynamically assign addresses as boards are connected. While most Tower layers can have their I2C addresses changed in software, the Tricolor layer can have its address reset to the default value by holding down the button when the board is plugged in or power is turned on.

Layer Code

The include file for the Tricolor layer contains one function for fading the LED to a desired color value. *(This function is written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **tricolor** function takes five arguments, the address of the tricolor board to talk to, the individual red, green, and blue values from 0 to 255, and the fade time to transition to that color, in hundredths of a second, also ranging from 0 to 255. The values are all sent, in order to the layer itself, preceded by a "0" to indicate that a color fade is being performed.

```
to tricolor :addr :r :g :b :time
    i2c-start
    i2c-write-byte :addr
    i2c-write-byte 5
    i2c-write-byte 0
    i2c-write-byte :r
    i2c-write-byte :g
    i2c-write-byte :b
    i2c-write-byte :time
    i2c-stop
end
```

Examples of Use

Using the Tricolor layer is as simple as just calling the tricolor function with the desired color value. For now, let's say that we want to make the LED on address 20 fade to purple, and take 1 second to get there. We can just type:

```
tricolor 20 255 0 255 100
```

In RGB values, purple is 255, 0, 255, meaning that both red and blue are at full, and green is at zero. Since the time argument is in hundredths of a second, and we wanted a 1 second fade time, we sent a value of 100 for that argument.

There really aren't any more complicated things to do with the Tricolor layer, but just for fun, let's have it fade to random color every two seconds. The code to do that would look like this:

```
loop [tricolor 20
           (random % 256)
           (random % 256)
           (random % 256)
           200
      wait 20]
```

In this loop, the tricolor function is called with three random color values, each a random value modulo 256, to give the desired 0 to 255 range. Since we want a new color every two seconds, we send a 200 as the fade time argument. To ensure that new fade commands are only sent every two seconds, we wait for two seconds at the end of each pass through the loop.

Appendix R: Proto Layer Documentation

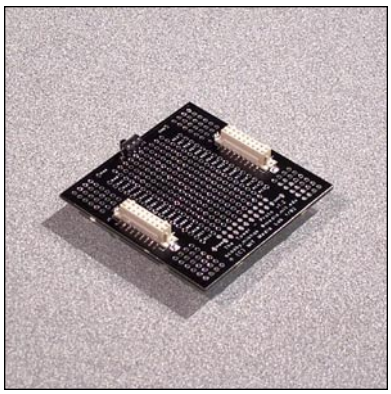


Figure R.1 - The Proto layer.

Description

The Proto layer *(Figure R.1)* is designed to give users the ability to wire up their own electronics into the Tower system. Essentially a blank perforated board with Tower connectors and tap points for every I/O pin, this layer can be used to quickly prototype new circuits and to connect external electronics to any of the available pins on the foundation.

Hardware Detail

The Proto layer has easy-to-access tap-points for all 33 I/O pins passed up from the PIC Foundation. There are two power and ground connection points, and common distribution busses. The PWR points can be switched between the primary and secondary power busses by the switch at the top of the board. A diagram of the board is shown below *(Figure R.2)*:

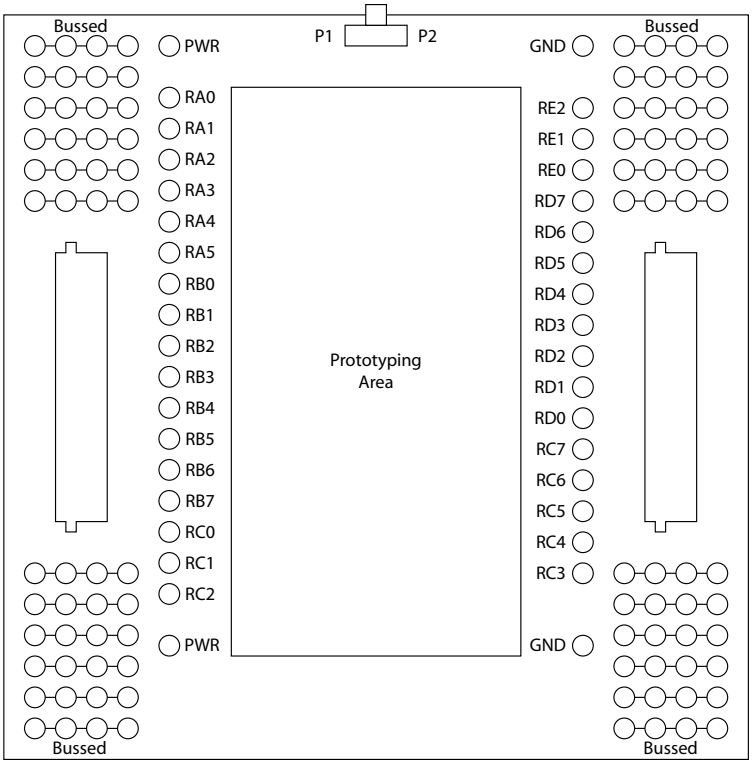


Figure R.2 - A diagram of the Proto layer.

Layer Code

Since the Proto layer can be built up and configured in any way the users desires, there is no actually pre-existing layer code for it. All communications with anything built on the layer are done directly with the processor on the foundation.

Examples of Use

The Proto Layer has no inherent functionality, so there are no specific examples of how to communicate with it. For general I/O communication, please refer to the documentation for the Foundation that you are using.

Appendix S: PICProto Layer Documentation

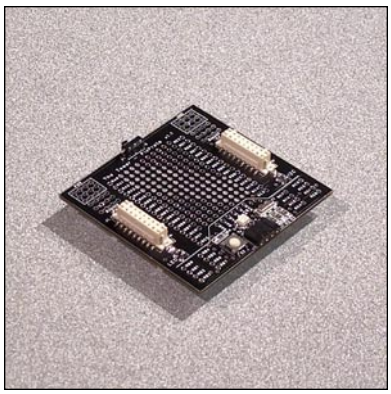


Figure S.1 - The PICProto layer.

Description

The PicProto layer *(Figure S.1)* is designed to give users the ability to make their own layers for the Tower system. In addition to the blank perforated board and tap-points for every I/O pin on the Foundation, this layer has a Logochip on-board, which is another PIC Processor running the same virtual machine as the foundation. With its I/O pins accessible as well, and a programming header on-board, anyone can easily build new layers that add functionality to the Tower system.

Hardware Detail

The PICProto layer has easy-to-access tap-points for all 33 I/O pins passed up from the foundation and all 22 I/O pins from the on-board LogoChip. There are power and ground connection blocks, and common distribution busses. The PWR points can be switched between the primary and secondary power busses by the switch at the top of the board. The I^2C connection between the LogoChip and the foundation can be jumpered if desired, as can the bicolor LED also on-board. A diagram of the board is shown below *(Figure S.2)*:

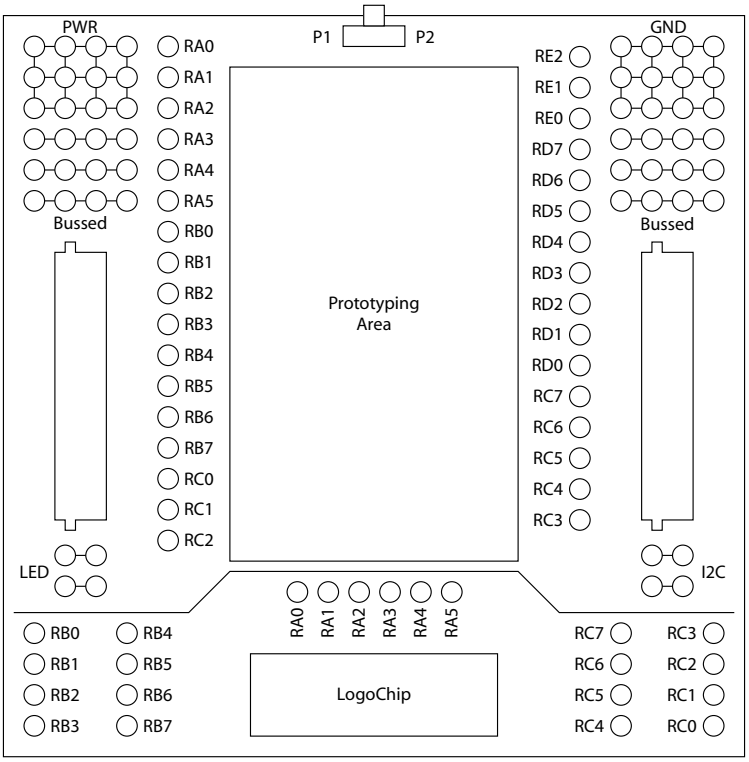


Figure S.2 - A diagram of the PICProto layer.

The LogoChip can be programmed through the on-board serial-programming header when connected to a serial cable with an RS-232 module on the end. The pin configuration of the header on the board is shown below *(Figure S.3)*:

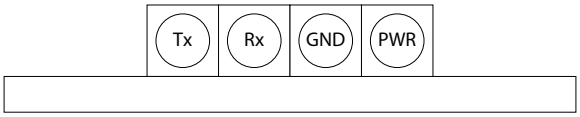


Figure S.3 - The PICProto serial port connector.

In addition to downloading Logo code, it is also possible to download assembly code. To download assembly, the board must be powered on while the button is depressed. At that point, assembly code can be downloaded through the Tower Development Environment. After downloading new assembly code, it is necessary to power-cycle the layer before the new code will run.

Layer Code

Since the PicProto layer can be built up and configured in any way the users desires, there is no actually pre-existing layer code for it. All communications with anything built on the layer are done directly with the processor on the foundation.

In order to use the I^2C protocol to communicate with the foundation, there are four new important functions that can be called from a Logo program. One allows you to check for new data, one gets data that has been received in the buffer, one puts data into the buffer, and the last one sets a flag saying that the buffer has been filled and is ready to be sent back.

It's important to first understand a bit about how the I^2C protocol is working on the Tower layers. When the foundation sends out information over the bus, every layer checks to see if its address matches the one the foundation wants to talk to.

If it matches, the next byte that comes from the foundation tells the layer how many bytes it's going to be sending. The layer keeps grabbing bytes and putting them in the receive buffer until it's gotten the number that it was told it was receiving. All of that work is done in the background, but as soon as it's completed, a flag inside the chip is set saying that new data is present. The flag can be checked using the **new-i2c?** primitive like this:

```
print new-i2c?
> 1
```

When that function returns a value of "1", we can go ahead and start looking at the data that came in. To grab a byte out of the receive buffer, we use the **get-i2c** primitive. The primitive takes a single argument, which is the buffer location to get the data out of. Location 0 will have the number of arguments

that were sent, not including itself, and the actual arguments will begin in location 1. For example, if we sent 3 arguments to the layer, location 0 would have a "3" in it, and the actual data would be in locations 1, 2, and 3. Let's see what the first argument is:

```
print get-i2c 1
> 14
```

It's that simple to get data from the foundation. Once we have the data, we probably want to do something based on what arguments were sent. Often, we'll want to send data back to the foundation.

To return values to the foundation, all of the layers have a separate transmit buffer. The transmit buffer operates almost identically to the receive buffer, in that its first location contains the number of values being sent back, followed by the values itself. To put data in the transmit buffer, we use the **put-i2c** primitive. The primitive takes two arguments, the buffer location, and the value to put there. Let's say we want to send back the number 23 to the Foundation. To do so, we can write:

```
put-i2c 1 23
```

We put the value in the first location of the buffer, but there's something we forgot. We need to tell the foundation how many data values it's going to be getting. In this case, we're only sending a single byte back, so we should store a "1" in location 0 of the transmit buffer, like this:

```
put-i2c 0 1
```

But there's still one more important step. The I²C protocol is master-driven, meaning that that layers don't send back data until the foundation specifically asks for it. There could be a potential problem if the foundation tried to grab data when the transmit buffer was only half-filled. To solve that problem, there's another flag inside the chip, which tells the foundation whether or not it's ready to send back data.

The layer will just tell the foundation that it's not ready, and to keep asking until the data is actually complete. After we've filled the buffer, we need to set that flag using the **ready-i2c** primitive. No arguments are needed, and it can just be called as follows:

```
ready-i2c
```

Now the foundation is free to grab all of the data out of the buffer, and continue about its business. The flag will be automatically cleared as soon as the data has all been sent.To make a new layer for the

Tower using the PICProto layer, two pieces of code need to be written. There's one part that runs on the PICProto layer itself, and another that has to run on the foundation in order to talk to it.
For a simple example of communicating between layers and the foundation, let's program the layer to take two arguments, add them together, and send them back to the foundation. The code for the PICProto would look something like this:

```
on-startup [logochip-add]

to logochip-add
    loop
    [
        waituntil [new-i2c?]
        setn (get-i2c 1) + (get-i2c 2)
        put-i2c 1 n
        put-i2c 0 1
        ready-i2c
    ]
end
```

This function has a loop, sits at the beginning waiting for an I^2C communication. As soon as one is received, we're ready to add. We set the global variable "n" equal to the sum of the first two arguments. We then store "n" in the first location of the transmit buffer, and store a "1" in location 0, telling the chip that we're only sending back the one value. Finally, we set the ready-flag, and go back to the top of the loop to wait for another query.

The on-startup statement at the top just ensures that this code will start running when the PICProto layer is powered on. The layer will only respond to the foundation if the program is running. Remember, you can't talk to a LogoChip or foundation while it's running code, so you need to stop the program by pushing the white button before you can reprogram it with something new.

Now that we've got that half of the code taken care of, let's write the function that will run on the foundation. This will be pretty much identical to what you'd find in the include files for any of the other layers, since it's using the exact same method to communicate with the PicProto layer as it does with any other. The add function we want to write should resemble this:

```
to add :n1 :n2
    i2c-start
    i2c-write-byte $1e
    i2c-write-byte 2
    i2c-write-byte :n1
    i2c-write-byte :n2
    i2c-stop
    i2c-start
    i2c-write-byte $1f
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

This function opens communication to the PICProto layer, whose default address is $1e, and then sends our two arguments to be added. Remember, the "2" is just saying that two arguments are going to follow. After that, we need to open communication in the other direction by sending out the address $1f. This is where that ready-flag becomes important. That waituntil statement will sit there until it gets back a "1" from the layer, indicating that one byte is going to follow. Until the ready-flag has been set on the layer, i2c-read-byte will just keep receiving 255s. As soon as we know the data is ready, we grab it and store it, stop the I²C communication, and then output it to whatever function called this one. *(This function was written for the PIC foundation. On other foundations, the code would differ slightly, due to the presence of local variables.)* Let's try running the function from the foundation to see if it works:

```
print add 3 5
> 8
```

There is one very important note that we should mention. In most cases, you would never just sent the arguments to your function without a preceding byte for a dispatch routine. For example, we could make our first argument be either a "0" or a "1", where a "0" would indicate that it wanted to add the two numbers together, and a "1" would tell the layer to perform a subtraction instead. After that first byte, the two arguments themselves would be sent just as before. Even if a layer only performs one function, it's still a good idea to send a "0" or some other number first.

The reason why, is because the first argument carries special weight. If the first argument happens to be a 255, the layer enters addressing mode, in which the address of the board can be easily changed. While that's not a problem that can't be fixed, it can make things very frustrating to debug if your program keeps changing the address of the layer that it's trying to communicate with.

Appendix T: RS-232 Module Documentation

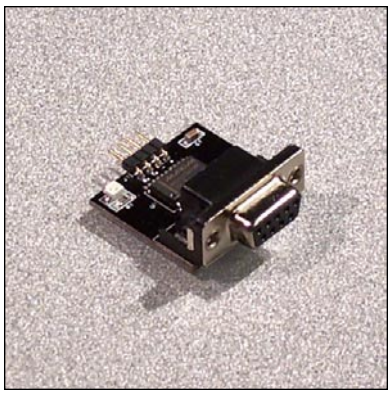


Figure T.1 - The RS-232 module.

Description

The RS-232 module *(Figure T.1)* is used to convert a line-level serial signal from a computer down to the logic-level signals needed to communicate with the Tower.

Hardware Detail

The RS-232 module uses a Maxim MAX233 to perform the actual level conversion, and has a female DB9 serial connector on-board. The side that connects to the Tower is a male 4-pin connector with the following pin configuration *(Figure T.2)*:

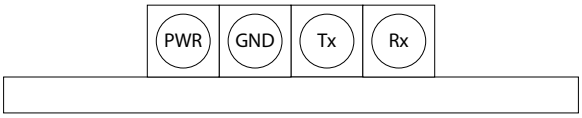


Figure T.2 - The RS-232 serial port connector.

The board also contains a bicolor LED, which will light up green when data is being transmitted, and red when data is being received.

Layer Code

Since this module is just a level-shifter and contains no processor, there is no code to actually communicate with it.

Examples of Use

Being just a communications adapter, there are no examples of how to use this board on its own. For using devices connected to it, please refer to their respective documentation files.

Appendix U: Application Code - ScoobyScope

```
; ********************************************************************
; ScoobyScope - A program for plotting sensor data in real-time.
;
; Chris Lyon, scooby@mit.edu
;
;     Needed:
;             PIC Foundation
;             Sensor Layer
;             Display Layer
;     Attach:
;             A sensor to port 1 on the Sensor layer.
; ********************************************************************

include pic/include/standard.inc
include pic/include/sensor.inc
include pic/include/display.inc

on-startup [scope]
on-white-button [scope]

globals [x y oldx oldy]

to scope
    display-print-string 7 "|ScoobyScope v1.1|
    display-draw-line 0 53 127 53
    setx 0
    setoldx 0
    setoldy 50
    loop
    [
        sety 0
        repeat 51 [display-clear-pixel x y sety y + 1]
        sety (50 - ((sensor 1) / 82))
        display-draw-line oldx oldy x y
        setoldx x
        setoldy y
        setx x + 1
        if (x = 128) [setx 0 setoldx 0]
    ]
    wait 1
end
```

Appendix V: Application Code - 2D Plotter

```
; ************************************************************************
; 2D Plotter - A program to control a 2D plotting device to draw
;              pictures with a computer-controlled marker.
;
; Chris Lyon, scooby@mit.edu
;
;     Needed:
;             PIC Foundation
;             DC Motor Layer
;             Servo Motor Layer
;             Display Layer
;     Attach:
;             Four DC motors as follows:
;                 Motor 1: X-Axis Motor
;                 Motor 2: X-Axis Motor (Structurally reversed)
;                 Motor 3: Y-Axis Motor
;                 Motor 4: Y-Axis Motor (Structurally reversed)
;             A servo motor to port 1 on the Servo Motor layer
;                 (For pen up/down control)
; ************************************************************************

include pic/include/standard.inc
include pic/include/print.inc
include pic/include/address.inc
include pic/include/motor.inc
include pic/include/servo.inc
include pic/include/display.inc

globals [plotter-xdiff plotter-ydiff current-x current-y
         serial-byte1 serial-byte2 serial-byte3 serial-byte4]

on-startup [mainloop]
on-white-button [mainloop]

to mainloop
    init
    loop [process-packet]
end

to process-packet
    waituntil [new-serial?]
    if (not (get-serial = 255)) [stop]
    waituntil [new-serial?]
    setserial-byte1 get-serial
```

```
    waituntil [new-serial?]
    setserial-byte2 get-serial
    waituntil [new-serial?]
    setserial-byte3 get-serial
    waituntil [new-serial?]
    setserial-byte4 get-serial
    waituntil [new-serial?]
    if (not (get-serial = 0)) [stop]
    if (serial-byte1 > 200) [setserial-byte1 200]
    if (serial-byte2 > 200) [setserial-byte2 200]
    if (serial-byte3 > 200) [setserial-byte3 200]
    if (serial-byte4 > 200) [setserial-byte4 200]
    draw-line serial-byte1 serial-byte2 serial-byte3 serial-byte4
end

to draw-line :x1 :y1 :x2 :y2
    display-clear-line 2
    display-clear-line 4
    display-clear-line 5
    display-clear-line 6
    display-clear-line 7
    display-print-string 2 "|Drawling line...|
    display-print-string 4 "|    Start X: |
    display-print-string 5 "|    Start Y: |
    display-print-string 6 "|      End X: |
    display-print-string 7 "|      End Y: |
    display-print-num 4 :x1
    display-print-num 5 :y1
    display-print-num 6 :x2
    display-print-num 7 :y2
    pen-up
    move-line current-x current-y :x1 :y1
    pen-down
    move-line :x1 :y1 :x2 :y2
    pen-up
    setcurrent-x :x2
    setcurrent-y :y2
    display-clear-line 2
    display-clear-line 4
    display-clear-line 5
    display-clear-line 6
    display-clear-line 7
    display-print-string 2 "|Waiting for data...|
end

to init
    display-clear
    display-print-string 0 "|2D Plotter|
```

```
    display-print-string 2 "|Waiting for data...|
    servo-on 1
    wait 1
    pen-down
    re-home
end

to re-home
    ; 100,100 is the center position
    ; 0,0 and 200,200 are the corners of the plotting area
    setcurrent-x 100
    setcurrent-y 100
end

to pen-up
    turn-servo 1 45
end

to pen-down
    turn-servo 1 0
end

to move-line :x1 :y1 :x2 :y2
    if (:x1 = :x2)
        [
            ifelse (:y2 > :y1)
            [
                repeat (:y2 - :y1) [motor-kick-pos-y]
                stop
            ]
            [
                repeat (:y1 - :y2) [motor-kick-neg-y]
                stop
            ]
        ]
    if (:y1 = :y2)
        [
            ifelse (:x2 > :x1)
            [
                repeat (:x2 - :x1) [motor-kick-pos-x]
                stop
            ]
            [
                repeat (:x1 - :x2) [motor-kick-neg-x]
                stop
            ]
        ]
    setplotter-xdiff (:x2 - :x1)
```

```
setplotter-ydiff (:y2 - :y1)
if (plotter-xdiff < 0)
    [setplotter-xdiff plotter-xdiff - (2 * plotter-xdiff)]
if (plotter-ydiff < 0)
    [setplotter-ydiff plotter-ydiff - (2 * plotter-ydiff)]
ifelse (plotter-ydiff > plotter-xdiff)
[
    ifelse ((:x2 - :x1) > 0)
        [
            ifelse (:y2 > :y1)
                [repeat plotter-xdiff
                        [motor-kick-pos-x
                         repeat (plotter-ydiff / plotter-xdiff)
                                [motor-kick-pos-y]]]
                [repeat plotter-xdiff
                        [motor-kick-pos-x
                         repeat (plotter-ydiff / plotter-xdiff)
                                [motor-kick-neg-y]]]
        ]
        [
            ifelse (:y2 > :y1)
                [repeat plotter-xdiff
                        [motor-kick-neg-x
                         repeat (plotter-ydiff / plotter-xdiff)
                                [motor-kick-pos-y]]]
                [repeat plotter-xdiff
                        [motor-kick-neg-x
                         repeat (plotter-ydiff / plotter-xdiff)
                                [motor-kick-neg-y]]]
        ]
]
[
    ifelse ((:y2 - :y1) > 0)
        [
            ifelse (:x2 > :x1)
                [repeat plotter-ydiff
                        [motor-kick-pos-y
                         repeat (plotter-xdiff / plotter-ydiff)
                                [motor-kick-pos-x]]]
                [repeat plotter-ydiff
                        [motor-kick-pos-y
                         repeat (plotter-xdiff / plotter-ydiff)
                                [motor-kick-neg-x]]]
        ]
        [
            ifelse (:x2 > :x1)
                [repeat plotter-ydiff
                        [motor-kick-neg-y
```

```
                            repeat (plotter-xdiff / plotter-ydiff)
                                   [motor-kick-pos-x]]]
                    [repeat plotter-ydiff
                            [motor-kick-neg-y
                             repeat (plotter-xdiff / plotter-ydiff)
                                   [motor-kick-neg-x]]]
            ]
    ]
end

to motor-kick-pos-x
    motor-on-forward 1 255 0
    motor-on-reverse 2 255 0
    mwait 25
    motor-stop 1
    motor-stop 2
    mwait 25
end

to motor-kick-neg-x
    motor-on-reverse 1 255 0
    motor-on-forward 2 255 0
    mwait 25
    motor-stop 1
    motor-stop 2
    mwait 25
end

to motor-kick-pos-y
    motor-on-forward 3 255 0
    motor-on-reverse 4 255 0
    mwait 25
    motor-stop 3
    motor-stop 4
    mwait 25
end

to motor-kick-neg-y
    motor-on-reverse 3 255 0
    motor-on-forward 4 255 0
    mwait 25
    motor-stop 3
    motor-stop 4
    mwait 25
end
```

Appendix W: Application Code - WeatherStation

```
; ********************************************************************
; WeatherStation Transmitter- A program for taking time-stamped sensor
;                             data and transmitting it wirelessly to
;                             an indoor base station.
;
; Chris Lyon, scooby@mit.edu
;
;     Needed:
;             PIC Foundation
;             Sensor Layer
;             Clock Layer
;             IR Layer
;     Attach:
;             A sensor to port 1 on the Sensor layer.
; ********************************************************************

include pic/include/standard.inc
include pic/include/sensor.inc
include pic/include/clock.inc
include pic/include/ir.inc

on-startup [send-weather]
on-white-button [send-weather]

to send-weather
      loop
      [
            put-ir 255
            mwait 10
            put-ir clock-read 2 ; hour
            mwait 10
            put-ir clock-read 1 ; min
            mwait 10
            put-ir clock-read 0 ; sec
            mwait 10
            put-ir (sensor 1) / 16
            mwait 10
            put-ir (sensor 2) / 16
            mwait 10
            put-ir 0
            wait 10
      ]
end
```

```
; ************************************************************************
; WeatherStation Receiver- A program for receiving time-stamped sensor
;                          data wirelessly, plotting it on a display,
;                          and sending it to a computer for analysis.
;
; Chris Lyon, scooby@mit.edu, 1/03
;
;     Needed:
;            PIC Foundation
;            IR Layer
;            Display Layer
;            RS-232 Module
;     Attach:
;            No external connections needed
; ************************************************************************

include pic/include/standard.inc
include pic/include/print.inc
include pic/include/ir.inc
include pic/include/display.inc

globals [ir-byte1 ir-byte2 ir-byte3 ir-byte4 ir-byte5]

on-startup [get-weather]
on-white-button [get-weather]

to get-weather
    display-clear
    display-print-string 0 "|ScoobyStation v1.0|
    display-print-string 1 "---------------------
    display-print-string 3 "|Time:|
    display-print-string 5 "|Sensor 1:|
    display-print-string 6 "|Sensor 2:|
    loop [process-packet]
end

to process-packet
    waituntil [new-ir?]
    if (not (get-ir = 255)) [stop]
    waituntil [new-ir?]
    setir-byte1 get-ir
    waituntil [new-ir?]
    setir-byte2 get-ir
    waituntil [new-ir?]
    setir-byte3 get-ir
    waituntil [new-ir?]
    setir-byte4 get-ir
    waituntil [new-ir?]
```

```
    setir-byte5 get-ir
    waituntil [new-ir?]
    if (not (get-ir = 0)) [stop]
    display-set-line-index 3 6
    display-set-line-index 5 10
    display-set-line-index 6 10
    display-print-string 5 "|   |
    display-print-string 6 "|   |
    display-set-line-index 5 10
    display-set-line-index 6 10
    if (ir-byte1 < 10)
    [
        display-print-num 3 0
    ]
    display-print-num 3 ir-byte1
    display-print-string 3 ":
    if (ir-byte2 < 10)
    [
    display-print-num 3 0
    ]
    display-print-num 3 ir-byte2
    display-print-string 3 ":
    if (ir-byte3 < 10)
    [
        display-print-num 3 0
    ]
    display-print-num 3 ir-byte3
    display-print-num 5 ir-byte4
    display-print-num 6 ir-byte5
    print ir-byte5
end
```

Appendix X: Application Code - ALF

```
; ********************************************************************
; ALF- A program for controlling an animatronic head by means of serial
;      communication with a host computer.
;
; Chris Lyon, scooby@mit.edu
;
;     Needed:
;             PIC Foundation
;             Servo Motor Layer
;     Attach:
;             Six servos as follows:
;                 Servo 1: Neck
;                 Servo 2: Mouth
;                 Servo 3: Eyes Vertical
;                 Servo 4: Eyes Horizontal
;                 Servo 5: Ears
;                 Servo 6: Eyebrows
; ********************************************************************

include pic/include/standard.inc
include pic/include/servo.inc

on-white-button [do-stuff]
on-startup [alf-wrapper]

to alf-wrapper
    alf-init
    setbaud-9600
    loop [alf-loop]
end

to alf-loop
    waituntil [new-serial?]
    if (not (get-serial = 101))
        [stop]
    waituntil [new-serial?]
    setn get-serial
    waituntil [new-serial?]
    setnn get-serial
    if (n = 0)
    [
        put-serial 1
        stop
    ]
```

```
    if (n = 1)
    [
        turn-head nn
        stop
    ]
    if (n = 2)
    [
        move-mouth nn
        stop
    ]
    if (n = 3)
    [
        eyes-updown nn
        stop
    ]
    if (n = 4)
    [
        eyes-sideways nn
        stop
    ]
    if (n = 5)
    [
        wiggle-ears nn
        stop
    ]
    if (n = 6)
    [
        move-eyebrows nn
        stop
    ]
end

to do-stuff
    alf-init
    loop
    [
        mouth-open
        wait 200
        ears-back
        eyes-down
        wait 100
        repeat 3 [eyebrows-up wait 2 eyebrows-down wait 2]
        wait 150
        eyes-right
        wait 200
        eyes-left
        wait 100
        eyes-center
```

```
        wait 350
        ears-forward
        mouth-close
        wait 150
        turnhead-left
        wait 200
        eyebrows-up
        wait 100
        mouth-open
        wait 250
        mouth-close
        turnhead-right
        wait 200
        mouth-open
        repeat 2 [ears-back wait 2 ears-forward wait 2]
        wait 250
        turnhead-center
        wait 200
    ]
end

to alf-init
    turnhead-center
    mouth-close
    eyes-center
    ears-forward
    eyebrows-down
    servo-on 1
    wait 2
    servo-on 2
    wait 2
    servo-on 3
    wait 2
    servo-on 4
    wait 2
    servo-on 5
    wait 2
    servo-on 6
    wait 2
end

to turnhead-left
    turn-head 0
end

to turnhead-right
    turn-head 100
end
```

```
to turnhead-center
    turn-head 50
end

to turn-head :n
    turn-servo 1 (((:n % 101) * 30 / 100))
end

to mouth-open
    move-mouth 0
end

to mouth-close
    move-mouth 100
end

to move-mouth :n
    turn-servo 2 ((:n % 101) * 30 / 100)
end

to eyes-down
    eyes-updown 100
end

to eyes-up
    eyes-updown 0
end

to eyes-updown :n
    turn-servo 3 ((:n % 101) * 40 / 100)
end

to eyes-left
    eyes-sideways 100
end

to eyes-right
    eyes-sideways 0
end

to eyes-sideways :n
    turn-servo 4 ((:n % 101) * 70 / 100)
end

to eyes-center
    eyes-sideways 50
    eyes-updown 50
end
```

```
to ears-forward
    wiggle-ears 100
end

to ears-back
    wiggle-ears 0
end

to wiggle-ears :n
    turn-servo 5 ((:n % 101) * 35 / 100)
end

to eyebrows-up
    move-eyebrows 100
end

to eyebrows-down
    move-eyebrows 0
end

to move-eyebrows :n
    turn-servo 6 ((:n % 101) * 40 / 100)
end
```

Appendix Y: Application Code - ScoobySnake

```
; ********************************************************************
; ScoobySnake- A Tower-based version of the videogame "Snake".
;
; Chris Lyon, scooby@mit.edu
;
;     Needed:
;             PIC Foundation
;             Sensor Layer
;             EEPROM Layer
;             Clock Layer
;             Display Layer
;     Attach:
;             Input devices as follows:
;                 Sensor 1: Neck
;                 Sensor 5: Joystick-Down
;                 Sensor 6: Joystick-Right
;                 Sensor 7: Joystick-Up
;                 Sensor 8: Joystick-Left
; ********************************************************************

include pic/include/sensor.inc
include pic/include/eeprom.inc

globals [nnnn nnn nn i2c-byte n button-hit worm-dir worm-head-x
         worm-head-y worm-tail-x worm-tail-y tail-dir collision
         grow even-space gridx gridy gridlength game-on worm-delay
         hit-last-move score lives level level-done up-button-hit
         down-button-hit left-button-hit right-button-hit hs hs-day
         hs-month-char0 hs-month-char1 hs-month-char2 hs-name-char0
         hs-name-char1 hs-name-char2 hs-name-char3 hs-name-char4
         hs-name-char5 start-score disp-n disp-nn]

on-startup [nibbles]
on-white-button [nibbles]

to nibbles
    loop
    [
        print-title-screen
        setscore 0
        setlives 2
        setlevel 0
        setlevel-done 0
        play-until-die
```

```
        display-clear
        ifelse (score > (read-hs 6))
        [
            seths-name-char0 127
            seths-name-char1 127
            seths-name-char2 127
            seths-name-char3 127
            seths-name-char4 127
            seths-name-char5 127
            display-print-string 1 "|You Set A High Score!|
            display-print-string 3 "|  -Enter Your Name-|
            setlives 0
            print-status-line
            set-high-score
            print-hs-table
            resett
            waituntil [((sensor 1) > 1000) or (timer > 5000)]
            waituntil [(sensor 1) < 1000]
        ]
        [
            display-print-string 2 "|      Game Over|
            mwait 50
            setlives 0
            print-status-line
            wait 20
        ]
    ]
end

to set-high-score
    get-name-char 0
    get-name-char 1
    get-name-char 2
    get-name-char 3
    get-name-char 4
    get-name-char 5
    seths-day clock-read 3
    get-month-name clock-read 4
    setn 1
    repeat 6
    [
        if (score > (read-hs n))
        [
            setnnnn 5
            repeat (6 - n)
            [
                move-score-down nnnn
                setnnnn nnnn - 1
```

```
            ]
            ee-write 0 (n * 12) hs-name-char0
            ee-write 0 (n * 12) + 1 hs-name-char1
            ee-write 0 (n * 12) + 2 hs-name-char2
            ee-write 0 (n * 12) + 3 hs-name-char3
            ee-write 0 (n * 12) + 4 hs-name-char4
            ee-write 0 (n * 12) + 5 hs-name-char5
            ee-write 0 (n * 12) + 6 (score % 256)
            ee-write 0 (n * 12) + 7 (score / 256)
            ee-write 0 (n * 12) + 8 hs-month-char0
            ee-write 0 (n * 12) + 9 hs-month-char1
            ee-write 0 (n * 12) + 10 hs-month-char2
            ee-write 0 (n * 12) + 11 hs-day
            stop
        ]
        setn n + 1
    ]
end

to move-score-down :n
    setnnn 0
    repeat 12
    [
        setnn ee-read 0 (:n * 12) + nnn
        ee-write 0 ((:n + 1) * 12) + nnn nn
        mwait 50
        setnnn nnn + 1
    ]
end

to print-title-screen
    setgame-on 0
    loop
    [
        if (game-on = 1)
        [
            stop
        ]
        display-clear
        display-print-string 0 "|   ScoobySnake v1.1|
        display-print-string 2 "|    -Press Button-|
        display-print-string 3 "|      -To Start-|
        display-print-string 4 "|(Hold For HighScores)|
        display-print-string 6 "|      Grassroots|
        display-print-string 7 "|   Invention  Group|
        waituntil [(sensor 1) > 1000]
        check-for-hs-table
        waituntil [(sensor 1) < 1000]
```

```
    ]
end

to check-for-hs-table
    resett
    loop
    [
        if (timer > 2000)
        [
            print-hs-table
            waituntil [(sensor 1) < 1000]
            waituntil [(sensor 1) > 1000]
            stop
        ]
        if ((sensor 1) < 1000)
        [
            setgame-on 1
            stop
        ]
    ]
end

to play-until-die
    loop
    [
        if (lives = -1)
        [
            stop
        ]
        load-level
        ifelse (level < 25)
            [setworm-delay 50 - ((level / 5) * 10)]
            [setworm-delay 0]
        setstart-score score
        mainloop
    ]
end


to load-level
    if ((level % 5) = 0)
    [
        display-clear
        print-level-line
        print-status-line
        wait 20
        init
        draw-worm 20 10 5
```

```
    draw-horiz-wall 5 5 31
    draw-vert-wall 10 8 7
    draw-vert-wall 30 8 7
    repeat 10 [draw-fruit]
    stop
]
if ((level % 5) = 1)
[
    display-clear
    print-level-line
    print-status-line
    wait 20
    init
    draw-worm 20 10 5
    draw-vert-wall 5 3 11
    draw-vert-wall 10 3 11
    draw-vert-wall 15 3 11
    draw-vert-wall 25 3 11
    draw-vert-wall 30 3 11
    draw-vert-wall 35 3 11
    repeat 10 [draw-fruit]
    stop
]
if ((level % 5) = 2)
[
    display-clear
    print-level-line
    print-status-line
    wait 20
    init
    draw-worm 20 10 5
    draw-vert-wall 5 5 7
    draw-horiz-wall 5 11 11
    draw-vert-wall 15 5 7
    draw-horiz-wall 15 5 11
    draw-vert-wall 25 5 7
    draw-horiz-wall 25 11 11
    draw-vert-wall 35 5 7
    repeat 10 [draw-fruit]
    stop
]
if ((level % 5) = 3)
[
    display-clear
    print-level-line
    print-status-line
    wait 20
    init
```

```
            draw-worm 20 10 5
            draw-vert-wall 3 3 11
            draw-horiz-wall 3 3 35
            draw-vert-wall 37 3 11
            draw-horiz-wall 3 13 12
            draw-horiz-wall 26 13 12
            draw-vert-wall 26 6 8
            draw-vert-wall 14 6 8
            repeat 10 [draw-fruit]
            stop
        ]
        if ((level % 5) = 4)
        [
            display-clear
            print-level-line
            print-status-line
            wait 20
            init
            draw-worm 20 10 5
            draw-vert-wall 2 2 13
            draw-vert-wall 6 2 13
            draw-vert-wall 10 2 13
            draw-vert-wall 14 2 13
            draw-vert-wall 18 2 13
            draw-vert-wall 22 2 13
            draw-vert-wall 26 2 13
            draw-vert-wall 30 2 13
            draw-vert-wall 34 2 13
            draw-vert-wall 38 2 13
            draw-horiz-wall 2 14 5
            draw-horiz-wall 6 2 5
            draw-horiz-wall 10 14 5
            draw-horiz-wall 14 2 5
            draw-horiz-wall 22 2 5
            draw-horiz-wall 26 14 5
            draw-horiz-wall 30 2 5
            draw-horiz-wall 34 14 5
            repeat 10 [draw-fruit]
            stop
        ]
    end

    ; Main grid goes form 0,0 to 40,16

    to init
        display-clear
        wait 1
        display-draw-line 1 0 126 0
```

```
    display-draw-line 126 0 126 53
    display-draw-line 1 53 126 53
    display-draw-line 1 53 1 0
    setworm-dir 0
    settail-dir 0
    setbutton-hit 1
    setleft-button-hit 0
    setright-button-hit 0
    setcollision 0
    setgrow 0
    sethit-last-move 0
    setgridx 0
    setgridy 0
    setgridlength 0
    seteven-space 0
    print-status-line
end

to mainloop
    wait 10
    loop
    [
        if ((sensor 1) > 1000)
        [
            waituntil [(sensor 1) < 1000]
            display-print-string 7 "|       -Pause-|
            waituntil [(sensor 1) > 1000]
            waituntil [(sensor 1) < 1000]
            print-status-line
        ]
        setcollision check-for-collision
        if (collision = 2)
        [
            wait 10
            stop
        ]
        if (collision = 1)
        [
            if ((score = (start-score + 100)))
            [
                open-door
            ]
            setgrow 4
        ]
        move-worm
        remove-tail
        if (level-done = 1)
        [
```

```
            setlevel-done 0
            stop
        ]
        check-for-button
        mwait worm-delay
    ]
end

to draw-fruit
    get-valid-fruit-location
    display-set-pixel gridx gridy
    display-set-pixel gridx + 1 gridy
    display-set-pixel gridx gridy + 1
    display-set-pixel gridx + 1 gridy + 1
end

to get-valid-fruit-location
    loop
    [
        setgridx ((random % 41) * 3) + 3
        setgridy ((random % 17) * 3) + 2
        if (not (display-test-pixel gridx gridy))
        [
            stop
        ]
    ]
end

to draw-horiz-wall :x :y :length
    setgridx (:x * 3) + 3
    setgridy (:y * 3) + 2
    setgridlength (:length * 3) - 2
    display-draw-line gridx gridy (gridx + gridlength) gridy
    display-draw-line gridx (gridy + 1) (gridx + gridlength) (gridy + 1)
end

to draw-vert-wall :x :y :length
    setgridx (:x * 3) + 3
    setgridy (:y * 3) + 2
    setgridlength (:length * 3) - 2
    display-draw-line gridx gridy gridx (gridy + gridlength)
    display-draw-line (gridx + 1) gridy (gridx + 1) (gridy + gridlength)
end

to check-for-collision
    ; 0 = Up
    ; 1 = Right
    ; 2 = Down
```

```
; 3 = Left

if (hit-last-move = 1)
[
    sethit-last-move 0
    output 0
]
if (worm-dir = 0)
[
    ifelse (display-test-pixel worm-head-x (worm-head-y - 1))
    [
        ; Fruit or wall?
        ifelse ((display-test-pixel (worm-head-x - 1)
                                    (worm-head-y - 1))
            or (display-test-pixel (worm-head-x + 2)
                                   (worm-head-y - 1))
            or (display-test-pixel worm-head-x (worm-head-y - 3)))
        [
            ; Wall
            setlives lives - 1
            output 2
        ]
        [
            ; Fruit
            sethit-last-move 1
            setscore score + 10
            print-status-line
            output 1
        ]
    ]
    [
        ; Nothing
        output 0
    ]
]
if (worm-dir = 1)
[
    ifelse (display-test-pixel (worm-head-x + 1) worm-head-y)
    [
        ; Fruit or wall?
        ifelse ((display-test-pixel (worm-head-x + 1)
                                    (worm-head-y - 1))
            or (display-test-pixel (worm-head-x + 1)
                                   (worm-head-y + 2))
            or (display-test-pixel (worm-head-x + 3) worm-head-y))
        [
            ; Wall
            setlives lives - 1
```

```
            output 2
        ]
        [
            ; Fruit
            sethit-last-move 1
            setscore score + 10
            print-status-line
            output 1
        ]
    ]
    [
        ; Nothing
        output 0
    ]
]
if (worm-dir = 2)
[
    ifelse (display-test-pixel worm-head-x (worm-head-y + 1))
    [
        ; Fruit or wall?
        ifelse ((display-test-pixel (worm-head-x - 1)
                                    (worm-head-y + 1))
            or (display-test-pixel (worm-head-x + 2)
                                   (worm-head-y + 1))
            or (display-test-pixel worm-head-x (worm-head-y + 3)))
        [
            ; Wall
            setlives lives - 1
            output 2
        ]
        [
            ; Fruit
            sethit-last-move 1
            setscore score + 10
            print-status-line
            output 1
        ]
    ]
    [
        ; Nothing
        output 0
    ]
]
if (worm-dir = 3)
[
    ifelse (display-test-pixel (worm-head-x - 1) worm-head-y)
    [
        ; Fruit or wall?
```

```
            ifelse ((display-test-pixel (worm-head-x - 1)
                                        (worm-head-y - 1))
                or (display-test-pixel (worm-head-x - 1)
                                       (worm-head-y + 2))
                or (display-test-pixel (worm-head-x - 3) worm-head-y))
            [
                ; Wall
                setlives lives - 1
                output 2
            ]
            [
                ; Fruit
                sethit-last-move 1
                setscore score + 10
                print-status-line
                output 1
            ]
        ]
        [
            ; Nothing
            output 0
        ]
    ]
end

to check-for-button
    if (((sensor 5) < 1000) and ((sensor 6) < 1000) and
        ((sensor 7) < 1000) and ((sensor 8) < 1000))
    [
        setbutton-hit 0
    ]
    if (button-hit = 0)
    [
        if ((sensor 5) > 1000)
        [
            setdown-button-hit 1
            setbutton-hit 1
        ]
        if ((sensor 6) > 1000)
        [
            setright-button-hit 1
            setbutton-hit 1
        ]
        if ((sensor 7) > 1000)
        [
            setup-button-hit 1
            setbutton-hit 1
        ]
```

```
    if ((sensor 8) > 1000)
    [
        setleft-button-hit 1
        setbutton-hit 1
    ]
]
if (even-space = 0)
[
    if (worm-dir = 0)
    [
        if (left-button-hit = 1)
        [
            turn-worm 0
        ]
        if (right-button-hit = 1)
        [
            turn-worm 1
        ]
    ]
    if (worm-dir = 1)
    [
        if (up-button-hit = 1)
        [
            turn-worm 0
        ]
        if (down-button-hit = 1)
        [
            turn-worm 1
        ]
    ]
    if (worm-dir = 2)
    [
        if (right-button-hit = 1)
        [
            turn-worm 0
        ]
        if (left-button-hit = 1)
        [
            turn-worm 1
        ]
    ]
    if (worm-dir = 3)
    [
        if (down-button-hit = 1)
        [
            turn-worm 0
        ]
        if (up-button-hit = 1)
```

```
            [
                turn-worm 1
            ]
        ]
        setup-button-hit 0
        setright-button-hit 0
        setdown-button-hit 0
        setleft-button-hit 0
    ]
end

to remove-tail
    if (grow > 0)
    [
        setgrow grow - 1
        stop
    ]
    if (tail-dir = 0)
    [
        if (worm-tail-y = 0)
        [
            display-clear-pixel worm-tail-x worm-tail-y
            display-clear-pixel (worm-tail-x + 1) worm-tail-y
            setlevel level + 1
            setlevel-done 1
            stop
        ]
        ; Going up
        ifelse (display-test-pixel (worm-tail-x - 1) worm-tail-y)
        [
            ; Turning left
            display-clear-pixel (worm-tail-x + 1) (worm-tail-y - 1)
            display-clear-pixel (worm-tail-x + 1) worm-tail-y
            setworm-tail-y (worm-tail-y - 1)
            settail-dir 3
            stop
        ]
        [
            ifelse (display-test-pixel (worm-tail-x + 2) worm-tail-y)
            [
                ; Turning Right
                display-clear-pixel worm-tail-x (worm-tail-y - 1)
                display-clear-pixel worm-tail-x worm-tail-y
                setworm-tail-x (worm-tail-x + 1)
                setworm-tail-y (worm-tail-y - 1)
                settail-dir 1
                stop
            ]
```

```
        [
            ; Going Up
            display-clear-pixel worm-tail-x worm-tail-y
            display-clear-pixel (worm-tail-x + 1) worm-tail-y
            setworm-tail-y (worm-tail-y - 1)
            stop
        ]
    ]
]
if (tail-dir = 1)
[
    ; Going right
    ifelse (display-test-pixel worm-tail-x (worm-tail-y - 1))
    [
        ; Turning Up
        display-clear-pixel worm-tail-x (worm-tail-y + 1)
        display-clear-pixel (worm-tail-x + 1) (worm-tail-y + 1)
        settail-dir 0
        stop
    ]
    [
        ifelse (display-test-pixel worm-tail-x (worm-tail-y + 2))
        [
            ; Turning Down
            display-clear-pixel worm-tail-x worm-tail-y
            display-clear-pixel (worm-tail-x + 1) worm-tail-y
            setworm-tail-y (worm-tail-y + 1)
            settail-dir 2
            stop
        ]
        [
            ; Going Right
            display-clear-pixel worm-tail-x worm-tail-y
            display-clear-pixel worm-tail-x (worm-tail-y + 1)
            setworm-tail-x (worm-tail-x + 1)
            stop
        ]
    ]
]
if (tail-dir = 2)
[
    ; Going down
    ifelse (display-test-pixel (worm-tail-x - 1) worm-tail-y)
    [
        ; Turning left
        display-clear-pixel (worm-tail-x + 1) worm-tail-y
        display-clear-pixel (worm-tail-x + 1) (worm-tail-y + 1)
        settail-dir 3
```

```
            stop
        ]
        [
            ifelse (display-test-pixel (worm-tail-x + 2) worm-tail-y)
            [
                ; Turning Right
                display-clear-pixel worm-tail-x worm-tail-y
                display-clear-pixel worm-tail-x (worm-tail-y + 1)
                setworm-tail-x (worm-tail-x + 1)
                settail-dir 1
                stop
            ]
            [
                ; Going Down
                display-clear-pixel worm-tail-x worm-tail-y
                display-clear-pixel (worm-tail-x + 1) worm-tail-y
                setworm-tail-y (worm-tail-y + 1)
                stop
            ]
        ]
    ]
    if (tail-dir = 3)
    [
        ; Going left
        ifelse (display-test-pixel worm-tail-x (worm-tail-y - 1))
        [
            ; Turning Up
            display-clear-pixel (worm-tail-x - 1) (worm-tail-y + 1)
            display-clear-pixel worm-tail-x (worm-tail-y + 1)
            setworm-tail-x (worm-tail-x - 1)
            settail-dir 0
            stop
        ]
        [
            ifelse (display-test-pixel worm-tail-x (worm-tail-y + 2))
            [
                ; Turning Down
                display-clear-pixel (worm-tail-x - 1) worm-tail-y
                display-clear-pixel worm-tail-x worm-tail-y
                setworm-tail-x (worm-tail-x - 1)
                setworm-tail-y (worm-tail-y + 1)
                settail-dir 2
                stop
            ]
            [
                ; Going Left
                display-clear-pixel worm-tail-x worm-tail-y
                display-clear-pixel worm-tail-x (worm-tail-y + 1)
```

```
                setworm-tail-x (worm-tail-x - 1)
                stop
            ]
        ]
    ]
end

to draw-worm :x :y :length
    draw-vert-wall :x :y :length
    setworm-head-x (:x * 3) + 3
    setworm-head-y (:y * 3) + 2
    setworm-tail-x (:x * 3) + 3
    setworm-tail-y (:y * 3) + (:length * 3)
end

to move-worm
    ; 0 = Up
    ; 1 = Right
    ; 2 = Down
    ; 3 = Left
    if (worm-head-y = 0)
    [
        stop
    ]
    ifelse (even-space = 0)
    [
        seteven-space 2
    ]
    [
        seteven-space even-space - 1
    ]
    if (worm-dir = 0)
    [
        setworm-head-y worm-head-y - 1
        display-set-pixel worm-head-x worm-head-y
        display-set-pixel (worm-head-x + 1) worm-head-y
        stop
    ]
    if (worm-dir = 1)
    [
        setworm-head-x worm-head-x + 1
        display-set-pixel worm-head-x worm-head-y
        display-set-pixel worm-head-x (worm-head-y + 1)
        stop
    ]
    if (worm-dir = 2)
    [
        setworm-head-y worm-head-y + 1
```

```
        display-set-pixel worm-head-x worm-head-y
        display-set-pixel (worm-head-x + 1) worm-head-y
        stop
    ]
    if (worm-dir = 3)
    [
        setworm-head-x worm-head-x - 1
        display-set-pixel worm-head-x worm-head-y
        display-set-pixel worm-head-x (worm-head-y + 1)
        stop
    ]
end

to turn-worm :dir
    ; 0 = turn left
    ; 1 = turn right
    if (worm-dir = 0)
    [
        if (:dir = 0)
        [
            setworm-dir 3
        ]
        if (:dir = 1)
        [
            setworm-dir 1
            setworm-head-x worm-head-x + 1
        ]
        stop
    ]
    if (worm-dir = 1)
    [
        if (:dir = 0)
        [
            setworm-dir 0
            setworm-head-x worm-head-x - 1
        ]
        if (:dir = 1)
        [
            setworm-dir 2
            setworm-head-x worm-head-x - 1
            setworm-head-y worm-head-y + 1
        ]
        stop
    ]
    if (worm-dir = 2)
    [
        if (:dir = 0)
        [
```

```
            setworm-dir 1
            setworm-head-x worm-head-x + 1
            setworm-head-y worm-head-y - 1
        ]
        if (:dir = 1)
        [
            setworm-dir 3
            setworm-head-y worm-head-y - 1
        ]
        stop
    ]
    if (worm-dir = 3)
    [
        if (:dir = 0)
        [
            setworm-dir 2
            setworm-head-y worm-head-y + 1
        ]
        if (:dir = 1)
        [
            setworm-dir 0
        ]
        stop
    ]
end

to print-status-line
    display-clear-line 7
    display-print-string 7 "Score:
    display-print-num 7 (score / 10000)
    display-print-num 7 ((score / 1000) % 10)
    display-print-num 7 ((score / 100) % 10)
    display-print-num 7 ((score / 10) % 10)
    display-print-num 7 (score % 10)
    display-print-string 7 "|   Lives:|
    display-print-num 7 lives
end

to open-door
    display-clear-pixel 62 0
    display-clear-pixel 63 0
    display-clear-pixel 64 0
    display-clear-pixel 65 0
end

to print-hs-table
    display-clear
    display-print-string 0 "|  High Score Table:|
```

```
    display-print-string 1 "|--------------------|
    setn 1
    repeat 6 [print-high-score-line n setn n + 1]
end

to print-high-score-line :n
    seths-name-char0 read-hs-name 0 :n
    seths-name-char1 read-hs-name 1 :n
    seths-name-char2 read-hs-name 2 :n
    seths-name-char3 read-hs-name 3 :n
    seths-name-char4 read-hs-name 4 :n
    seths-name-char5 read-hs-name 5 :n
    seths read-hs :n
    seths-day read-hs-day :n
    seths-month-char0 read-hs-month 0 :n
    seths-month-char1 read-hs-month 1 :n
    seths-month-char2 read-hs-month 2 :n
    display-print-num (:n + 1) :n
    display-print-string (:n + 1) ":
    display-print-char (:n + 1) hs-name-char0
    display-print-char (:n + 1) hs-name-char1
    display-print-char (:n + 1) hs-name-char2
    display-print-char (:n + 1) hs-name-char3
    display-print-char (:n + 1) hs-name-char4
    display-print-char (:n + 1) hs-name-char5
    display-print-string (:n + 1) "| |
    display-print-num (:n + 1) (hs / 10000)
    display-print-num (:n + 1) ((hs / 1000) % 10)
    display-print-num (:n + 1) ((hs / 100) % 10)
    display-print-num (:n + 1) ((hs / 10) % 10)
    display-print-num (:n + 1) (hs % 10)
    display-print-string (:n + 1) "| |
    display-print-char (:n + 1) hs-month-char0
    display-print-char (:n + 1) hs-month-char1
    display-print-char (:n + 1) hs-month-char2
    display-print-string (:n + 1) ".
    display-print-num (:n + 1) ((hs-day / 10) % 10)
    display-print-num (:n + 1) (hs-day % 10)
end

to read-hs :n
    output (256 * (ee-read 0 (:n * 12) + 7)) + (ee-read 0 (:n * 12) + 6)
end

to read-hs-name :char :n
    output ee-read 0 (:n * 12) + :char
end
to read-hs-day :n
```

```
    output ee-read 0 (:n * 12) + 11
end

to read-hs-month :char :n
    output ee-read 0 (:n * 12) + :char + 8
end

to print-name-enter-line
    display-clear-line 5
    display-print-string 5 "|        |
    display-print-char 5 hs-name-char0
    display-print-char 5 hs-name-char1
    display-print-char 5 hs-name-char2
    display-print-char 5 hs-name-char3
    display-print-char 5 hs-name-char4
    display-print-char 5 hs-name-char5
end

to get-name-char :n
    set-global (*hs-name-char0 + (2 * :n)) 65
    print-name-enter-line
    loop
    [
        if ((sensor 6) > 1000)
        [
            set-global (*hs-name-char0 + (2 * :n))
                       (get-global (*hs-name-char0 + (2 * :n))) + 1
            if ((get-global (*hs-name-char0 + (2 * :n))) = 127)
            [
                set-global (*hs-name-char0 + (2 * :n)) 32
            ]
            print-name-enter-line
            resett
            waituntil [((sensor 6) < 1000) or (timer > 200)]
        ]
        if ((sensor 8) > 1000)
        [
            set-global (*hs-name-char0 + (2 * :n))
                       (get-global (*hs-name-char0 + (2 * :n))) - 1
            if ((get-global (*hs-name-char0 + (2 * :n))) = 31)
            [
                set-global (*hs-name-char0 + (2 * :n)) 126
            ]
            print-name-enter-line
            waituntil [((sensor 8) < 1000) or (timer > 200)]
        ]
        if ((sensor 1) > 1000)
        [
```

```
            waituntil [(sensor 1) < 1000]
            stop
        ]
    ]
end

to clock-read :addr
    i2c-start
    i2c-write-byte $0e
    i2c-write-byte 2
    i2c-write-byte 1
    i2c-write-byte :addr
    i2c-stop
    i2c-start
    i2c-write-byte $0f
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output ((i2c-byte / 16) * 10) + (i2c-byte % 16)
end

to get-month-name :n
    if (:n = 1)
    [
        seths-month-char0 74
        seths-month-char1 97
        seths-month-char2 110
        stop
    ]
    if (:n = 2)
    [
        seths-month-char0 70
        seths-month-char1 101
        seths-month-char2 98
        stop
    ]
    if (:n = 3)
    [
        seths-month-char0 77
        seths-month-char1 97
        seths-month-char2 114
        stop
    ]
    if (:n = 4)
    [
        seths-month-char0 65
        seths-month-char1 112
        seths-month-char2 114
```

```
    stop
]
if (:n = 5)
[
    seths-month-char0 77
    seths-month-char1 97
    seths-month-char2 121
    stop
]
if (:n = 6)
[
    seths-month-char0 74
    seths-month-char1 117
    seths-month-char2 110
    stop
]
if (:n = 7)
[
    seths-month-char0 74
    seths-month-char1 117
    seths-month-char2 108
    stop
]
if (:n = 8)
[
    seths-month-char0 65
    seths-month-char1 117
    seths-month-char2 103
    stop
]
if (:n = 9)
[
    seths-month-char0 83
    seths-month-char1 101
    seths-month-char2 112
    stop
]
if (:n = 10)
[
    seths-month-char0 79
    seths-month-char1 99
    seths-month-char2 116
    stop
]
if (:n = 11)
[
    seths-month-char0 78
    seths-month-char1 111
```

```
        seths-month-char2 118
        stop
    ]
    if (:n = 12)
    [
        seths-month-char0 68
        seths-month-char1 101
        seths-month-char2 99
        stop
    ]
end

to print-level-line
    display-print-string 3 "|      Level:|
    display-print-num 3 level + 1

end

to display-clear
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 1
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end

to display-clear-line :line
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 2
    i2c-write-byte 5
    i2c-write-byte :line
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end

to display-print-string :line :string
    setdisp-nn display-get-string-length :string
    i2c-start
```

```
    i2c-write-byte $16
    i2c-write-byte disp-nn + 2
    i2c-write-byte 6
    i2c-write-byte :line
    setdisp-n :string
    repeat disp-nn
    [
        i2c-write-byte read-prog-mem disp-n setdisp-n disp-n + 1
    ]
    i2c-stop
    i2c-start
    i2c-write-byte $17
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
end

to display-get-string-length :string
    setdisp-n :string
    loop
    [
        ifelse ((read-prog-mem disp-n) = 0)
            [output disp-n - :string]
            [setdisp-n disp-n + 1]
    ]
end

to display-print-char :line :char
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 7
    i2c-write-byte :line
    i2c-write-byte :char
    i2c-stop
end

to display-set-line-index :line :index
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 8
    i2c-write-byte :line
    i2c-write-byte :index
    i2c-stop
end

to display-set-pixel :x :y
```

```
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 9
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
end

to display-clear-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 10
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
end

to display-test-pixel :x :y
    i2c-start
    i2c-write-byte $16
    i2c-write-byte 3
    i2c-write-byte 11
    i2c-write-byte :x
    i2c-write-byte :y
    i2c-stop
    i2c-start
    i2c-write-byte $17
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end

to display-print-num :line :n
    if :n > 9999 [display-print-digit :line :n 10000]
    if :n > 999 [display-print-digit :line :n 1000]
    if :n > 99 [display-print-digit :line :n 100]
    if :n > 9 [display-print-digit :line :n 10]
    display-print-digit :line :n 1
end

to display-print-digit :line :n :d
    display-print-char :line ((:n / :d) % 10) + 48
end
```

```
to display-draw-line :x1 :y1 :x2 :y2
    if (:x1 = :x2)
        [ifelse (:y2 > :y1)
            [
                setn 0
                repeat (:y2 - :y1) + 1
                       [display-set-pixel :x1 :y1 + n setn n + 1]
                stop
            ]
            [
                setn 0
                repeat (:y1 - :y2) + 1
                       [display-set-pixel :x1 :y2 + n setn n + 1]
                stop
            ]
        ]
    if (:y1 = :y2)
        [ifelse (:x2 > :x1)
            [
                setn 0
                repeat (:x2 - :x1) + 1
                       [display-set-pixel :x1 + n :y1 setn n + 1]
                stop
            ]
            [
                setn 0
                repeat (:x1 - :x2) + 1
                       [display-set-pixel :x2 + n :y1 setn n + 1]
                stop
            ]
        ]
end

to mwait :n
    resett
    waituntil [timer = :n]
end
```

Appendix Z: Application Code - Engine Meter

```
; ************************************************************************
; Diesel Engine Meter- A program for measuring timing patterns on a
;                      diesel engine flywheel.  An optical interrupter
;                      switch is used to observe rotating fins, while
;                      timing data is stored in a buffer.  When 320
;                      datapoints have been taken, the data is sent
;                      over 2400 baud serial to a host computer.  Two
;                      LEDs are used for status monitoring.  The green
;                      LED blinks in sync with switch events, and the
;                      red LED toggles when the data buffer is full and
;                      output is generated.
;
; Chris Lyon, scooby@mit.edu
; Isaac Chuang, ichuang@media.mit.edu
; Amy Sun, asun@media.mit.edu
;
;     Needed:
;          PIC Foundation
;            Proto Layer
;            RS-232 Module
;     Attach:
;            Pin B2 - Green LED
;            Pin B3 - Red LED
;            Pin C2 - Optical Interrupter (this pin needed for capture)
; ************************************************************************

; Global Constant Declarations
; basic config
    [const @ 0]
    [const timer 1]
    [const pcl 2]
    [const status 3]
    [const c 0][const z 2][const bankl 5][const bankh 6][const ibank 7]
    [const @@ 4]
    [const porta 5][const porta-ddr $85]
    [const portb 6][const portb-ddr $86]
    [const portc 7][const portc-ddr $87]
    [const portd 8][const portd-ddr $88]
    [const porte 9][const porte-ddr $89]
    [const option 1] ; bank 1
; ad conversion
    [const adres $1e]
    [const adcon0 $1f]
    [const adcon1 $1f] ;bank 1
```

```
    [const adon 0][const adgo 2]
; interrupts
    [const intcon $0b]
    [const gie 7][const peie 6][const t0ie 5][const inte 4]
    [const intf 1][const t0if 2]
    [const pir1 $0c]
    [const rcif 5][const txif 4][const sspif 3][const ccp1if 2]
    [const pie1 $0c] ; bank 1
    [const ccp1ie 2]
; timer
    [const tmr1l $0e]
    [const tmr1h $0f]
; flash
    [const eedatah $0e][const eedatal $0c] ;bank 2
    [const eeaddrh $0f][const eeaddrl $0d] ;bank 2
    [const eecon1 $0c] ; bank 3
    [const rd 0][const wr 1][const wren 2][const eepgd 7]
    [const eecon2 $0d] ;bank 3 ; timer
    [const t1con $10]
; ssp
    [const sspbuf $13]
    [const sspcon1 $14] ; bank 0
    [const sspen 5][const sspm3 3]
    [const sspcon2 $11] ; bank 1
    [const gcen 7][const ackstat 6][const ackdt 5][const acken 4]
    [const rcen 3][const pen 2][const rsen 1][const sen 0]
    [const sspadd $13] ; bank 1
    [const sspstat $14] ; bank 1
    [const smp 7][const bf 0]
; capture
    [const ccpr1l $15]
    [const ccpr1h $16]
    [const ccp1con $17]
; uart
    [const baud $19] ; bank 1
    [const txctrl $18] ; bank 1
    [const rxctrl $18]
    [const txreg $19]
    [const rxreg $1a]

; ram locations - all in the shared high bytes of the ram banks
; these are our primary local variables
    [const tmp $78] ; tmp 16 bit register - low byte
    [const tmp2 $79] ; tmp 16 bit register - high byte
    [const mybank $7a] ; what bank we're using for the data buf
    [const data $7b] ; data - low byte
    [const data2 $7c] ; data - high byte
    [const counter $7d] ; general purpose counter - used in putdec
```

```
    [const int-a $7e]
    [const int-status $7f]


;-------------------------------------------------------------------------
; performance parameters
    [const capture-mode $05] ; trig on every rising edge
    [const timer-mode $31] ; div by 8 = 4 us ((8mhz/4)/8))


;-------------------------------------------------------------------------
; Main program: starts here
start
    [bsr io-init] ; run io-init
    [bsr greeting]
mainloop ; infinite loop - we skip the middle man!
    [bra mainloop] ; program is entirely interrupt driven


;-------------------------------------------------------------------------
; subroutine: print greeting, with useful parameters
greeting
    [ldan $65] ; 'e'
    [bsr serial-send]
    [ldan $33] ; '3'
    [bsr serial-send]
    [ldan $3a] ; ':'
    [bsr serial-send]
    [ldan $20] ; ' '
    [bsr serial-send]
    [ldan capture-mode]
    [bsr puthex]
    [ldan timer-mode]
    [bsr puthex]
    [bra crlf]


;-------------------------------------------------------------------------
; subroutine: test routines
test-mem ; should store 0-255 then dump
    [ldan $20] ; initial address for data buffer
    [sta @@] ; store in FSR
    [ldan $0] ; initial bank
    [sta mybank]
    [ldan 0]
    [sta data]
tm_loop
    [lda data] ; fill data buf with 0-255
```

```
    [bsr push-byte]
    [incsz data]
    [bra tm_loop]
    [bra dumpdat]
test-pd16
    [ldan $30] ; should output 12345
    [sta data2]
    [ldan $39]
    [sta data] ; $3039 = 12345
    [bsr putdec16]
    [ldan $00] ;
    [sta data2]
    [ldan $00]
    [sta data] ; should output zero
    [bra putdec16]
test-putdec ; output 000-255 in decimal
    [ldan 0] ; test routine for putdec
    [sta mybank]
tpd_lp [lda mybank] ; output in hex
    [bsr putdec] ;
    [ldan 32] ; output space
    [bsr serial-send]
    [incsz mybank] ; increment, just once through
    [bra tpd_lp]
    [bra mainloop]


;-------------------------------------------------------------------------
; serial output routine
serial-send
    [btss txif pir1]
    [bra serial-send] ; wait until the serial out buffer is clear
    [sta txreg] ; store the value in the tx holding register
    [rts]


;-------------------------------------------------------------------------
; I/O initialization
io-init
    [bclr 2 portb] ; clear both led pins
    [bclr 3 portb]
    ; buffer pointer init
    [ldan $20] ; initial address for data buffer
    [sta @@] ; store in FSR
    [ldan $0] ; initial bank
    [sta mybank]
    [bset bankl status] ; set led pins as outputs
    [bclr 2 portb]
```

```
    [bclr 3 portb]
    ; serial port init
    [ldan 51][sta baud] ; 51 = 2400 baud (@8mhz) ; 12 = 9600
    [ldan $20][sta txctrl] ; enable the transmitter
    [bclr bankl status]
    [ldan $90][sta rxctrl] ; and the receiver
    ; timer init
    [clr tmr1l] ; reset timer 1 module
    [clr tmr1h]
    [ldan timer-mode] ; set timer 1 prescaler & turn on
    [sta t1con] ; should give resolution of 4 us ((8mhz/4)/8))
    ; capture module init
    [ldan capture-mode] ; enable compare mod1
    [sta ccp1con]
    [bset bankl status] ; bank 1
    [bset ccp1ie pie1] ; enable capture interrupt
    [bclr bankl status] ; bank 0
    [bclr ccp1if pir1] ; clear the capture interrupt flag
    [bset peie intcon] ; enable peripheral interrupts (Capture is one)
    [bset gie intcon] ; general interrupt enable
    [rts]


;-------------------------------------------------------------------------
; interrupt routine
int-routine ; If we made it here, we've triggered a rising edge
    [sta int-a] ; save the accumulator
    [lda status]
    [sta int-status] ; save status
    [bclr bankl status] ; make sure we're in bank 0
    [bclr bankh status]
    [lda portb] ; toggle green LED every pulse
    [xorn $04]
    [sta portb]
    [lda ccpr1h] ; get high byte of timer1, and save in data buf
    [bsr push-byte]
    [lda ccpr1l]
    [bsr push-byte] ; get low byte of timer1, and save in data buf
    ; is the data buffer full? check mybank for overflow
    [btss 2 mybank] ; if bit 2 = 1 then overflow: end of buffer
    [bra iret] ; not overflow - go to interrupt return
    ; we have a full buffer - dump to serial
    [lda portb] ; toggle red LED to indicate buffer full
    [xorn $08]
    [sta portb]
    [ldan $20] ; start address for data buffer
    [sta @@] ; store in FSR
    [ldan $0] ; initial bank
```

```
    [sta mybank]
    [bsr dumpdat] ; dump data to serial port
iret ; interrupt return
    [clr tmr1l] ; clear timer1 registers
    [clr tmr1h]
    [bclr ccp1if pir1] ; clear the interrupt flag
    [lda int-status] ; restore status and accumulator registers
    [sta status]
    [swap int-a] ; tricky way of doing an lda without
    [lswap int-a] ; affecting the status
    [rti]


;-----------------------------------------------------------------------
; subroutine: dump data buffer to serial port
dumpdat
    [ldan $20] ; start address for data buffer
    [sta @@] ; store in FSR
    [ldan $0] ; initial bank
    [sta mybank]
ddloop
    [bsr read-byte-inc] ; read byte (high)
    [sta data2] ; save as high byte
    [bsr read-byte-inc] ; read byte (low)
    [bsr putdec16] ; output to serial port
    [ldan 32] ; send space
    [bsr serial-send]
    [btss 2 mybank] ; if bit 2 = 1 then end of buffer
    [bra ddloop] ; not overflow - keep looping
    [ldan $20] ; initial address for data buffer
    [sta @@] ; store in FSR
    [ldan $0] ; reset the buffer pointer
    [sta mybank]
crlf
    [ldan 10] ; send crlf
    [bsr serial-send]
    [ldan 13]
    [bra serial-send]


;-----------------------------------------------------------------------
; subroutine: push byte onto data buffer and increment pointer
push-byte
    [bclr ibank status] ; set bank according to lowest two
    [bclr 7 @@] ; bits of mybank (status.ibank and fsr.7)
    [btsc 0 mybank]
    [bset 7 @@] ; fsr.7 is low bit of bank
    [btsc 1 mybank]
```

```
    [bset ibank status] ; status.7 (IRP) is high bit of bank
    [sta @] ; store data in location pointed to by FSR
inc-ptr
    [bclr 7 @@] ; clear bit 7 so we can detect end of this bank
    [inc @@] ; increment FSR
    ; $20 = 0010 0000, $70 = 0111000
    [btss 6 @@] ; test for $70 overflow (only use $20-$6f)
    [rts] ; return to caller now if FSR.6 is clear
    [btss 5 @@] ; return to caller now if FSR.5 is clear
    [rts] ; [btss 4 @@] ; return to caller now if FSR.4 is clear
    [rts] ;
    [ldan $20] ; reset FSR
    [sta @@]
    [inc mybank] ; move to next bank (only bits 0,1 matter)
    [rts] ; return to caller


;-------------------------------------------------------------------------
; subroutine: read byte from buffer and inc pointer - return in data
read-byte-inc
    [bclr ibank status] ; set bank according to lowest two
    [bclr 7 @@] ; bits of mybank (status.ibank and fsr.7)
    [btsc 0 mybank]
    [bset 7 @@] ; fsr.7 is low bit of bank
    [btsc 1 mybank]
    [bset ibank status] ; status.7 (IRP) is high bit of bank
    [lda @] ; store data in location pointed to by FSR
    [sta data]
    [bra inc-ptr] ; branch to increment pointer


;-------------------------------------------------------------------------
; subroutine: hex output function
puthex
    [sta data] ; save the value
    [sta data2] ; save a second copy
    [ror data2] ; rotate through 4 times to right to grab high nibble
    [ror data2] ; should be able to use swap here
    [ror data2]
    [lror data2]
    [bsr nibble-out] ; send the high nibble
    [lda data] ; reload the value
    ; send the low nibble
nibble-out
    [andn $0f] ; strip off high nibble
    [sta data2] ; save it
    [subn 9] ; are we greater than 9?
    [lda data2] ; reload the accumulator with the data
```

```
    [btss c status] ; if we're greater than 9
    [addn 7] ; add 7 to the ascii value will be a/b/c/d/e/f
output-num
    [addn 48] ; add 48 to get the ascii decimal value
    [bsr serial-send] ; send out the byte
    [rts]


;-----------------------------------------------------------------------
; subroutine: decimal output function - call with data in accumulator
; algorithm:
; cnt = 0;
; while(x >= 100) { cnt++; x -= 100; }
; print cnt;
; cnt = 0;
; while(x >= 10) { cnt++; x -= 10; }
; print cnt;
; print x;
putdec
    [sta data] ; save data
    [ldan 0] ; zero counter
    [sta counter] ;
    [ldan 100] ; start with hundred's digit
    [bsr pd_digit]
    [ldan 10] ; ten's digit
    [bsr pd_digit]
    [lda data]
    [bra pd_dig2] ; output final one's digit
pd_digit ; output digit, number to sub in acc
    [subm data] ; subtract acc from data and store in data
    [btsc c status] ; C=0 means result negative
    [bra pd_pos] ; result was positive
    [addm data] ; result was negative: add back
    [lda counter] ; output counter value
pd_dig2
    [addn $30]
    [bsr serial-send] ; send to output
    [ldan 0] ; zero counter
    [sta counter] ;
    [rts]
pd_pos
    [inc counter] ; increment counter
    [bra pd_digit] ; loop


;-----------------------------------------------------------------------
; 16-bit subtraction: subtract tmp from data
sub16
```

```
    [lda tmp]
    [subm data] ; data -= tmp
    [btss c status] ; C=0 means result negative
    [bra sub16c] ; negative! handle carry
    [lda tmp2] ; positive! no carry to high byte
    [subm data2] ; data2 -= tmp2 [rts]
sub16c
    [inc tmp2] ; works for our limited cases of interest
    [lda tmp2] ; and the resulting carry is right
    [subm data2] ; data2 -= tmp2+1
    [dec tmp2]
    [rts]


;-------------------------------------------------------------------------
; 16-bit addition: add tmp to data
add16
    [lda tmp]
    [addm data] ; data += tmp
    [btsc c status] ; C=1 means carry needed
    [inc data2] ; carry set! handle carry by inc
    [lda tmp2] ; no carry to high byte
    [addm data2] ; data2 += tmp2
    [rts]


;-------------------------------------------------------------------------
; print out 16-bit decimal number - call with the value in [data2,data]
putdec16
    [ldan 0] ; zero counter
    [sta counter] ;
    [ldan $27] ; start with 10000 digit
    [sta tmp2]
    [ldan $10]
    [sta tmp] ; $2710 = 10,000
    [bsr pd16_digit]
    [ldan $3] ; next: 1000 digit
    [sta tmp2]
    [ldan $e8]
    [sta tmp] ; $3e8 = 1,000
    [bsr pd16_digit]
    [ldan $0] ; next: 100 digit
    [sta tmp2]
    [ldan 100]
    [sta tmp]
    [bsr pd16_digit]
    [ldan 10] ; ten's digit
    [bsr pd_digit]
```

```
    [lda data]
    [bra pd_dig2] ; output final one's digit
pd16_digit ; output digit, number to sub in [tmp2,tmp]
    [bsr sub16] ; subtract tmp from data and store in data
    [btsc c status] ; C=0 means result negative
    [bra pd16_p] ; result was positive
    [bsr add16] ; result was negative: add back
    [lda counter] ; output counter value
    [addn $30]
    [bsr serial-send] ; send to output
    [ldan 0] ; zero counter
    [sta counter] ;
    [rts]
pd16_p
    [inc counter] ; increment counter
    [bra pd16_digit] ; loop
```