UNIVERSITY OF CALIFORNIA
Santa Barbara

Support for Resource Constrained Microcontroller Programming by a Broad Developer Community

TR ID: 2010-24

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Media Arts and Technology

by

Amichi Amar

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Curtis Roads

Professor Steven Butner

December 2010

The Dissertation of
Amichi Amar is approved:

Professor Curtis Roads

Professor Steven Butner

Professor Chandra Krintz, Committee Chairperson

December 2010

Support for Resource Constrained Microcontroller Programming by a Broad Developer Community

TR ID: 2010-24

Copyright © 2010

by

Amichi Amar

Dedication and Gratitude

This dissertation is dedicated my grandfather Yaakov Amar, Z"L.

I would like to thank two women in my life that have been guiding lights through this journey, without their support, guidance and sympathy this work may not have been possible: my mother Shoshana Wolf-Amar and my advisor Professor Chandra Krintz.

I would like to thank Bob Frankel for his contribution to this work. I very much enjoyed all our coffee talk programming language design sessions. Thank you for your time and effort.

I would like to thank the Media Arts & Technology program for their support and the opportunity to pursue this work. And thank you all my MAT friends that helped make the whole experience that much better!

IF

If you can keep your head when all about you
Are losing theirs and blaming it on you,
If you can trust yourself when all men doubt you
But make allowance for their doubting too,
If you can wait and not be tired by waiting,
Or being lied about, don't deal in lies,
Or being hated, don't give way to hating,
And yet don't look too good, nor talk too wise:

If you can dream - and not make dreams your master,
If you can think - and not make thoughts your aim;
If you can meet with Triumph and Disaster
And treat those two impostors just the same;
If you can bear to hear the truth you've spoken
Twisted by knaves to make a trap for fools,
Or watch the things you gave your life to, broken,
And stoop and build'em up with worn-out tools:

If you can make one heap of all your winnings
And risk it all on one turn of pitch-and-toss,
And lose, and start again at your beginnings
And never breath a word about your loss;
If you can force your heart and nerve and sinew
To serve your turn long after they are gone,
And so hold on when there is nothing in you
Except the Will which says to them: "Hold on!"

If you can talk with crowds and keep your virtue,
Or walk with kings - nor lose the common touch,
If neither foes nor loving friends can hurt you;
If all men count with you, but none too much,
If you can fill the unforgiving minute
With sixty seconds' worth of distance run,
Yours is the Earth and everything that's in it,
And - which is more - you'll be a Man, my son!

- Rudyard Kipling

Acknowledgements

The image in Figure 2.2 is ifixit.com content licensed under the open source Creative Commons license.

Curriculum Vitæ

Amichi Amar

Education

2010	**Doctor of Philosophy in Media Arts and Technology** University of California, Santa Barbara.
2006	**Master of Professional Studies** Interactive Telecommunications Program, New York University, New York.
2001	**Bachelor of Science in Computer Science** College of Creative Studies, University of California, Santa Barbara.

Experience

2009	**Course Developer & Instructor,** Modular Embedded Systems Programming, University of California, Santa Barbara.
2008	**Instructor,** Sensors and Interfaces for Media Art, University of California, Santa Barbara.
2006-2007	**Course Developer & Instructor,** Digital Design Studio for Creative Application Development, University of California, Santa Barbara.
2004-2006	**Freelance Technology Designer,** New York, New York.
2003	**Lead Hardware & Software Engineer,** Engine 27 Sound Gallery for Spatial Audio, New York, New York.
2002-2003	**Lead Developer,** Studio for Electro-Instrumental Music (STEIM), Amsterdam, The Netherlands.

Awards

2006-2010	**Regents Special Fellowship,** University of California, Santa Barbara.
2005	**Individual Artist Grant Recipient,** New York State Council on the Arts.
2002-2004	**Departamental Fellowship,** Interactive Telecommunications Program, New York University.

Publications

”The Bass-Station: a community based information space.” Authors: A. Amar and M. Argo. Proceedings of the ACM SIGGRAPH Conference, June 2003.

”WiFisense.” Authors: A. Amar and M. Iossifova. Proceedings of the ACM Conference on Ubiquitous Computing, September 2003.

Field of Study: Media Arts and Technology, Computer Science

Abstract

Support for Resource Constrained Microcontroller Programming by a Broad Developer Community

TR ID: 2010-24

Amichi Amar

Resource constrained microcontrollers with as little as several hundred bytes of RAM and a few dozen megahertz of processing power are the most prevalent computing devices on earth. Microcontrollers and the many application components that interface to them, such as sensors, actuators, transceivers and displays are now cheap and readily available. Once costly development tools are now downloadable from the Internet and usable for free. Interest in application development using resource constrained microcontrollers has expanded beyond embedded system engineers to a broad audience of those that include artists, designers, students and product developers in all sectors of industry and fields of research.

Developing application software for resource constrained systems is a complex process. The lack of microcontroller resources preclude the use of modern high-level programming languages and operating systems. Modern development practices that support uniform software development across hardware platforms are virtually non-existent. Additionally, device manufacturers adhere to customer lock-in business prac-

tices making compatibility between vendor tools hard to come by and transitions between vendor technologies costly and time consuming.

The focus of this dissertation is to support the development of software for resource constrained microcontroller-based systems by an audience with a broad range of technical skills. Our goal is to support uniform development for a diversity of application categories on heterogeneous hardware. Specifically, we design, implement and evaluate a new high-level programming language called Em with constructs and support for modularity, abstraction, software reuse, portability and reconfigurability for differing application requirements, hardware configurations, and quantities of runtime resources. For additional application development support we design, implement in Em, and evaluate a hardware abstraction layer and model for runtime concurrency.

Our empirical results indicate that high-level language constructs can effectively be used in a resource constrained environment and achieve at least equivalent resource utilization compared to C and other related systems. We show through a demonstration and evaluation of real applications how we can support modern software development practices for authoring reusable, configurable, portable software for a diversity of hardware platforms. A hardware abstraction layer and a runtime model for concurrency provide additional support for development by, respectively, providing uniform interfaces to hardware functionality and relieving developers from individually implementing concurrency mechanisms. Finally, we conduct a user study with university students

showing how non-embedded systems experts and people with generally less technical expertise can successfully learn and develop non-trivial applications with Em.

Contents

List of Figures

Chapter 1

Introduction

Microcontrollers are small programmable computers with integrated CPU, memories and peripherals. These computers currently outnumber humans on our planet five to one. 98% of CPUs manufactured globally each year are embedded and over 55% go into microcontrollers [10] with as little as a few hundred *bytes* of RAM, several *kilobytes* of program memory, and at most a few dozen *megahertz* of processing power. These resource constrained computing devices interface our physical world to the digital and are embedded into all facets of our lives; they are in the roads we drive on, the buildings we work in, the walls that house us, the vehicles that carry us, and even our own bodies.

Systems that integrate microcontrollers are small, highly customizable, low-cost and readily available. Dozens of online electronic component retailers exist offering microcontrollers, accelerometers, wireless radios, small displays and other components at low cost. The printed circuit boards that host the components can be fabricated

rapidly in any shape and size, ridged and flexible. Devices can be created to fit into almost any form factor. The programmability of the microcontroller allows it to be precisely customized for applications in diverse industries and fields of research.

A diverse audience is attracted to the applicability of resource constrained systems. Artists are creating interactive artworks and performances that bridge the physical and the virtual [61, 41]. Designers are prototyping new technological products and expe-riences [35, 31]. Students from humanities to engineering sciences are learning the fundamentals of technology, hands-on [50, 88]. Experts in various non-engineering disciplines are creating custom instruments and tools to enhance their research and development.

Software development for these resource constrained systems present challenges not encountered in resource rich environments. Unlike our desktop and server computers, high level programming languages cannot be used because their high degree of dynamism and rich runtime environments compromise efficient use of scarce resources. General purpose operating systems that abstract low-level hardware details and provide services for concurrency and resource management do not fit within available resources. Development tools and support is offered by device manufactures whose interest strictly lies in developers using their devices. As such, there are technical challenges and little incentive for uniform development across hardware platforms.

Attempts to advance the state-of-the-art in software development for micro-controller-based systems remain limited to specific application domains and only accessible to highly skilled developers. Languages and tools that attempt to improve the development practice target narrow application domains such as wireless sensor networks [81]. While these systems possess appealing features of modern programming languages, their use remains accessible only to highly trained researchers and engineers. More of an incremental advancement of the languages that exist, these systems do little to support a diversity of hardware and broad application categories.

Development tools for non-experts either trivialize the user's capabilities, compromise critical system resources, or suffer from similar pitfalls of expert tools. To remain accessible to a generally less technical audience, extant tools often limit the customizability of functionality and integration of hardware to what has been provided by the tool maker. Other systems, in order to allow the use of expressive high-level languages, impose a requirement to tether devices to resource rich hosts which control the device remotely. Platforms that attempt to simplify and make the dominant languages and technologies used by experts more accessible to non-experts, still fail to provide more structured programming methodologies and support for more diverse hardware.

Support for resource constrained application development by a broad developer audience is essential to enabling a future generation of progress and innovation. The growing audience of non-programming experts have potentially innovative ideas yet

lack the proper means to realize them. Productivity and effectiveness of expert engineers is compromised by a lack of modern programming technologies and practices. Languages used for development by experts today are uninteresting for new programmers to learn and infrequently taught at an introductory level in institutions around the world. Without support for high-level programming languages that are appealing and usable by a less technical audience, while not compromising performance and resource utilization needed by experts, the realization of compelling applications cannot scale to meet the growing demand for creation.

1.1 Thesis Question

The key question we explore in this dissertation can be stated as follows:

> *Can a development environment be designed and implemented to facilitate development of efficient software for a wide range of resource-constrained microcontroller-based device applications by a wide range of developers?*

Our research takes a programming language approach to answer this question. We design, implement and evaluate a domain specific language for resource constrained devices that is usable by experts in the field and accessible to relative novices. We select features from existing high-level programming languages that have successfully supported a broad range of application development by developers with diverse technical skills. We address the specific difficulties present in existing languages and software

development practices present in our domain. Namely, support for high-level language constructs that enable abstraction, software reuse, portability across hardware architectures, variability in software for differing hardware configurations and availability of resources, and efficient support for concurrency.

This dissertation contributes a new programming language for resource constrained microcontrollers called Em. Our language is for application development on resource constrained microcontrollers and incorporates constructs that enable development practices present in modern high-level programming languages. To achieve a degree of efficiency available using existing languages in this domain, code written in Em is translated into portable C. We then utilize existing compilers for diverse microcontroller targets and harness their mature optimization capabilities to achieve resource footprints comparable to extant systems.

We design and implement a number of different features and tools that together, make it easier to develop software and complex applications for microcontroller-based systems that fit and execute efficiently within the severe resource constraints of these devices. In particular, we employ constructs for modularity and the separation of concerns through encapsulation and information hiding, which are popular and effective in high-level languages. We implement the proxy design pattern as a language construct that enables a separation of interface from implementation, adding abstraction capabilities. Our use of proxies and interfaces also enables software support for differing

implementations of functionality that vary due to hardware specifics or execution environment constraints. We support systematic code reuse though interface inheritance, templates that help sidestep copy-and-paste reuse, and by enabling the authoring of adaptable and configurable software modules. We add further support for variability and configurability through the holistic integration of a build-time execution context that supports introspection, dynamic memory allocation and general purpose computation that executes on the resource rich build host. We introduce configuration parameters and host functions that enable module configuration along with opportunities to offload computation from the target device onto the build-host. To support portability and distributability we introduce a construct called a *composite* and support a packaging construct similar to that present in Java. Finally, our language supports the use of legacy C content in order to leverage existing software and ease developer's transition to the language.

We additionally contribute the design and implementation in Em of key non-language support features for efficient application development. Specifically, a hardware abstraction layer to enable the creation of software independent of low-level hardware specific details and a reactive model for concurrency that is fitting to a resource constrained environment with potentially strict power consumption requirements.

In this dissertation we furthermore contribute an empirical analysis of our language implementation, a demonstration and evaluation of real applications in Em, and a user

study. We show how software reusability, variability, configurability and portability is achievable. We find that we can effectively achieve a portable and reusable implementation of software supporting hardware functionality such as a device driver. We show that we can also achieve reuse and portability of software functionality which depends on, but does not support hardware. The code is configurable for differing amounts of runtime resources and supports variable implementations of hardware functionality and software policies without major modifications. Through an empirical evaluation of applications containing building block functionality we show that we can achieve equivalent and better resource utilization as compared to existing systems. Lastly, through teaching an introductory course on embedded systems programming at a university level, we conduct a user study on the ability for non-embedded systems experts and relative novice programmers to learn Em and implement non-trivial applications. We discuss the results of surveys conducted and articulate several applications created by students.

In summary, with this dissertation, we contribute a new high-level, general purpose, embedded systems programming language for development of highly resource constrained microcontroller applications, support for runtime concurrency and hardware abstraction, an empirical analysis of our language design and implementation, a demonstration and evaluation of real applications written in Em, and a user study with non-embedded system engineers. We find that both expert developers and less experi-

enced programmers are capable of developing non-trivial applications with equivalent or better resource utilization as compared to C and existing systems.

1.2 Dissertation Organization

We organize this dissertation as follows. In chapter 2 we first provide background information, discuss terminology related to resource constrained systems, application development, and define the domain within which our research applies. In chapter 3 we discuss current state-of-the-art systems used to develop device applications and their limitations. We look at tools, including high-level graphical systems, that are used by experts and those accessible to and commonly used by non-engineers and relative novices.

In chapters 4 and 5 we describe the design and implementation of a programming language we contribute to address our thesis question. We present the design of the language features necessary to support the development of applications for resource constrained systems by a broader audience of people. We present the implementation of the language and compare and contrast its details with extant systems closely related to our domain.

Following the chapters on language support we present in chapters 6 and 7 additional non-language features important to support application development in our do-

main. The design of a hardware abstraction layer, a runtime concurrency model, and development environment features for productivity are discussed, followed by discussion of their implementation. We additionally compare and contrast our design and implementation with other closely related systems.

In chapter 8 we present an empirical evaluation and demonstration of the efficacy of our design choices and implementation. Chapter 9 summarizes our contributions and discusses future research directions.

Chapter 2

Background

The goal of this dissertation is to bring high-level programming language constructs and modern software development practices that are found in resource rich systems to the domain of highly resource constrained embedded systems. Through a programming language approach, we seek to enable the existence of an ecosystem of modular, reusable and portable software in systems that tightly couple hardware and software functionality. Furthermore, we are interested in providing support for developers to adapt their software for this domain to various hardware configurations and to differences in runtime resources with minimal effort. In addition, our goal is to enable developers to extend their software with support for new hardware without significant impact on existing code. The result, we believe, will enable people with a wider range of backgrounds and levels of expertise, and thus a greater number of people, to create applications with resource constrained embedded systems.

Before we describe how we have achieved these goals, we provide background related to and in support of our methods. In particular, we define what we refer to as resource constrained embedded systems. We discuss the state of the art of software development for both resource constrained and resource rich environments, and we articulate features of the latter that can benefit the former. We then define in detail the domain that we target with our work.

2.1 Resource Constrained Systems

Resource constrained embedded systems, sometimes referred to simply as resource constrained systems, are perhaps surprisingly, the most prevalent form of computer system today. 98% of all processors (central processing units (CPUs)) manufactured globally each year are embedded. More than 55% of these processors are 8-bit resource constrained microcontrollers[10]. These devices take many forms and are ubiquitous in our lives. They are often embedded into the products they enable and serve as the primary interface between our physical world and the digital one. Some common applications for these systems are washing machines, microwaves, remote controls, video game controllers, computer peripherals such as mice and drawing tablets, printers and scanners, electronic toys, musical keyboards, radio controlled vehicles, wristwatches, parking meters, car stereos and engine control units, car and garage door openers, med-

ical devices such as pacemakers and blood glucose monitors, thermostats and home security systems, agricultural watering and monitoring systems, industrial manufacturing equipment, and many more.

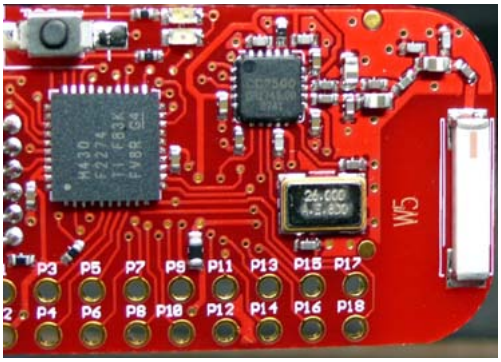

Figure 2.1: A printed circuit board with microcontroller.

A resource constrained system is comprised of a programmable microcontroller unit (MCU) with limited amounts of integrated program and data memories, application specific input and output devices, power supply and electronic connectors. The physical components of the system are arranged in a specific configuration on a printed circuit board (PCB) (e.g. see Figure 2.1) which is embedded within an enclosure or integrated into a larger structure (e.g. see Figure 2.2). Software is programmed into the MCU that drives the hardware peripherals and provides the device's application functionality. Each system is designed to perform a small number of domain specific functions as required by the application they target. Moreover, there is a high-level of integration between the hardware and software functionality.

Figure 2.2: A open device showing integrated PCB.

The term *resource constrained* herein refers to the limited amounts of on-chip memories of the microcontroller and stringent low-power consumption requirements of the system as a whole. Data memory or on-chip RAM ranges from hundreds of *bytes* to tens of *kilobytes*. On-chip program memory, where the microcontroller software resides, sometimes referred to as ROM or flash memory, is typically less constrained than RAM. The microcontrollers we target contain *tens to a few hundred kilobytes* of program memory. Energy is also a constrained resource in many of these devices since some applications of these systems require autonomous and/or extended operation (months to years) and utilize low power sources such as batteries and solar cells.

Along with the large number and diversity of applications of resource constrained systems, there exists an even larger and more diverse array of hardware components for them. Dozens of microcontroller manufacturers exist today each producing numerous

families of microcontrollers some with hundreds of devices with specifications differing in memory quantities, integrated peripherals, processing speed and power consumption. Every year, manufacturers advance their product offering with even more advanced and capable hardware. Hundreds of manufacturers that produce components such as sensors, actuators, displays, power supplies, and application specific integrated circuits also exist. Accelerometers, gyroscopes, pressure sensors, altimeters, thermistors, solenoids, motors, buzzers, LEDs, LCD displays, OLED displays, mp3 encoders and decoders, analog to digital converters, digital to analog converters, and power regulators, inverters and converters are some examples of the diversity. The proliferation and diversity of these components grows considerably each year.

Resource constrained systems inherently interface with and are embedded into the physical world and have different requirements than resource rich systems. Sensing and responding to physical stimuli from humans, the environment or other machines, resource constrained systems have strict requirements for real-time, low latency response to external stimuli unlike most resource rich systems. Additionally, since energy is a constrained resource, and since power consumption is directly related to CPU utilization, RAM and peripherals, many applications have requirements for periodic, limited data processing that occurs in response to stimuli as opposed to the continuously running, heavy processing requirements of resource rich systems. Finally, since resource constrained systems are embedded into other objects and are interacted with in many

ways, they require smaller, more compact and variable form-factors. This is also in contrast to resource rich systems that often consist of a box to house the system, a keyboard, a mouse, and monitor.

All resource constrained systems inherently have a high degree of variability in components, their physical interconnection and their functional configuration. Applications in different domains require microcontrollers with different kinds of integrated peripherals, processing speeds and power consumption. Additionally, several options for the input/output devices that support application functionality exist with differences in the way they interface to the microcontroller, size, cost and other characteristics. Finally, depending on the requirements and characteristics of their application, each component may need to be configured in a specific manner. For example, the ubiquitous car door opener requires a microcontroller and a wireless radio. The microcontroller for this application must be cheap, small, power efficient and need not have much processing power. The radio for the application must be power efficient, configurable and easily connected to the microcontroller. Depending on the communication interfaces available on the microcontroller (e.g. SPI, I2C, etc.) and the interface available on the radio, their interconnection may differ; manufacturers often offer devices with identical functionality with differing interfaces. Since the door opener must be powered from batteries and since the wireless signal should not interfere with other makes of

door openers, the transmission power and operational frequency of the radio must be configurable in an application specific manner.

While many applications already exist for resource constrained systems, their programmability, compact form factor, low cost, and the diversity of components available for them present many opportunities for new and innovative applications. The programmable nature of microcontrollers allow applications to be created for almost every domain imaginable and those that are yet to be imagined. The continual miniaturization of electronic components and circuit boards is enabling these systems to fit into the smallest of spaces, from forks and knives to tire valves and heart valves. Their relatively low cost makes the use of resource constrained systems economical in places where previously either none were used, or were used more sparsely. For example, from 2002 to 2004 the number of microcontrollers used in mid-range automobiles almost doubled [9], an indicator of decreasing costs and economic viability of their use. Finally, the sheer number of configurations of components and microcontrollers that exist can lead to innovative new applications and new application domains.

2.2 Application Development

The physical nature and use of resource constrained systems greatly affects the software development for device applications. We refer to a device application as the com-

bination of hardware and software that collectively realize the system's purpose. The development of device application software shares similarities with software developed for resource rich systems. Unfortunately, the software development technologies and processes available for resource rich systems greatly outpaces those available for resource constrained systems.

In this section we present the state of the art of software development technologies and processes available in resource constrained systems. We first, however, present aspects of the state of the art in application development for resource rich systems. We do so to draw a contrast between what is available for these two classes of systems. We also identify the technologies available from tools and processes in the resource rich domain that are lacking for the resource constrained domain. We also discuss the potential impact that such features have should they be available for device application development.

2.2.1 Development for Resource Rich Systems

Applications developed on resource rich systems rarely have to consider low-level, platform specific, details because of the presence of an operating system. The operating system manages the operation of hardware devices, provides support for concurrency, arbitrates access to resources and hides the low-level, platform specific implementation details of such processes from programmers. The very presence of operating systems

removes the need for the programmer to implement the features they offer in each of their applications. With standard interfaces to OS facilities, programmers are also able to write software usable on different operating systems without the need to even consider hardware. Operating systems have thus enabled larger numbers of programmers with more diverse skill sets to develop applications. Today, the average software developer has little, if any, knowledge of the implementation details of how their applications interact with and make use of the hardware, concurrency mechanisms and other services they depend on.

High level languages and development tools that allow uniform development across application domains and execution environments exist for resource rich systems. For more than four years now, object oriented programming languages are the most prevalently used languages and eight of the ten currently most popular high-level languages support the object-oriented programming model [83]. Irrespective of application domain - finance, ecommerce, health, agriculture, entertainment and others - these high level languages are being used for development. Some of these languages are compiled and others are interpreted. In both cases, the development tools - languages, compilers, debuggers and development environments - that support them operate on and can target the majority of resource rich platforms. For example, versions of the GNU compiler collection (GCC) [27] exist for Linux, Windows and Macintosh operating systems, support multiple object-oriented languages, and can build software for just about any

resource rich system. Also, Python interpreters [68] also exist for all the above mentioned operating systems on many different hardware platforms. Finally, development environments such as Eclipse [22] also function uniformly across operating systems and hardware platforms and support development in a number of high-level and object oriented languages.

The success of high-level languages comes from the collection of formal constructs they support. Support for modularity and the separation of concerns is one of the tenets of high-level languages and is enabled through encapsulation and information hiding. The decoupling of a modules functionality, or interface, from the implementation of that functionality is another important aspect of high-level languages. This capability is supported by language constructs such as interfaces along with polymorphic type capabilities. Finally, features such as garbage collection, dynamic type systems and succinct syntax also contribute to the success of today's high-level languages.

Modularity, or the separation of concerns, has enabled abstraction in software and more complex systems to be developed. By grouping related functionality and data into modular units and enforcing the private and public access to them, software modules become treatable and usable like abstract objects. Once a software module is developed, a programmer need only concern themselves with its specified functionality, which is often an abstract representation of a physical object or process. This has enabled programmers to develop applications by breaking down complex problems into

independently solvable units that are later combined into complete solutions. Different developers can create parts of applications that others will use without concern for implementation details, only functional purpose.

The decoupling of software modules' interface from their implementation has been key to supporting variability and change in software systems in addition to aiding in software abstractions. The notion of an interface, or simply the specification of functionality without provision of an implementation, allows programmers to abstractly define the behaviors of software modules. Implementations of those interfaces can vary while all modules implementing a specified interface can be assumed to provide the same functionality. A type system that supports polymorphism permits the development of code in which different modules that share a similar interface to take on different behaviors at different times. This enables change and variability to take place in software systems with minimal impact on existing software. New implementations for modules already in use in a system can be changed or even replaced without requiring change to software that depends on that functionality. As application or system requirements change, software systems are capable of adapting with minimal development overhead. When the change or variability is required to take place at runtime, a dynamic polymorphic type system is used and many of today's high-level languages support such type systems.

Dynamic features of high-level languages such as runtime memory allocation, garbage collection and runtime polymorphism are features that promote reliability and productivity in development. By providing capabilities for allocation and automatic disposal of data types at runtime has freed programmers from the error prone process of managing memory usage in their applications. As a result, programs have become more reliable, less prone to memory mishandling errors, and programmers are more productive. Additionally, the flexible and dynamic type systems offered by some languages such as Python, Ruby, and others, and supported by rich runtime systems, has relieved the programmer from arduous specification and restricted usage of types. All of these dynamic capabilities of high-level languages remove time-consuming tasks of development from the programmer and build them into the language runtimes, increasing developer productivity and application reliability in the process.

The combined constructs and capabilities found in today's high-level languages have evolved a development process that subsists on software reuse, portability and distribution. Since today's languages and development tools transcend application domains and are available on numerous platforms, software reuse is key for rapid, effective and reliable software development. Reusable elements of software that transcend any one languages, also known as design patterns, have been enabled by high-level languages and are also prevalent in software engineering practices. With the support of operating systems, many high-level languages have also overcome previously limiting

hardware boundaries. Software reuse across platforms, also known as software portability, has also been a focus of development productivity in software engineering. With the connectivity that the Internet provides us today, the distribution of software that functions across hardware platforms is essential. Many software libraries, frameworks, code fragments and complete applications are available for download and reuse across operating systems and hardware platforms.

The availability of large amounts of reusable, portable software content in high-level programming languages has enabled a wider audience of people, even novice programmers, to create applications. Today we see non-programmers using Javascript to customize functionality of web-pages, novice programmers using web toolkits like Ruby on Rails [73] to create non-trivial web applications, and generally less experienced programmers reusing domain-specific content to develop powerful tools for their fields of expertise. Leveraging the efforts of more experienced programmers through straightforward reuse of their code, many more people are able to create functional content. Through further learning, some are becoming more experienced programmers while others simply continue to use other's efforts. In both scenarios, the audience of people developing software content has widened dramatically as a response to the availability of high-level languages and reusable, portable content in resource rich environments.

2.2.2 Development for Resource Constrained Systems

General purpose operating systems that manage hardware resources and abstract away low-level device details are unusable in the resource constrained systems we are concerned with. Due to limited memories, requirements for low-latency response to external stimuli and limited power consumption, general purpose operating systems are not used. Instead, developers of applications for resource constrained systems must implement the facilities found in an operating system that they require along with their application logic. Device drivers, mechanisms for concurrency and memory management facilities are often the responsibility of the application developer to implement. The average developer for resource constrained software must therefore assume all the roles of device driver programmer, operating system implementer and application developer.

Since developers must implement both low-level device specific functionality and higher level application logic, the two are often interleaved with little separation between concerns. Counter principles of good software engineering practice, which advocate for a separation of concern between functionally distinct parts of an application, it is common today to see the accessing of device specific registers, to drive a hardware peripheral for example, interleaved with application specific logic. From applications already deployed in the field to introductory tutorials for the novice microcontroller programmer, the lack of separation of concerns is widespread.

Up until 2004 [10], the most prevalently used programming languages for resource constrained systems software development did not formally support high-level constructs for modularity and abstraction. Assembly language and C were the most widely used languages until the use of C++ surpassed assembly. While C++ does formally support constructs for modularity and abstraction, many of the language's high-level features, such as interfaces, virtual functions and abstract classes, are not usable in highly resource constrained systems. The memory overhead of virtual function tables and the runtime overhead of resolving pointer indirections, make these languages features compromise cycle accurate operations and therefore unusable.

In addition to a general lack of support for modularity and abstraction, the support for reusable and portable software is largely lacking. The way most resource constrained software is written, with an interleaving of low-level code and high-level application functionality, there is generally little code that is reusable across applications. The lack of formal constructs for encapsulation in the language leaves it up to the developer's programming discipline to write code such that it is reusable. Even if code is written in a manner reusable by the programmer who wrote it, differences in programming style and organization makes reuse of that code by others more complicated. Moreover, since device applications are closely integrated with the hardware they utilize, the standard practice is to write software for a specific set of hardware, fixed in a specific configuration. This makes software, as its written today, even less reusable

and in no way portable. Complicating portability further is the fact that some microcontroller manufacturers augment their C compilers with device-specific extensions. While meant to make some device capabilities easier to use, such extensions make code that utilizes them non-portable. In general, portable software continues to be elusive in the domain of resource constrained software.

The practice of software reuse that does exist today is largely a copy-paste-modify approach. Without language support for modularity and the lack of practice of separation of concerns in resource constrained software, developers are left with error prone copy-paste-modify approaches to software reuse. Microcontroller manufacturers that provide example code and sample applications for using their devices, often do not format it such that it can be reused directly. Developers must read and understand the code, find the parts that are relevant to their need, extract it from the sample and figure out how to integrate it, along with any dependencies it may have, into their own implementations. It is only with the utmost care and discipline from an experienced embedded systems programmer that software for resource constrained system exists and is usable without the copy-past-modify approach.

The inherent variability in hardware components, their interconnectivity and configuration implies that resource constrained software must be adaptable to change and variability. However, the standard method to support change and variability is through programming with extensive use of C preprocessor macros. Directives such as `#defines`

are used to define symbolic names for device registers and other hardware parameters that may need to change. Conditional directives such as `#ifdef/#endif` are interspersed and interleaved into programs such that blocks of code can be conditionally compiled into or left out of an application. While achieving the desired effect of adapting an application for varied hardware configuration or differences in execution resources, the use of both kinds of directives causes code to be more difficult to read, understand, modify and maintain. Since often times the conditional directives are defined in the language of a build tool such as Make, today's process for adapting to change and variability entails understanding the build tool language (e.g. Make), proficiency with the syntax and semantics of the C preprocessor, and the programming language itself (C or C++).

The tools available for microcontroller application development are highly fragmented, largely as a result of microcontroller vendor's business practices. Every microcontroller vendor has traditionally tried to attract developers to their microcontroller products not only by competing hardware features but also through software support for developing applications. Compilers optimized for their hardware, development environments that make programming easier, sample code for driving peripherals and even sample applications are made available. To be certain they don't lose customers, those providing the tools - the microcontroller vendor or closely associated third parties - make no effort towards compatibility with other vendor's devices. Compilers are

extended with device specific features such that code written for that device cannot be used in another; development environments only recognize and operate with a specific vendors microcontrollers; sample code and applications are written with device specific features and implementation details tightly interwoven. As a result, dozens of development environments, compilers, and tools for downloading code onto devices exist each for a specific vendor's device. An expert developer working with several devices often has several IDE's and a handful of compilers installed on their workstation and on their desk several hardware download utilities can be found, one for each vendor's microcontroller.

2.2.3 Development by a Wider Audience

In recent years, the cost of microcontroller development kits and peripheral components has decreased dramatically while their availability has increased enabling many, for whom device application development was previously out of reach, to embrace it. A microcontroller development kit today costs on average $50USD and is available though various online retailers, internationally. The same kit, less than ten years ago, would have cost hundreds of dollars. To customize the development board, accelerometers, gyroscopes, LCD and LED displays, motors and many other sensors, actuators and miscellaneous components are available. Retailers such as Sparkfun Electronics [76], Adafruit Industries [1], Jameco [36] and others sell both development boards and com-

ponents at prices within reach for hobbyists, students, designers, artist and others. To make these components even more usable by such people, these retailers often design small hardware modules containing the main component that easily interface with popular development boards. Due to the convenience of such offerings, even expert developers purchase devices from such retailers.

The cost of software development tools have also decreased dramatically in recent years. Development tools for microcontroller programming such as compilers, debuggers, device programmers and integrated development environments used to cost hundreds to thousands of dollars just a few years ago making development accessible only to businesses and research institutions that could afford their cost. Today, open-source tools exist for free, downloadable from the Internet instantly, and support microcontroller development for devices from numerous manufacturers (e.g. GCC for ARM, AVR, MSP430). In addition to development environments made available by microcontroller vendors, several simpler, open-source solutions have recently become increasingly popular.

The decrease in cost and increase in availability of software development tools and hardware components has broadened the scope of who is trying to develop microcontroller applications. For the same cost of a textbook, students can now purchase a development board and several additional components in order to learn how to realize a device application. In universities around the world, there are a growing number of

courses being offered in digital arts, electrical engineering and computer science that aim to teach students how to develop microcontroller applications. In high schools, predominantly in the USA and Europe there are growing numbers of robotics teams and workshops in computational arts and crafts that utilize the same microcontroller hardware and tools to teach how to make digitally enabled artifacts. Outside of an academic context, and as seen in newspaper articles [6, 43], television shows [18], popular blogs [49, 28], and major events [48, 15, 59], there is a constant stream of creative projects created by non-embedded systems engineers and relatively less experienced programmers. Clearly, the interest in creating custom device applications, and those attempting to create them, has expanded well beyond the traditional experts of resource constrained application programmers.

2.3 Definition of Domain

Our work specifically targets the domain of software development for a wide spectrum of applications utilizing resource constrained microcontroller-based systems developed by an audience of people who have a diversity of programming skills, from novice to expert. The microcontrollers used in device applications of our domain are highly constrained and consist of a few hundred bytes to few tens of kilobytes of RAM. They have a maximum of a few hundred kilobytes of program memory and are unable

to utilize currently available off-the-shelf general purpose operating systems such as Linux. There is, and continues to be, a constant flux of hardware in our domain and an even greater variation in the combinatorial possibilities of that hardware. We do not define the intended applications of these resource constrained devices, however, the general class of applications are those that require real-time low latency response to external stimuli, have low data processing requirements, and potentially severe power consumption constraints. The audience of developers includes expert embedded systems engineers well versed in C and assembly, existing programmers with little or no knowledge of embedded systems programming, novice programmers and even non-programmers that are familiar enough with basic scripting languages such as Javascript and are capable of reusing existing content to customize existing applications for their needs.

2.4 Summary

In summary, resource constrained systems are widespread and have unique functional requirements that differ from those for resource rich systems. The ability to respond to external stimuli in real-time with low latency is more crucial in resource constrained systems than heavy data processing. The presence of very low quantities of data and program memory as well as potentially strict power consumption requirements

heavily constrain the systems of the domain we are interested in. The lack of operating systems to provide concurrency, resource management and to abstract away low-level hardware details complicates software development.

Despite their differences, resource rich and resource constrained systems share similarities in the development of software. Applications in both domains require programming in a language that will enable rapid and efficient development of robust software. Adaptability to changing application requirements and variability in hardware with minimal impact on existing software is essential for both types of systems. Reuse of software, its portability and ease of distributability is essential for enabling productivity in the development of applications for both kinds of systems. However, the tools, technologies and processes available for developing applications for resource rich systems greatly outpaces what is available in our defined domain. Given the shared similarities in application development for both domains, the languages, tools and processes for application development in our domain could greatly benefit from features found in the domain of application development for resource rich systems.

Finally, the availability of affordable hardware and free development tools has helped many people beyond domain experts take their first steps towards development of device applications. However, the software development tools and processes for those applications is a stumbling block for many. Downloading and installing development tools from the Internet is easy. Purchasing hardware and components for a custom

application is simple and affordable. Connecting the parts together is relatively straightforward. Programming device drivers to enable the hardware, implementing operating system facilities, and then developing the application functionality is complex for people of all levels of expertise. Relieving the lack of readily available, reusable software that can be used across device application hardware without compromising critical resources can help expand the number and diversity of people creating compelling and potentially innovative applications.

Chapter 3

Related Work

Our work takes a programming language approach to advancing the state of the art of application development using resource constrained microcontrollers. Our intent is to support a broad audience of application developers that are able to rapidly create robust applications utilizing the diversity of hardware that is readily available today and will be available tomorrow. Language support for modularity, software reuse and portability are central in our work along with flexible application configurability to support variability in hardware and software. Without compromising efficient usage of critical resources, our work aims to support both novices developers and experts embedded systems engineers.

In this chapter we discuss work and systems related to our research. Work in both language and non-language support for embedded systems application development is discussed. Technology for providing high-level language constructs with efficient implementations is related to our work and several systems are discussed. Additionally,

we discuss systems targeting development of microcontroller applications by novices. High level programming environments have enabled domain experts with less programming experience develop complex applications and we discuss such related systems targeted at both engineers and non-engineers. The broad audiences of such tools are a point of related interest to our work. While we discuss work related to our efforts in this section, we empirically compare and contrast some of these systems in further individual chapters.

3.1 Language Support for Modularity

In this section we describe existing systems that provide language support for modularity in software for resource constrained systems. We also describe systems that have attempted to provide high-level language constructs without compromising scarce resources. The systems discussed have been used by members of highly technical communities in academia and industry. In some instances the development has been intended for a specific application domain such as wireless sensor networks, while others attempt to provide more general facilities.

3.1.1 NesC & TinyOS

The nesC programming language [26] and TinyOS operating system [81] is an attractive platform for sensor network application development using microcontroller-based devices. The nesC language is a dialect of C that provides a structured, component-oriented programming model for reusable software modules. TinyOS is a collection of software components integrated into an application that provides support for a specific set of hardware devices, task concurrency via reactive runtime and scheduler, and other domain-specific functionality.

The component model of nesC uses a strictly local namespace for modules [82], completely decoupling module implementations from one another. Modules communicate through narrow interfaces that define fine grained functionality. NesC requires configurations that ”wire” together users and providers of interfaces in order to resolve the local references of modules to concrete implementations. Applications are implemented in nesC by providing configurations that wire together a set of modules used in the application.

The rational for nesC’s model is that component implementations remain stable over time, while configurations of components into applications varies from application to application. Configurations therefore are meant to enable flexible component reuse. Additionally, the use of narrow interfaces as the method of communication between modules is meant to enable a fine granularity of component reuse saving micro-

controller resources [82, 4] by not including in an application functionality that is not necessary.

While flexible component reuse and efficient use of scarce hardware resources are goals shared by Em, the pattern of usage in nesC's component model for reusing available components and creating applications is fundamentally different than the one we aim for. Requiring configurations and component wiring to utilize modules has not shown itself to be a straightforward or intuitive approach. Furthermore, time has shown that the black-box reuse of TinyOS has been largely unrealized [4] and it is widely acknowledged that using components and creating applications through configurations and wirings is a complex, intellectually challenging process [82]. The complexity associated with nesC's programming model restricts its use to a highly specialized expert community and makes it not amenable to the intended audience of Em which has a wide range of skills, from expert to novice.

3.1.2 RTSC

The RTSC/XDC [71, 72] toolset provides component oriented microcontroller programming facilities. The toolset provides an interface description language (IDL) that allows for the specification of modular units of code. The implementation of the interface is carried out in C with specific syntactical requirements in the naming of functions and variables such that they match the interface specification. The toolset additionally

provides the ability to script build-time functionality into modules with a scripting language similar to, yet different from, the IDL. The three languages together enable the support for modularity. The complexity of learning and using three different languages to realize software modularity is non-trivial. Existing embedded engineers often lack the high-level language skills to utilize such software based infrastructure. Novices inexperienced in either area face significant challenges in utilizing the toolset. The build-time configuration capabilities available are similar to those incorporated in our work, however RTSC/XDC utilize a language different from the implementation language for such functionality, unlike our design. While the toolset has been shown to work on a highly resource constrained device, large parts of the infrastructure were omitted and only one device from the manufacturer of the toolset was shown to be supported.

Since RTSC/XDC lacks any runtime services for concurrency, the DSP/BIOS [19] system was developed in RTSC/XDC and provides runtime services for concurrency. The DSP/BIOS real-time operating system from Texas Instruments provides sophisticated runtime support and a partial device driver model for applications. However, this system has been designed for devices requiring significantly more resources than those available in our domain. A resource constrained microcontroller has been targeted by this system, however it has not been shown that multiple platforms from differing vendors have been supported.

3.1.3 Other Systems

Embedded Java[74] devices such as the Javelin stamp [37] and JStamp/TINI processors [38] use Java to provide an object-oriented programming model for embedded processors. The way some systems use Java (JVM vs Java to native code compilation) and due to a lacking device abstraction model, code is not reusable across systems. Runtime support for concurrency is not uniformly supported in these devices either, with some providing elaborate real-time Java support and others lacking support all together. Systems offering real-time Java support require more resources than available in our target processors.

Jiazzi [53] adds explicit language constructs to Java for organization of code in terms of reusable software components. The authors identify key properties that are required by component systems to work with OO languages for large-scale modular construction of programs. Concepts of components and reusability from this past work are applicable to our work, however, this work does not target severely resource-constrained devices.

ExoVM [84] is a Java virtual machine and language design that together target embedded systems development. ExoVM provides analysis for computing reachable code and data in Java applications. Subramonian et. al discuss dynamic and static configuration mechanisms in component middleware for distributed real-time and embedded systems [77]. Both of these works provide insight into optimization techniques applica-

ble to the build process in Em. However, they target more resource-rich environments than our domain of resource-constrained embedded systems.

3.2 Microcontroller Development Support for Non-Experts

The popularity of microcontroller programming among hobbyists and non-experts has grown slowly though the 1990's and much more rapidly in the last five years. Early tools such as the BASIC Stamp carried much of the community that existed in the 1990's and more recent tools such as Wiring and Arduino are now leading a quickly expanding audience of people into device application development.

3.2.1 BASIC Stamp

The BASIC Stamp is a microcontroller break-out board that hosts a BASIC interpreter in ROM and has been used by hobbyists and amateurs since the early 1990's [62]. Compared to non-hobbyist development kits at the time, the BASIC stamp cost half the price and contributed to is popularity. The programming language for the device was BASIC which was relatively easier to use than C which, at the time, was often extended with device specific intrinsic functions. The availability of the BASIC Stamp helped extend the reach of microcontroller programming to less expert developers.

Despite being programmed in BASIC, the code written suffers from the same problems articulated above. Expressions directly manipulating device registers are interleaved with application functionality. There are no constructs for modularity in the language and code reuse is achieved through a copy-paste-modify approach. The language itself has been extended with functions for device specific peripherals. The development environment distributed with the device resembles those in use by experts and difficult for beginners to understand. Furthermore, a limiting aspect of the development board is the runtime interpretation of the application. Already lacking in resources, the additional CPU cycles required to interpret instructions limits the application domains the device can be used in. Time-critical applications with sub-millisecond response times are difficult to achieve. Furthermore, communication with external devices through serial interfaces is highly limited in bandwidth.

3.2.2 Wiring and Arduino

The Arduino platform [5], which is based on Wiring [90], is currently the most popular development platform for non-expert developers of resource constrained device applications. Arduino has lowered the barrier to entry for microcontroller programming by creating hardware that easily interfaces to a simple development environment and presenting users with an uncomplicated API to basic microcontroller capabilities. Due

to Arduino's initial ease of use and open-source offering, it has grown a large and vibrant user community that creates and openly shares a wide range of applications.

The API provides functions to set input/output directions for the microcontroller pins and to read/write their state. A simple, synchronous analog to digital converter read function, delay functions for millisecond and microsecond delays based on busy-wait loops, functions for reading and writing serial data from the UART and serial peripheral interface (SPI) port, and functions for attaching an interrupt handler triggered by a change of an input pin state exist.

The base Arduino API hides the low-level details of interacting with microcontroller registers to enable the functionality. Other features of the Arduino's microcontroller such as interfacing to the full set of interrupt sources, accessing peripheral state registers and configuring specific details of peripherals are not available. While Arduino's API does not abstract away its hardware, it does provide a simpler method of interacting with the fixed hardware that is present.

Beyond this API, the Arduino community has code supporting different devices such as LCD displays, sensors, and wireless radios. This code, however, is reused through a copy-paste-modify approach that is complex and highly error prone. Only advanced users with knowledge of low-level hardware details and sophisticated programming skills manage to take full advantage of what already exists. Because, there is no framework in support of modularity and that facilitates reuse of contributed code,

many less-experienced users struggle to make even basic applications fully functional (we have experienced this first-hand as we have attempted to help such users (students) in the classroom).

Reacting to concurrent environmental stimuli in a timely manner is an essential requirement for many applications in our domain. For example, responding to human input and reacting with display output, controlling sensors, and other activities in this domain, are time-sensitive operations. Moreover, performing these operations simultaneously requires support for concurrency. Arduino provides no such support. Coordinating the execution of application code and servicing multiple sources of I/O is functionality left entirely up to the developer to implement, thus developers implement the same functionality independently and repeatedly.

Finally, while Arduino does support a narrow collection of hardware with differing configurations, there is no structured programming model with the proper abstractions and capabilities to quickly adapt code to new configurations. Adding support for a new device involves adding code to and modifying the programming environment. Furthermore, all Arduino code assumes execution on microcontrollers from a small set of Atmel's AVR [8] family of processors. Non-Atmel processor support is not available. This lack of the necessary abstractions to introduce support for new devices (of which there are vast diversity) makes it infeasible to write software for new platforms using Arduino.

3.3 High Level Programming Environments For Non-Programmers

Part of the intent of this dissertation is to enable a wider audience of non-programmers to develop resource constrained device applications. High level programming environments exist for non-programmers with knowledge or expertise in a particular domain. These environments are related to our work since they have considerable usage and have enabled non-programmers to develop custom content with high level graphical programming environments.

3.3.1 Graphical Dataflow Process Networks

Graphical dataflow process networks are a formally defined model of computation that describe a computational system in terms of a graph. A process network consists of nodes, programmed in a host programming language, that represent a quantum of computation and arcs, representing data pathways, that interconnect the nodes. Rules defining how nodes 'fire' based on availability of data dictate how a graph will execute. The semantics of how arcs can be used in addition to the firing rules of the nodes is called the network's coordination language and defines how a complete computation system, or program can be created.

Variations on the firing rules can cause the dataflow network to have behaviors and characteristics that have and continue to feed an entire field of research. For example, when processing nodes statically define the amount of data they consume and produce, also known as synchronous dataflow networks [45], an execution schedule for the network can be statically determined. This eliminates the complexity and overhead of runtime scheduling and ensures a graph configuration that is executable [44]. If special processing nodes are allowed to alter their data production and consumption rate, as in boolean and dynamic dataflow networks[14, 13], the network cannot be statically scheduled and a runtime scheduler is necessary. However, in these dynamic networks, flow control constructs such as if/then/else and do/while, become available for use. One model, therefore, can have a faster execution speed at the expense of lacking expressive constructs, while the other model requires more runtime resources and allows for more dynamic behavior.

The programming model of graphical dataflow networks is clear, concise, and intuitive, and depending on the choice of network properties, various programming constructs are available to the programmer. Additional constructs exist in these networks which helps reduce programming complexity. Nesting networks within nodes enables hierarchical construction of process networks and is similar to the constructs of encapsulation and data hiding.

Several environments for working with dataflow process networks exist, most notably is the Ptolemy project [65]. These environments however, are not suitable for our domain as they are used primarily for research modeling and simulation. While code generation for embedded devices exists on some level [91] and embedded systems projects have been developed with these systems [40, 89], they are targeted for use by researchers and highly specialized experts within the embedded systems domain.

Dataflow process networks are related to other graphical models of computation such as Petri nets [63] and Kahn process networks [39]. Statecharts [30] are also a graphical model to describe system behaviors. Each of these models differs in their domain of application, as well as the rules that dictate composition of a computation. Dataflow process networks are a generalized case of the Kahn process network.

3.3.2 Max/MSP and PureData

Max [66] and PureData (PD) [67] are two graphical programming environment that are popular with performers, composers, artists, hobbyists and students working with music and multimedia. These development environments facilitate the creation of real-time, interactive, audio, video, and control data processing software. Max is a commercial product while PD is open source and historically, the two programs come from a common origin and are conceptually similar [52].

Max and PD have a clear programming model that has been largely appealing to its users as is evidenced by the large following these applications have. Both environments embody a form of the graphical dataflow model of computation [46] which lends itself well to processing real-time data streams – an intuitive and effective model of computation for many cyber-physical applications. A processing graph is made up of objects, each of which is a component from a loadable C/C++ or Java library. The objects are interconnected with patch cords that represent data and event streams. Data sources bring audio, video, timing, and control data into the environment which then flows through the cords to drive the processing graph. Data and event sinks within the graph facilitate output of information from the environment.

The programming model of Max and PD has enabled large numbers of users of our domain to develop sophisticated software applications. However, neither Max nor PD is intended for development of embedded device applications as they inherently assume execution on resource rich environments using conventional operating systems. The majority of objects in these environments focus on digital signal processing of audio and video and execution on a resource constrained microcontroller would impede their function. Additionally, the environments rely entirely on the underlying operating system for the transferring of data between hardware devices and the software application. Thus, timely handling of data input and output is hard to guarantee and application delays or bottlenecks are addressed by using faster processors and more memory. In the

hardware systems of our domain, resources are inherently constrained in CPU power, memory size, and potentially battery supply. Most applications created in Max or PD will not function effectively (or even at all) in these environments.

In addition, some Max and PD applications are used only to generate or process control data instead of to manipulate audio and video. The interfaces available to these applications for input and output of control data is dependent on the operating system drivers and Max/PD objects available to access the hardware. Thus, these types of applications tend to be limited to using interfaces such as USB, Ethernet and serial ports. Adapting the Max and PD environments to new hardware is also challenging as both an operating system driver and a Max or PD object must be developed to do so. The complexity is problematic for our domain as frequently, new hardware must be added to a system and the environment must be flexibly adaptable to supporting it.

The Max and PD environments however, do facilitate expressibility by providing a rich set of programming constructs. Processing objects enable fine grained control over the flow of data through a network. Patcher objects in Max encapsulate entire networks and through the use if inlets and outlets in the encapsulating patcher, the network becomes a simple object. This construct allows for the reduction of complexity and abstraction of functionality in order to build sophisticated systems. Finally, processing objects in these environments are loadable libraries that can be reused on any platform the software is supported on. Also, networks made up of objects, called patches, can

be saved and reused as objects inside other patches. Max and PD therefore provide inherently modular and reusable functionality.

3.3.3 CLAM

The C++ Library for Audio and Music (CLAM) [3, 2], developed at the Music Technology Group of the University Pompeu Fabra, is a complete software framework for research and application development in the audio and music domain. CLAM uses the synchronous dataflow network model of computation. The host language for processing elements is C++ and an extensive object oriented hierarchy of elements exist which are available to extend and specialize. The coordination language of CLAM is either C++ or, alternatively, a simple network editor application exists which provides a visual environment for constructing and executing networks. More advanced users of CLAM may use C++ as their mode of development, however those knowledgeable in the audio and music domain that are not technically skilled in programming can still develop advanced applications through the visual development environment.

CLAM embodies most of the programming language constructs we have outlined as necessary for expressibility in our domain. Additionally, it implements a variation of the synchronous dataflow model of computation which allows its data and event streams to be combined in a processing graph [85] that can be scheduled statically, enabling efficient execution of the application. Due to CLAM's choice of dataflow

model it does not have the ability to alter or control the flow of data using if/then/else or do/while constructs. Moreover, CLAM is a platform for research in the audio and music domain and uses analysis and synthesis algorithms that make use of complex, advanced, computational techniques restricting its use on resource constrained systems. Finally, because of the varying formats of analyzed audio data, CLAM has at its core a flexible and highly dynamic type system, a useful feature for resource rich environments but prohibitive for execution on microcontrollers.

Other systems with design intent similar to CLAM are ChucK [87] and OpenSoundWorld [16]. ChucK is an audio programming language for real-time synthesis, composition, performance, and analysis. It differs from CLAM in its programming model, as ChucK doesn't use a dataflow processing model, and its capabilities to modify code dynamically. OpenSoundWorld is an extendible programming environment to process sound in response to real-time control input. Like CLAM it is based on a dataflow model and its applications can be programmed in C++ or using a visual editor. Unlike CLAM, however, OpenSoundWorld allows dynamic manipulation of the application at runtime.

3.3.4 Other Tools and Systems

Other high-level programming environments that can process digital signals exist such as LabView [42], Isadora [34], OpenMusic [12], and Quartz Composer [69].

These environments all use forms of dataflow programming models and differ in their intended applications. LabView is for digital signal processing for data acquisition and control systems. Isadora is primarily focused on video processing with only minimal support for audio. Open Music is focused on audio signal processing. Quartz Composer is focused on digital image and video processing although also has capabilities for audio analysis and manipulation. All these environments share the capability of handling control input from various sources to control data processing and none of them can currently execute on resource constrained microcontrollers.

3.4 Summary

In this chapter we present extant systems and tools which represent various aspects of the state of the art as related to our work. Systems with language support for modularity exist although they are targeted at specialized application domains or more resource rich devices and remain accessible only to highly trained researchers and engineers. These systems do not support a general purpose solution to modularity in resource constrained devices and either their programming model is challenging even for the experts that use them or they require mastery of several programming languages to employ them.

Other, high-level, object-oriented languages such as Java and C++, while appealing for their familiarity and rich constructs, demand more resources for their dynamic capabilities than available in our target devices and are therefore not suitable as a basis for a general solution. Other systems that demonstrate techniques for optimization in compilation and runtime environments inform mechanisms for resource conservation. However, these extant systems target more resource rich environments and would be non-trivial to modify for use in devices of our domain.

Platforms for resource constrained application development in use by a less technical audience today lower the barrier to entry for development, however, since they are based on the same extant languages and tools already in use, they ultimately suffer from similar problems found in expert tools. These environments do not provide language constructs for modularity, software reuse depends on a copy-paste-modify approach, only a narrow set of devices from a single manufacturer is supported, and it remains difficult for non-experts to take full advantage of functionality present in the hardware they support.

High-level programming environments for non-programmers have successfully supported large communities of developers. The extant systems, while useful in a broad range of application domains, assume execution in resource rich environments and cannot be augmented to effectively enable development in highly resource constrained en-

vironments. Their ability to support a broad community of users, however, does inform us of usage models and environment features necessary to do so.

Our work, described in the chapters ahead, develops the programming language support necessary for application development in resource constrained environments. We address shortcomings in extant systems by providing constructs for modularity, software reuse, variability, configurability and portability that does not compromise critical resources and is suitable for a wide range of devices and application domains. Furthermore, we strive in our design and implementation to make such language support accessible to and usable by a less-technical audience; goals which have not been successfully addressed by other systems for this domain.

Following the chapters on the design and implementation of our language support, we address additional development support for effective hardware abstraction, concurrency, and general application development. Such support is essential for expediting the development process and relieving the burden on developers to repeatedly write device drivers and operating system facilities to support their applications. In the final chapters of the dissertation, we provide an empirical evaluation of our work demonstrating our ability to support efficient development, relative to extant systems and technologies, that is usable on a diversity of hardware, in differing applications of significant complexity and by an audience with generally less technical expertise.

Chapter 4

Language Support for Application Development

The goal of our work is to advance the state of the art of resource-constrained microcontroller application development. Taking a programing language approach to our problem, we design and develop a domain-specific language that will support a broad range of programmers with diverse technical skills in the realization of a spectrum of device applications. The language itself must posses features that enable the realization of robust and complex software systems. Additionally, the language must support modern software development practices that depend on the existence of an ecosystem of software that is reusable and portable across application domains and hardware platforms.

In this chapter, we describe the language features we have selected for the design of a programming language specific to our stated domain. The features have been selected from existing high-level languages where they have proven to achieve objectives desir-

able in, yet lacking from, a language for resource constrained microcontrollers. We articulate both the objectives desired and the language features that have been chosen to meet those objectives. We also point out some languages features popularly found in high-level languages that have been explicitly left out of our design. To conclude the chapter, we contrast our selection of features with existing work in our domain.

4.1 Efficiency

A primary objective in our design is runtime efficiency. As discussed in the background section, many resource constrained applications interface with the physical world and require deterministic or cycle accurate response times to external stimuli. Assembly language and C are the most widely used languages in this domain precisely because they afford such requirements to be satisfied. Furthermore, the compiler and assembler technologies available today have benefited from decades of refinement towards the optimal use of the microcontroller's data and program memories.

Any new language to support resource constrained software development must not compromise the levels of efficiency and accuracy that can be achieved today using existing languages. To that extent, we have elected to use C as a portable assembly language. All the high-level constructs our language design introduces translate into efficient C code. Moreover, we do not introduce additional runtime behavior above

what exists in the C runtime. This decision enables us to achieve at least the level of execution efficiency expected and available today. Additionally, this decision allows us to take advantage of existing compiler technology for the numerous microcontroller targets that exist and leverage all the optimization capabilities they posses.

Traditionally, in our domain, runtime efficiency has come at the expense of application development efficiency and it is our objective that this not be the case in our design. The use of low-level assembly language may produce cycle accurate results, however, any such code must be rewritten in its entirety to function on a microcontroller with a differing architecture. This demands significant development resources; both time and money. While C, being higher level than assembly, can be used to produce portable code, it is used in practice as a more advanced assembly language with programmers interleaving access to low-level device registers with higher level application functionality. This practice stifles the reuse and portability of software and limits the efficiency and effectiveness of the development process.

A balance between resource and development efficiency has not been struck by existing languages in our domain, however, development efficiency is significantly more advanced in other domains. Development processes and the languages to support them have advanced and been refined for more than two decades in resource rich environments. While less critical of resource consumption, processes of software development and architecture, with formal constructs in languages to support them, have achieved

significantly higher levels of development efficiency. In the subsections that follow that describe the language constructs in our design, it is our intention throughout to preserve the resource efficiency and accuracy achievable in today's languages for resource constrained application development while supporting the effective development processes that exist elsewhere.

4.2 Readability and Writability

The developer community that we target with this work is significantly broader than the traditional embedded systems community. In addition to experts of resource constrained microcontroller programming, we seek to advance the capabilities of novice and non-programmers to create device applications. Our intention for this comes from the fact that thousands of non-technical people have in the last five years embraced the customization of technology for many applications that require an interface between the physical and digital world; people from arts, music, digital media, and many other non-engineering disciplines. Interestingly, members of the same audience we target have been successful in creating rich Internet applications in recent years using scripting languages such as Javascript, PHP, Python and Ruby. These languages appear to be readable, understandable and usable by an audience with less technical skill as well

as by experts. Beyond readability and writability, the style of the languages mentioned also seem to draw people to learn them.

While a less technical design decision, we feel the ability for novice and expert alike to be able to read, understand and write code written in our language is critical for its success. Several factors contribute to the readability and writability of code including the language's overall simplicity and its syntactic elements. To address the overall simplicity of the code we have chosen a style of programming that reads from top to bottom and does not incorporate any compile-time constructs that selectively include or exclude code that is present in the source file (e.g. `#define/#ifdef` preprocessor directives in C) or that alters the flow of execution of the code at runtime in an unstructured manner (e.g. the goto statements of C).

To address the readability of code in our design we have chosen a syntactic style that attempts to find a balance between what existing programmers are familiar with and what new and novice programmers have found success with in scripting languages. Specifically, the syntactic constructs for designating code blocks, control statements and general expressions found in C, C++ and Java have been selected since a tremendous amount of existing programmers, across application domains, can instantly read and understand them; we have not deviated from the semantics of these syntactic elements. We also believe that the syntactic elements of these languages helps in writing clean code which lends itself greatly to readability. To leverage the momentum of pop-

ularity of scripting languages, we also attempt to match the syntactic style and feel of scripting languages. Specifically, in our design we have relaxed the requirement of terminating lines with semicolons, have chosen variable and function declaration constructs found in Javascript, Python and Ruby, and have employed sparse syntax constructs such as not requiring parameter types to be respecified in function definitions.

A critical factor of enabling people to write their own applications is that they be able to read them. Beyond that however, to support the writability of applications we have attempted to select keywords that reflect their intent in the representation of control and data structures. The following section discusses these representations.

4.3 Modularity and Abstraction

Modularity is inherent in the physical design of device application hardware. Each chip on a circuit board serves a specific function whose precise implementation is largely unknown. The functionality of each component is enabled through interaction with the interface the device provides. The microcontroller software must then drive the devices it is interfaced with and coordinate the flow of data and commands between devices and higher-level application processes. In our design, we seek to have a software representation of the devices and subsystems that make up the device application.

Having formal support for modularity in the language helps establish a separation of concerns between distinct parts of the application while lending itself to structuring the software in a manner that reflects the hardware structure. Since hardware components and subsystems are often singular, we have chosen to have each software module be a singular entity. In our design, modules are not replicable through instantiation as objects are from classes in object oriented languages. Our decision for this relates to the fact that device applications in our domain have a tight integration between hardware and software.

Formal support for the separation of concerns is a cornerstone of high-level programming languages and brings about modularity. By grouping related functionality and data into modular units, or encapsulation, programmers can solve complex problems by breaking them down into smaller, more manageable units. The separation of concerns also entails removing, to the extent possible, dependencies between functional groups, or modules, that make up a program. These smaller modular units then come together and interact to solve the larger problem on hand. Teams of potentially geographically distributed programmers can develop large applications and each programmer can focus on a smaller problem at any one time, leading to more reliable implementations. When errors in applications are encountered, finding and fixing them is assisted by the separation of concerns since functionality is localized and interdependence between other application functionality has been minimized. As a testament

of its widespread use in modern high-level languages, formal constructs for modularity exist in nine out of ten of today's most commonly used languages [83].

Constructs for modularity and its benefit to software development are largely lacking from prevalently used languages in resource-constrained systems. Both assembly language and C lack the formal constructs that enable modularity. As such, developers are left to their own discipline of programming to bring about the advantages that modularity brings about. It is the case, as a result of this practice, that incompatibilities between programmer's styles of implementing modularity exist, making it difficult for developers to leverage the efforts of others in their own work. Furthermore, it is the exception, and not the rule that software is developed with an inherent separation of concerns. To leverage the well proven benefits of modularity, we find it necessary in our design to formally provide the language support to enable it.

In conjunction with encapsulation, information hiding enables modular units to be treated abstractly and brings about software abstraction. Information hiding is the enforcement of public or private access to data and functionality that has been encapsulated within a module. This enforcement allows programmers to interact with modules based on the public functionality they expose with no assumption or knowledge of their implementation. This abstraction of a module's implementation further reduces the details a programmer must be concerned about at any one time and allows them to focus on the functionality of the application as a whole. Furthermore, abstraction gives pro-

grammers the ability to reason with programmatic elements as concepts or ideas, aiding in the fluid translation of ideas into functional software systems.

In addition to information hiding, the construct of an interface aids in furthering the ability to abstract away implementation details. The notion of an interface, as present in Java and C#, specifies the functionality a module will possess without specifying any implementation. Software modules implement the interface by providing definitions for all the functions specified in it. Contractually, interfaces make it clear to the programmer what functionality a module possesses and entirely abstracts away and removes concern for implementation details. This abstraction furthers developer's ability to reason about complex software systems and effectively translate them into working implementations.

Formal enforcement of modularity and constructs that enable abstraction such as interfaces, are also largely lacking in the dominant languages used for resource-constrained microcontroller programming. While the use of C++ has introduced some modularity into resource-constrained software, the language generally lacks interfaces as present in Java and C#. While functionality similar to interfaces can be reaped from C++, the overhead of its usage makes it unsuitable for our domain. Because of the benefit of abstraction brought about by modularity and the use of interfaces, we have incorporated them both in the design of our language.

4.4 Variability and Configurability

A microcontroller-based device application is made up of both hardware and software. The hardware consists of a printed circuit board which hosts a microcontroller, a variety of peripherals needed by the application, supporting circuitry and connectors. The software, usually embedded into the microcontroller's on-chip flash memory, contains both drivers to enable and interact with the hardware as well as the higher-level application-specific logic. For each device that is created, something about the configuration of physical hardware or software of that device is different from others. For example, hardware components are wired to the microcontroller differently, quantities of microcontroller memories and their specific memory maps differ, the method of controlling peripherals differs and the trade off between requirements of time and space for applications differs.

In addition to the inherent variability in devices, change, in both hardware design and software requirements, often occurs during the development process. Sometimes hardware components are replaced due to new offerings, availability or economic considerations. The physical PCB layout often changes because of mistakes or space considerations, and software that depends on particular hardware functionality may need to adapt to new or evolving application needs. Clearly, the software supporting the

functionality of a microcontroller-based device application must be able to endure and adapt to variability and change with minimal use of programmer resources.

Applications developed for resource-rich environments such as our desktop and laptop computers have largely been shielded from low-level hardware details by operating systems. These general purpose operating systems (OSs) provide interfaces to hardware specific functionality and resource management facilities such as virtual memory. By utilizing the abstract interfaces and services the OS provides, programmers can assume a uniform execution environment and are relieved from the the burden of implementing, or even considering, low-level, hardware specific code and the need to consider configuration of their application functionality for a particular execution context. Moreover, features and constructs found in high-level languages such as polymorphism and dynamic type systems provide additional support for software development to cope with change.

In the resource-constrained environment, general purpose operating systems and language features such as polymorphism and dynamic type systems consume too much memory and too many CPU cycles to be utilized. Current mechanisms to cope with change and variability using C, the dominant language of microcontroller programming, involves interleaving preprocessor directives such as `#define` and `#ifdef/ #endif` with application functionality. To vary the configuration of applications and tune them for particular execution contexts, developers currently use tools such as Make

and other build systems to pass to the C preprocessor and compiler the appropriate definitions that will build the application as necessary. Such practices make reading, understanding and maintaining code arduous and error prone. Furthermore, it requires developers to use yet another tool in the development process that demands significant effort and experience to learn how to use. In our design of a language for resource-constrained devices, we see it essential that the language have constructs and features, with unobtrusive syntax and easy to understand semantics that allow software to be written such that it can cope with change, variability and be configured for specific execution contexts from the onset of development.

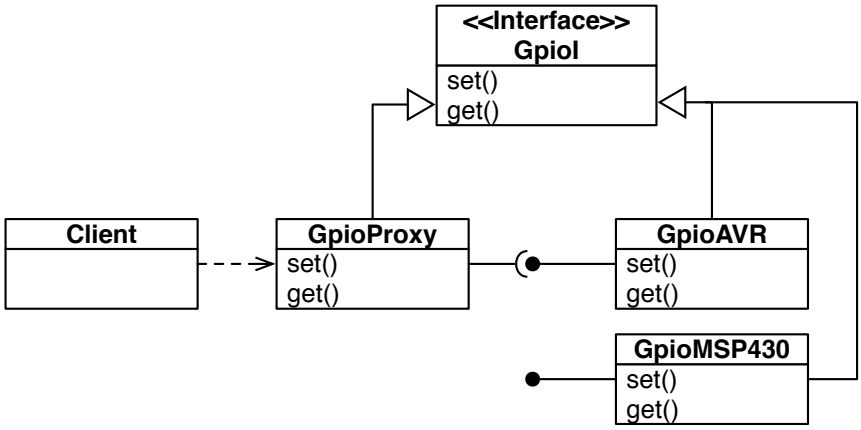

Figure 4.1: Proxy software design pattern in UML.

A language construct and software technique we have identified to cope with change and variability that does not demand runtime resources are interfaces and the proxy software design pattern [25]. As stated in the above section, an interface specifies the

functionality a module provides without defining an implementation. Modules in turn implement the interface and provide a specific implementation. The proxy software design pattern, shown in Figure 4.1, makes use of interfaces, a proxy, and a delegate module to enable the decoupling of a module's interface from its implementation. Software modules requiring a particular functionality, referred to here as clients, utilize a proxy which implements a particular interface. The proxy presents the client with the functionality the interface specifies. The proxy, however, does not provide any implementation and makes use of a delegate module that implements the same interface as the proxy for that purpose. The binding between the proxy and its delegate is a loose one allowing different delegate modules implementing the same interface to be bound to the proxy. In Figure 4.1 a client module utilizes a proxy that implements an interface for the functionality of a microcontroller's general purpose I/O pin. Two other modules shown implement the functionality of the interface for the Atmel AVR and Texas Instruments MSP430 microcontrollers. By changing the proxy's binding to a delegate the client can be adapted to different implementations of the same functionality.

To support capabilities of configuring modules for differing application requirements and execution contexts, we have designed functionality that exists in application build tools directly into the language itself. Specifically, we have allowed module variables and functions to be designated as configuration parameters and host executable functions. A phase in the software build process permits the inspection and manip-

ulation of configuration parameters and utilization of the configuration functions by modules. Functions designated as host executable functions that are programmed into modules allow modules to utilize a build-time execution context. It is during this context that modules can inspect other modules and adapt their functionality accordingly. We have designed this functionality to utilize syntax and semantics symmetrical to other module functions. The result of this design allows modules to be built such that they can configure themselves, or be configured by other modules at a point in time, specifically at application build time, when the precise application requirements, modules included in an application and target hardware configuration are all known.

4.5 Software Reuse

Modern software development practices depend on the existence of reusable software and high-level languages support such practices. Software development efforts consume both time and money and making the process as efficient as possible is imperative to rapidly building complex systems. Key to making the development process efficient is avoiding redeveloping functionality for an application that has already been developed in the past. Software reuse via copy-and-paste may work well for a single developer or a small team. However, this method does not scale and does not enable a systematic reuse of code.

Language features such as inheritance and dynamic parametric polymorphism (dynamic dispatch) have traditionally supported systematic reuse of code. Inheritance supports reuse by letting programmers absorb existing functionality saving the retyping and continual maintenance of that code. Polymorphism supports reuse by letting existing software use different implementations of functionality without requiring modification. Unfortunately, both of these features pose challenges for our device domain and the design goals we have previously laid out. Traditional methods of inheritance can violate modularity due to a problem known as the fragile base class problem [57]. Inheritance of implementation (code bodies) enables a programmer to leverage the functionality of a software element by absorbing all or parts of the implementation of the element. Saving the retyping of existing code and reusing its functionality, the programmer then only needs to implement the variations on the existing functionality they require. The sharing of implementation details between inherited and inheriting elements however can lead to a violation of the principles of encapsulation and data hiding, or modularity. When modular elements have their implementation details exposed to and being depended on by inheriting elements they become fragile; any modifications to the inherited element can potentially break the functionality of the inheriting one.

Dynamic dispatch introduces overhead at runtime to both time and space. In terms of time, this feature requires that a function (or field access) be done at runtime depending on the underlying type of an object. This level of indirection introduces overhead

for look-up and indirect function invocation. We must use space to track this type information at runtime and to provide support for dynamic dispatch in the implementation of the runtime.

In our design, we seek to enable a systematic reuse of software while not consuming critical runtime resources or compromising development efforts. Toward this end, we employ limited forms of both inheritance and polymorphism in support of reuse. Our design includes inheritance of interfaces but not inheritance of implementation. By forbidding implementation inheritance we avoid the problem where a slight modification of an existing piece of code by one developer breaks the code of another because of some dependency on an inherited feature. We have, however, kept the ability for interfaces to inherit from other interfaces, since they strictly specify functionality without implementations (code bodies).

Our design provides static polymorphism, i.e. polymorphism at build-time, to facilitate flexibility in binding implementations to interfaces. In our design, modules can be developed using *proxies* that implement a specific interface as a substitute for a module implementing that interface. The binding of an implementation to the proxy can take place as late as the actual build process - once a specific hardware and software configuration is known. Different implementations of the same interface can be bound to the proxy in different contexts.

Using such static polymorphism, programmers can write modules that depend on other modules for functionality such that they are completely decoupled from an implementation. Modules that utilize hardware resources can use proxies with interfaces that define the hardware functionality becoming decoupled from a particular implementation and reusable across hardware systems. Likewise, modules that depend on other potentially software-only subsystems can utilize proxies for those subsystems enabling the module to be reused with differing implementations of those subsystems.

Other features within the design of our language in support of reuse include automatic source generation, limited delegation, and associating modules that target a particular functional purpose. To preclude cut-and-paste reuse but to gain benefits that are similar, we automate creation of new modules from existing ones through a feature called *templates*.

One benefit that users achieve from implementation inheritance is the ability to employ an implementation from another module directly, to customize the implementation, or to replace the implementation. Programmers can achieve such functionality though a limited form of delegation. If a programmer wants to extend an existing module with new functionality, she creates a new module with, at least, the same interface as the other module (the one she wants to extend). Where the functionality of the original module is required, the new module calls the existing module's functions within its function of the same name. If functionality differs, then the new module can take

its differing actions before and after the call to the original module. In this manner, the implementation of the original module is completely reused and the new module is immune to any implementation changes in the module it is using.

Finally, we include a construct we call a *composite* in our design. Composites are modules that aggregate and configure existing modules for a specific purpose. Composites aim to achieve reuse though composition [79]. Composites do not implement new functionality, rather they allow for the configuration of existing modules to provide functionality specialized for a particular context or application. For example, a composite can be used to create a module that represents a TCP/IP stack for a highly-resource constrained microcontroller configuring the protocol options available and packet buffer size, for example. Another composite can be used to create a stack configured in a different manner for a microcontroller with more resources, reusing the same modules.

We have included these reuse-based features in our design because they are likely to enable more people to create applications. The existence of large amounts of software content that spans application domains has contributed significantly to enabling less experienced and non-programmers create applications. We see this today in the domain of web applications, where libraries, frameworks, and other software content has enabled relative novices to create usable applications, and those with slightly more development experience, create viable products and services.

4.6 Software Portability and Distributability

Very little microcontroller software exists today that is usable across resource constrained device applications and readily packaged for distribution. While downloading C code for a microcontroller from the Internet is quite simple, it is far from likely that it will function on a microcontroller and hardware configuration different from the one for which it was developed.

In contrast, and using Java as an example, software in resource rich environments is by default portable across numerous hardware configurations and the language itself has constructs to easily package it for distribution. Our language design seeks to enable microcontroller software to be downloadable from the Internet and usable in applications with differing microcontroller architectures and hardware configuration. That software should be able to support a diversity of hardware components such as microcontrollers, sensors, actuators, communication interfaces and displays. The process should be as simple as downloading a module or collection of modules from the Internet, placing them in a directory and utilizing their functionality in applications immediately.

Our language design addresses portability and distributability by utilizing the constructs of the interface, proxy, and composite described previously. With interfaces we can define a hardware abstraction layer that defines the basic functionality of common microcontroller functionality such as general purpose input/output pins (GPIO),

timers, serial ports, analog to digital converters (ADC), and other integrated peripherals. The same can be done for non-integrated peripherals. The presence of such interfaces enables the development of higher level software that depends on hardware functionality without any concern for, or knowledge of, lower-level implementation details. A standard set of such interfaces along with microcontroller specific implementations right away enables code that utilizes them - free of device specific features - to be portable. When these interfaces represent non-integrated peripherals, the capacity to write portable device drivers also exists.

In conjunction with a hardware abstraction layer, proxies and configuration parameters enable software modules that utilize other modules to be written portably. The manner in which proxies enable reuse was described in the previous section. Along with the ability to use configuration parameters, modules that depend on hardware in a specific configuration can be written such that they can be portable. For example, the functionality of controlling a light emitting diode (LED), commonly used on hardware as a status indicator, is independent of any hardware or application. However, it uses a microcontroller pin to control the light which can be wired to the pin in one of two different ways. By implementing an LED module utilizing a proxy for the GPIO interface and a configuration parameter to specify if the physical LED is wired to the microcontroller in an active high or active low configuration, the module can be usable in any hardware configuration

To make entire applications portable, composites can be used to provide a central point of configuration for proxies and hardware specific configuration parameters. With all configuration for an execution context being made inside a composite, an entire application can be ported, simply by creating a new composite which configures the application modules differently. Of course, device specific implementations for all the interfaces used must be provided and identical hardware functionality must be present on each device on which the application is to function on.

To support ease of distributability of software, we have incorporated a packaging construct in our design similar to that which exists in Java. Every module belongs to a specific package and the construct for defining packages is identical to that in Java. This simple construct provides a way to both organize modules with related functionality as well as package them for easy distribution. By simply wrapping up a package the modules within it are distributable as a unit. Furthermore, with the use of composites to configure modules, an entire package can be configured for a particular purpose and be distributed in a ready-to-use manner.

4.7 Dynamic Behavior

Many behaviors associated with dynamism in high-level languages come from the ability to allocate and deallocate application memory at runtime and from dynamically

polymorphic type systems. Both of these general features of dynamism consume significant memory and CPU cycles and complicate the runtime system. As such, we omit them from our design. This decision, however, does not preclude the extension of our language and runtime with such functionality in the future.

The allocation of memory at runtime requires logic for a policy that will manage available memory. The deallocation of memory requires a garbage collector (GC) if the programmer is to be free from this error prone activity, or again, complex logic is necessary for the memory management policy if a GC is not available. The use of a GC is impractical since its functionality demands scarce data memory and CPU cycles at runtime, potentially interfering with the timely response to external stimuli. Explicit memory management imposes a significant burden on programmers.

Type systems that support dynamic polymorphism are also unsuitable for resource constrained systems because of the resources they demand. Additional memory is required at runtime to hold a virtual function table associated with each software element, or module. Since modules can potentially contain many functions and since the availability of memory is so constrained, this features is not suitable for resource constrained systems. Additionally, to support dynamic polymorphism, precious CPU cycles must be utilized to resolve overloaded functions. This indirection involved with function calls accumulate and can cause real-time deadlines to be missed. In an environment where CPU cycles are scarce, this aspect of dynamism has lower priority.

The downside of omitting dynamic memory allocation and dynamic type systems is a reduced degree of flexibility for programmers. Complex systems may have intricate memory use requirements and without dynamic allocation and memory management the burden of this complexity remains with the programmer. Lacking a dynamic type system, programmers lose the ability to flexibly modify the structure and operation of system components at runtime. Such flexibility could aid in code reuse and sophisticated application behavior.

While runtime dynamism is difficult to support in resource constrained environments, we have chosen to support a level of dynamism in our design that exists at the time applications are built, or build-time. Since our design incorporates an execution context at build-time, we provide the ability to allocate memory dynamically at this time. The allocation becomes static once an application is compiled, but during build-time, data structures can be created similarly to how they are created at runtime in other languages. Furthermore, values can be placed in the allocated data structures and manipulated. The fact that this takes place on the build host implies that zero runtime resources are required yet a level of dynamism is available.

4.8 Support for Legacy Content

The majority of code that exists for resource constrained systems is written in C and our design incorporates mechanisms to access such content. It is standard practice that a header file, distributed by microcontroller manufacturers, define the devices internal memory map. Applications written in C include this header file in order to access register addresses symbolically. Additionally, some microcontroller manufacturers support their devices with libraries of drivers written in C. We do not feel that it should be necessary to rewrite such existing content in our language in order to use it and incorporate a syntactic mechanism to ”drop down” into C to use symbolic names defined in headers and call functions of existing libraries. This is similar to the way assembly directives can be written in C applications allowing the programmer to ”drop down” into assembly.

There are two sides to this decision that are important to consider. On the one hand, providing a mechanism to allow often poorly structured yet functional code to be utilized in our language may lead programmers to lazily use this mechanism to avoid the overhead of redeveloping that code. While this may expedite an implementation, the pitfalls that come along with existing code written in C are carried into the implementation in our language. Over time, some programmers may get used to this practice and the language becomes a slightly better version of C, much like C is often used

as a slightly better version of assembly language. On the other hand, forbidding any such content to be reused requires reimplementation of a significant amount of existing content. This requires significant time and effort and could potentially detract from peoples' interest in the language.

We made the decision to support legacy content from a pragmatic point of view as opposed to an idealistic one. We feel it is necessary to incrementally evolve the solutions existing developers are already familiar with as opposed to force upon them entirely new systems and methods.

4.9 Summary

In this chapter, we present the design of a programming language for device applications using resource constrained microcontrollers. Overall, our design seeks to advance the state of the art in language support for device applications developed by existing programmers and intentionally supporting a broader audience with less technical skills. In consideration of and to appeal to our broader audience we have made decisions related to the readability and writability of the language. Syntactic simplification and a cohesive coding style that reads from top to bottom has been chosen. Modularity and the separation of concerns, a key concept of the majority of high-level programming languages, is central in our design. In conjunction with constructs for modularity, the

availability of an interface construct supports abstraction in the development process. To allow code to adapt and be usable in the numerous configurations that exist in the hardware systems of our domain, support for variability and configurability is essential. Our design supports variability and configurability through the decoupling of interface and implementation enabled by a construct based on the proxy design pattern. Additionally, the use of module configuration parameters and integration of the software build process into the language is a source of support for configurability.

Our design is sufficient to enable an ecosystem of reusable and portable software to exist and be easily distributable. In our selection of mechanisms for reuse, we intentionally avoid constructs that may violate modularity (e.g. implementation inheritance). We provide many of the benefits of implementation inheritance via delegation, templates, and proxies. The constructs we select to support configurability and variability, namely the interface/proxy combination, also lend themselves to software reuse and portability. Our introduction of a construct we call a composite helps support high-level configurability and a pattern for portability. Lastly, our design incorporates a simple construct for packaging modules for distribution.

Runtime and development efficiency are of critical importance in our design decisions. Maintaining the runtime efficiency achievable with today's languages and tools while introducing high-level language features is essential. To that extent our language design leverages the availability and optimization capabilities of extant tools. Addi-

tionally, our design omits a degree of dynamism as its runtime behavior and resource requirements are too costly in our domain. Lastly, our design chooses to enable existing code written in C to be usable despite the trade-offs this option presents.

Chapter 5

The Em Programming Language

This chapter presents the implementation of the language design presented in chapter 4. We call our language Em; reminiscent of its use in embedded systems. We present and discuss each language construct discussed in the previous chapter and give concrete examples of their usage.

5.1 Modules and Types

The fundamental unit of code in Em is the module. Figure 5.1 shows an example of an Em module. Em modules encapsulate the functions and data they define and implement. We refer to such functions and data as the module's *features*.

All features are declared (given names/types) within the `module` (public specification) or `private` (private specification) blocks. We differentiate such blocks syntactically to make data hiding explicit and clear. Modules can only access the public

```
package em.bios

module EventDispatcher {

    proxy GlobalInterrupt implements GlobalInterruptsI
    type EventHandler: Void( e: EventDispatcher.Event& )

    type Event: opaque {
        function init( handler: EventHandler ): Void
        host function initOnHost( handler: EventHandler ): Void
        function post(): Void
        function postFromInterrupt(): Void
    }

    function start() : Void
}

private  {
    def opaque Event {
        elem: BasicListManager.Element
        handler: EventHandler
    }

    var eventList: BasicListManager.List

    function dispatch(): Void
}
```

Figure 5.1: An Em module with public and private specification.

features of other modules. We refer to modules that do so as *clients* of the module they access. Modules in Em are singletons [25] and are not instantiated. Em has no notion of global functions or data, obviating external data dependencies. Names in a module cannot be reused and all names available to other modules are accessed via their fully qualified (module and feature) name (e.g. BasicListManager.List used in module

EventDispatcher of Figure 5.1 is a data type defined in the public specification of the module BasicListManager). Data types are discussed more below.

Function declarations take the form `function fnName (`$[paramType]^*$`) : returnType`, where `fnName` is the function name, $[paramType]^*$ are zero or more parameter data types separated by commas, and `returnType` is the data type of the return value. All functions must be declared and all declarations must be in specification blocks. All primitive data types in Em are passed by value while composite data types are always passed by reference. Further discussion of types in Em follows below.

Function definitions follow the specification blocks in a module. Function definitions take the form `def fnName (`$[paramName]^*$`) {...}`. We employ a sparse syntax approach of modern scripting languages for Em, to reduce significantly the amount of typing required (and thus potential for errors). For example, a semicolon is optional to end a line or a declaration and argument types are omitted in signatures of function definitions. The notion here is that a growing number of programmers are attracted to and familiar with scripting languages that reduce verbosity of code by relaxing syntactic notations often required by other languages. Em seeks to appeal to such programmers in its style of programming.

Module data is represented by variables with primitive and composite data types. Primitive types of integer, float and character exist. In the case of integers, their size must be specified at declaration time and must explicitly be declared as signed or un-

```
def dispatch() {

    while (eventList.hasElements() != 0) {
       var event : Event& = null
        event = eventList.getElement()
        GlobalInterrupt.enable()
        event.handler(event)
    }
}
```

Figure 5.2: Dispatch function from the EventDispatcher module.

signed with a size of 8, 16, or 32 bits; there is no `Int` type. Floating point numbers are always taken to be 32 bits in length. Composite data types available in Em are strings, arrays, structs and opaque types. Strings are represented as arrays of characters. The semantics of arrays are similar to their use in C/C++ and can hold instances of their declared type. Structs follow the semantics of C where only data, primitive or composite can be declared within a struct and all struct members are publicly accessible to anyone.

Opaque types are user defined types that can contain primitive, composite and other opaque types. All data defined as part of an opaque type, however, are private and accessible only through functions that are declared as part of the opaque. The name opaque refers to the fact that the data representation of the composite type are not visible to clients. The declaration of an opaque type takes the form `type typeName: opaque { [host] functionDeclaration* }`. The public

specification of Figure 5.1 shows the declaration of an opaque type called `Event`. The declaration only specifies the publicly accessible functions `init`, `initOnHost`, `post`, and `postFromInterrupt`. Since the data of opaque types are private, the definition of the structure of an opaque is in the private specification. In the example of Figure 5.1, the Event opaque type declares two other opaque types, `BasicListManger.Element` and `EventHandler`, as its members. Functions associated with opaques are declared like other module functions with the addition of the opaque's name prefixing the function name. For example the signature of the opaque `Event`'s `init` function would be `def Event.init(handler){...}`. Finally, while data internal to an opaque type is inaccessible to clients, they are accessible to the public and private functions of the module the opaque is defined within. This features is similar to the *friend* concept available in C++.

The intuition behind opaque types is that data inherently associated with a specific module often needs to be declared for use in and/or passed around to another module without compromising that data's safety. For example, an event dispatcher module which implements a scheduler for an event system must have events declared in and associated with modules that utilize those events. However, modules that use events should not be able to directly manipulate the internal representation of the event type; only the event dispatcher needs such control. The opaque type in Em therefore allows instances of the type to be declared in any module and be passed around safely. Only

through the public functions of the opaque can clients interact with its data, and only the defining module of the type can directly access and manipulate the private data of the type.

```
from em.bios import EventDispatcher
from BoardC import Led
from BoardC import TimerMilli0

module BlinkP {

    config rate: UInt16
}

private {

    var blinkEvent: EventDispatcher.Event
    function blink( event: EventDispatcher.Event ): Void
}

def em$configure() {
    rate = 500          # 500ms blink by default
}

def em$construct() {
    blinkEvent.initOnHost(blink)
}

def em$run() {

    TimerMilli0.start(rate, true, blinkEvent)
    EventDispatcher.start()
}

def blink( event ) {    # the event handler
    Led.toggle()
}
```

Figure 5.3: Using EventDispatcher to Blink a light emitting diode.

Figure 5.3 shows another module, BlinkP, which is a simple application to blink an Led and makes use of the EventDispatcher module and its publicly defined Event type. BlinkP is only able to interact with `Event` data using the functions declared in the opaque type for `Event`.

In addition to its use in declaring opaque types, Em uses the keyword `type` to alias a type similar to how `typedef` functions in C. This is useful when symbolic names to existing types make more sense to programmers who are reading code. For example, in Figure 5.1 a reference to a function that takes an Event type as an argument is renamed more appropriately to EventHandler. The `type` keyword can only be used in the public or private specification blocks of a module.

Module variables must be declared within the public/private specification blocks. The only other types of variables are local variables within functions. All data types are known and all storage is allocated statically. Type coercion is possible in Em but must be specified explicitly. Since subtle runtime memory errors can be introduced by coercion [17] Em forbids implicit type coercion by design. Variables are declared using the keyword `var` followed by the variable name, a colon, its data type, and optionally an equal sign and value (initialization).

Em supports *reference* variables whose semantics are similar to C++ references. References are declared using the & operator (following the type name, e.g. `var event: Event&`) for primitive and composite data types. Unlike C++ references

Em references can be assigned a null value at declaration and may be reassigned after declaration. Reference types provide developers with an option of a safer alternative to pointers. Pointer types are allowed in Em and have the same semantics as in C and C++. However, complex functionality can be achieved without pointers given the parameter passing semantics of Em and references. Advanced functionality requiring pointers and pointer manipulation is still available if necessary.

5.1.1 Pre-runtime Configuration

Em gives programmers the ability to configure modules before code is loaded onto the microcontroller target and executed. We refer to code that is executed on the microcontroller as target code and that which is executed on the build host while the application is being built as build-time or configuration code. Configuration parameters, host variables and host functions enable developers to program configuration functionality into modules. The syntax and semantics of host variables and functions are identical to target code. At the time an application is being built, the physical configuration of the hardware is fixed and known and all modules that make up the application are also known. The configuration parameters and host functions, therefore, can be utilized to leverage the rich resources of the application build host to configure a module for a precise hardware configuration and application execution context.

A configuration (`config`) parameter is a variable that is fully mutable at build time and results in a static constant when the process completes. We refer to the data and functions that are manipulated on the target device as target data and target functions, respectively. Figure 5.3 shows the config parameter `rate` of module BlinkP which sets the blink rate of an LED. Configuration parameters that appear in a module's public specification can be inspected and modified by other modules at build time enabling modules to reflect, act on, and respond to a particular application's configuration.

In Figure 5.3, the `rate` of BlinkP can be inspected and modified by other modules at build time before becoming a static constant in the resulting binary. Such constants are typically placed in a microcontrollers read-only memory to save scarce RAM. Configuration parameters that are never referenced by target functions have no representation in the target binary. Developers can identify such variables explicitly using the keyword `host`. This makes the variable available at build time only and the Em translator will issue an error during translation if the variable is referenced from within a target function.

The *host* keyword also applies to functions. Host functions implement code that executes on the development host during the build process. For example, a module that implements a band-pass filter can expose configuration parameters for clients to describe characteristics of the filter, e.g. its passband and its order. The module can then use a host function to access a filter design package on the build-host to compute the

filter's coefficients, to have them ultimately become static constants on the target microcontroller. In our current implementation, host functions are translated to Javascript and interpreted using Rhino [70]. Host functions can access the full power of Javascript and, since the interpreter is Java aware, any Java functionality to provide a wide-range of build time services including computations that cannot be performed on the target microprocessor due to resource constraints or missing functionality, e.g. arbitrary computations, data acquisition over a network, tests or target simulation, profiling, etc.

Host functions can access public members of any module as well as the private specification of their defining module. Host functions cannot execute target functions however. Since both the host and target code are implemented in Em, it unifies the syntax and combines the development and configuration languages. That is, developers need not learn multiple languages (e.g. C/Make, Java/ant, RTSC/XDC, and others) for their target and configuration operations. Moreover, developers can now be creative in deciding what build time operations to include. `initOnHost` of the opaque type Event from our first example module (Figure 5.1) is an example of a host function. This function statically initializes the event type's data members saving the operation from taking place at runtime.

Each module has three functions that are intrinsically part of the module and we refer to them as intrinsic functions. These functions are host-only functions and, if they are present in the module, get executed automatically at build-time. The func-

tions `em$configure` and `em$construct` are used to configure the public and private specifications of a module at build-time respectively. `em$construct` is analogous to a constructor of an object in object-oriented languages and is often used to initialize private features of the module. Figure 5.3 shows an event being initialized with its handler function in `em$construct`. `em$configure` is similar only it is used on any mutable public features. For example, proxies that are public are often bound to concrete implementations in `em$configure`. The last intrinsic function is `em$generateCode` which can be used to inject code into a module's C representation during translation from Em into C. A simple use of `em$generateCode` is to include a C header file by injecting a `#include` directive into a module's C source. We provide an example of this in a subsequent subsection once we define a few other features of Em.

5.1.2 Runtime Configuration

Every device application has a lifecycle which it goes through. At every stage of the lifecycle the application must often perform critical operations to ensure a proper execution environment. For example, upon applying power to the device, the microcontroller is reset at which time it must be configured properly to ensure the rest of the application can execute. Most device applications are intended to run perpetually with

a power-off condition indicating failure. However, some applications do require the ability to be shutdown and need certain operations to take place during that process.

Em modules each have a set of intrinsic functions which serve as entry-points for developers into a module's lifecycle at runtime. Functionality implemented in these intrinsics is commonly found in applications although where it resides and how it is integrated into an application's structure differs. Having these entry points defined and knowledge of their invocation at specific points of runtime makes understanding application functionality more direct.

The intrinsic functions associated with a module's runtime lifecycle are `em$reset`, `em$run`, `em$shutdown`, `em$startup`, `em$startupDone`, `em$fail` and `em$halt`. Each of these functions can be defined in any module, however, except for `em$startup`, only the first definition of each intrinsic encountered during translation is used. `em$startup` can appear in every single module and is used by modules to perform initialization at runtime on the target.

Upon powering up the microcontroller, the first function called, if it exists, is `em$reset`, followed by each occurrence of `em$startup` found in any module. The order in which `em$startup` functions are executed is undefined. Upon completion of all calls to `em$startup` the function `em$startupDone` is called to perform any further tasks that require modules to be in an already initialized state, yet must take place before the main application functionality begins. Finally `em$run` which is

Em's equivalent of main() in C is called. Since the majority of embedded systems are intended to operate indefinitely, the remaining functions, `em$shutdown`, `em$fail` and `em$halt` can be used to gracefully enter fail states should they be encountered at runtime.

5.2 Interfaces

To support software variability and change, Em provides interfaces and proxies, which together, decouple a module's interface from its implementation. Interfaces also aid in providing software abstraction. An Em interface, depicted in Figure 5.4, contains a collection of functions and type declarations that are to be implemented by a module. The notion is that interfaces only specify features that are incomplete. A module that implements a particular interface, in turn, must provide the complete implementation for all of the functions and data representations for opaque types declared in the interface they implement. Interfaces in Em resemble and function much like interfaces in Java.

Figure 5.4 shows the definition of an interface that represents the functionality provided by a microcontroller's general purpose input / output (GPIO) pin. As can be seen, the basic capabilities are specified and no implementation is provided. Figure 5.5 shows part of a module that implements the interface for a specific microcontroller family. Fol-

```
interface GpioI {

    function clear(): Void
    function get(): Bool
    function set(): Void
    function makeInput(): Void
    function makeOutput(): Void
    ...
}
```

Figure 5.4: A general purpose input / output pin interface.

lowing the name of the module is the keyword `implements` followed by the interface name. This specifies that the module provides implementations for all the functions declared in the interface. The implementing module does not respecify the interface functions in its public specification; the semantics of the keyword `implements` is such that all interface functions become part of the public specification of the module. Figure 5.6 shows an Led Module that makes use of a proxy that implements the GPIO interface. Proxies are further discussed in the following section.

Interfaces aid in abstraction in two ways. First, they allow developers to define functionality modules will implement separately from any specific implementation. This enables developers to think strictly in terms of capabilities and purpose of the module. Second, modules that implement interfaces enable developers that use them to not be concerned with their implementation; there is a contract as to what capabilities the modules possesses. Collections of interfaces, such as the GPIO interface above, can be used to define a hardware abstraction layer, for example. Such a collection

```
from em.hal import GpioI

module AvrGpio implements GpioI {
}
...

def em$generateCode(prefix) {
    |-> #include <avr/io.h>
}

def clear() {
    ^PORTA &= ~MASK
}

def get() {
    return ^PORTA & MASK
}

def set() {
    ^PORTA |= MASK
}
...
```

Figure 5.5: A module implementing the Gpio interface.

of interfaces ultimately enables modules that depend to hardware functionality to be implemented without any particular microcontroller target in mind.

5.3 Proxies

Em introduces a language construct, based on a common software design pattern, known as a proxy [25]. Proxies add a level of indirection between clients of a module's functionality and the supplier of that functionality. In conjunction with interfaces, proxies provide a notion akin to a static form of polymorphism. Specifically, modules in Em

```
module Led {

    config activeLow: Bool

    proxy Pin implements GpioI

    function on(): Void
    function off(): Void
    function toggle(): Void
}

def on() {

    if (activeLow) {
        Pin.clear()
    } else {
        Pin.set()
    }
}
...
```

Figure 5.6: Part of an Led module using a proxy.

declare proxies which are specified to implement an interface. The proxy can then be used throughout the implementation of the module with the guarantee that some module, whose implementation is unbeknownst to its client, will provide the functionality. Figure 5.6 shows an Led module that makes use of a general purpose I/O pin (GPIO) proxy (called `Pin`) for its implementation.

Before a module that declares proxies can be utilized, its proxies must be bound to modules that implement the same interface as the proxy. A module is bound to a proxy using the keyword `seal` followed by the proxy name followed by the keyword `as` followed by the module name. The binding of proxies can be specified in a module's

```
from mcu.atmega168.gpio import PB7

module Led {

    config activeLow: Bool
    proxy Pin implements GpioI
    ...
}

def em$configure() {

    seal Pin as PB7
}
...
```

Figure 5.7: Part of an Led module binding proxy in configure function.

intrinsic functions; in `em$configure` if the proxy is declared in the public specification or in `em$construct` if declared in the private specification. Additionally, proxies defined in the public specification can be bound within composites. The benefit of specifying the binding in a composite is that if a different module must be bound to the proxy, the module declaring the proxy need not be modified allowing that code to be reliably stable over time. Composites are further discussed in the next section. Figures 5.7 and 5.8 show how a proxy is bound in both the module itself as well as within a composite.

The combination of interfaces and proxies in Em enable modules to be written to a particular interface without knowledge of the implementation of that interface. Many implementations of an interface may therefore exist and are interchangeable which is analogous to the way classes that implement a particular interface in Java can be substi-

tuted anywhere an object of the interface's type appears. There is no notion of dynamic dispatch of functions in Em to reduce complexity of the runtime and overhead due to function dispatch.

5.4 Composites

A composite in Em is a special module that is used to configure proxies and set configuration parameters of other modules. Composites are special in that they contain no target code, i.e. they are host-only modules. Figure 5.8 shows a composite representing part of a device's hardware configuration.

Composites function and are usable like other modules, namely, they have a specification and an implementation and can be used by other modules to access the modules exported in their public specification. Composites can define functions as part of their private specification. All such functions are inherently host functions although must still be specified as such for consistency. Functions defined by composites can be used at build time for any computation that must be carried out or for logic to determine how proxies or configuration parameters must be bound and set respectively.

Composites have two intrinsic functions defined in which most of its configurations take place: `em$configure` and `em$preconfigure`. `em$configure` is where mutable configurations are made. That is, other modules may, at some point

```
package board.arduino

from em.mcu.atmega168 import GlobalInterrupts
from em.mcu.atmega168 import Mcu
from em.mcu.atmega168 import TimerMilli8BitT0 as TimerMilli0
from em.mcu.atmega168 import PD0 as D0
...
from em.mcu.atmega168 import PB5 as D13
from em.parts import LedT {activeLow: false} as Led

composite BoardC {
    export GlobalInterrupts
    export Led
    export TimerMilli0
    export D0
    ...
}

def em$preconfigure() {
    seal Led.Pin as D13
    Mcu.mcuFrequency = 16000000
}

def em$configure() {
    Uart.baudRate = 9600
}
...
```

Figure 5.8: A composite representing a board configuration.

in the build process, change the configurations that take place within that composite. `em$preconfigure` is where immutable configurations occur. Since the BoardC composite in Figure 5.8 reflects some hard-wired aspects of an application's hardware, namely its oscillator's frequency and a GPIO pin connected to an Led, these aspects must be configured such that no other module may change them. The Em transla-

tor enforces that configurations within `em$preconfigure` are not changed by other modules.

In Figure 5.3, BlinkP imports the Led and TimerMilli0 modules from BoardC of Figure 5.8. The composite's public specification exposes modules, potentially under aliased names. The composite's implementation provides configurations in the form of proxy bindings and settings for configuration parameters. Code within a composite does not make its way on to the target device - it is used only at build time to configure modules.

Composites can also be used to achieve application portability. Using the capability to export modules under aliased names, an application can be written such that all references to hardware features come from a top-level composite under application-specific names. For example the BlinkP application in Figure 5.3 imports an Led and millisecond timer from the BoardC composite shown in Figure 5.8. BlinkP's functionality is now dependent only on the modules named Led and TimerMilli0. For BlinkP to be ported to different hardware only a new composite must be created that exports a millisecond timer and Led under the same names BlinkP expects. None of the application code must be modified. Applications of all complexities can follow this pattern to decouple their dependencies on low-level hardware specific implementations and achieve portability.

5.5 Templates

A second special module is the template. Templates are used at configuration time to generate other Em modules automatically. Templates also, if required, can generate C code. Template modules, like composites, only exist at build time and execute on the build host (all template functions are host functions). We refer to the generation of a module from a template as instantiation of a template. Figure 5.9 shows an Em template.

A template intrinsic, `em$generateUnit`, is defined and is invoked at build time for each template instantiation that is encountered. The public specification of a template specifies parameters that are necessary for its instantiation. Templates, like composites, may have functions declared in their private specifications and implemented in their bodies. These functions are host-only functions and must also be declared as such for consistency. Template functions can be implemented to assist in decision making and computation necessary to generate a module via the template. Figure 5.8 shows an instantiation of an Led module via the LedT template. Within the `em$generateUnit` intrinsic the `|->` symbol denotes text that will be generated by the template and, within that line, text within backticks enables access to data and functions of the template module itself. Any logic and computation can be interwoven between lines of generated text to control how the resulting module is generated.

In addition to the intrinsic function `em$generateUnit`, templates have two often used host-only intrinsics variables, `em$packageName` and `em$unitName`. Since every module resides within a package the value of `em$packageName` is always defined to be the name of the package the template is instantiated in. For the Led module instantiated from template LedT in Figure 5.8, `em$packageName` will have a value of `board.arduino`. The value of `em$unitName` is always defined to be the name given to the module at instantiation and is `Led` for Figure 5.8.

It occasionally occurs that multiple instances of a module must exist that operate on different data values or must access different hardware registers, yet function identically. Since modules are singletons programmers should have a more efficient method to create such modules. Our goal with templates is to avoid copying and pasting across modules. That is, templates are used to generate modules that have similar functionality with minor variations – to reduce the potential of bug propagation from copy-paste activities.

As an example, the operation of a general purpose I/O pin is identical for all pins although each pin has a distinct bit in a specific hardware register it must operate on. In Figure 5.9, we define a template for the GPIO module. The module's public specification shows that the module requires a port name and pin number as input. A GPIO module is instantiated from within a composite or a module by importing the template, specifying its parameters, and naming the resulting module. Specifically, the

```
template GpioT {
    config port: String
    config pin: UInt8
}

def em$generateUnit() {
    var MASK: UInt16 = (1 << GpioT.pin)

    |-> package `GpioT.em$packageName`

    |-> from em.hal import GpioI

    |-> module `GpioT.em$unitName` implements GpioI { }

    |-> def em$generateCode( prefix ) {
    |->     |-> #include <msp430x22x4.h>
    |-> }

    |-> def clear() {
    |->     ^`GpioT.port`OUT &= ~`MASK`
    |-> }

    |-> def set() {
    |->     ^`GpioT.port`OUT |= `MASK`;
    |-> }

    ...
}
```

Figure 5.9: A template to generate GPIO modules.

statement `import GpioT {port: "P1", pin: 0} as P1_0` will create a GPIO module with the name P1_0.

As another example, the LedT template shown in Figure 5.10 is used to generate a module to control a light emitting diode (LED). It could be the case that a hardware design incorporates multiple LEDs. The control logic is identical for each LED, however to control each one independently they are often physically wired to different micro-

```
package em.parts

template LedT {
    host config activeLow: Bool = false
}

def em$generateUnit() {
    |-> package `LedT.em$packageName`
    ...
    |-> module `LedT.em$unitName` implements LedI {
    |->     proxy pin implements GpioI
    |-> }

    |-> def em$startup() {
    if (LedT.activeLow) {
        |->    pin.makeInput()
        |->    pin.set()
    }
    else {
        |->     pin.makeOutput()
        |->     pin.clear()
    }
    |-> }

    |-> def on() {
    if (LedT.activeLow) {
        |->     pin.clear()
    }
    else {
        |->     pin.set()
    }
    |-> }
    ...
}
```

Figure 5.10: Template to generate Led Module.

controller pins. Additionally, the manner in which they are wired - an active high verses active low - can differ. To save the programmer from implementing a module for each device, the template was created. The parameters for the template specify which microcontroller pin the LED is connected to and the exact configuration of its wiring. In this manner, multiple instances that share the same functionality can be easily created.

Finally, it is occasionally necessary to generate some C code for microcontroller specific functionality required by the application. For example, the definition of interrupt handling functions requires specific attributes added to the generated C function signature. To properly decorate this signature, a template can be used to generate that code. Additionally, in devices with configurable interrupt vectors, a template can be used to generate the entire table at build-time with knowledge of the exact interrupts being used. This allows the table to be initialized statically, saving both runtime memory and CPU cycles to achieve the same purpose on the target itself.

5.6 Inheritance

Inheritance, a common language feature of higher-level, object-oriented languages, is available to a limited degree in Em. In particular, interface and composite inheritance is supported while implementation inheritance is not. In addition, we support only single inheritance in these cases.

We achieve interface inheritance by extending an existing interface and declaring its defined functions as shown in Figure 5.12. All functions and type declarations are inherited by the extending interface. Composite inheritance enables an existing composite to be extended by another with the extending composite inheriting the public interface, that is the set of exports, of the extended composite.

```
interface GpioI {

    function clear(): Void
    function get(): Bool
    function set(): Void
    function makeInput(): Void
    function makeOutput(): Void
    ...
}
```

Figure 5.11: Part of an interface for an I/O pin.

We disallow implementation inheritance in Em since it can lead to situations where changes in an inherited class, either syntactic and semantic, break an inheriting class. This is the well known problem of the fragile base class [57, 78]. Such issues can lead to a breach in modularity, which with Em, we attempt to maintain strictly.

```
interface GpioEdgeDetectI extends GpioI {

    function clearDetect(): Void
    function disableDetect(): Void
    function enableDetect(): Void
    function setDetectFallingEdge(): Void
    function setDetectRisingEdge(): Void
    ...
}
```

Figure 5.12: An interface that extends GpioI.

The major benefits of implementation inheritance, however, can be achieved in Em through a simple pattern of forwarding. For example, suppose a module M implements the GpioI interface illustrated in Figure 5.11 and another module N implements the EdgeDetectGpioI interface which extends the interface of GpioI in Figure 5.12. To

avoid re-implementing the functionality of M within N, module N defines each function of the GpioI interface and then in each defined function forwards the call to the implementation in M, as shown in Figure 5.13. If the GpioI functions of module N must behave differently from the existing functionality defined in M, it can take its necessary actions before and after the calls to module M.

```
from em.hal import GpioEdgeDetectI
from em.mcu.atmega168 import PD0 as M

module N implements GpioEdgeDetectI {
...
}

def clear() { M.clear() }

def get() { return M.get() }

def set() { M.set() }
...
```

Figure 5.13: Example of implementation inheritance by forwarding.

Equivalent to implementation inheritance, if functions defined in M are modified, then N will utilize these modifications without requiring any change. Unlike implementation inheritance however, M's internal representation may be modified without consequence to N. With the aid of an integrated development environment such as Eclipse, it is trivial to automate the definition of functions that forward calls to save the programmer from typing them.

5.7 Support for Legacy Content

The Em language has several mechanisms with which Em code can interact with existing C code. There are two common reasons for such interaction. The first is to incorporate legacy source (e.g. as an initial step to get something running before refactoring it using Em). The second is to access the C code that is generated during module translation directly.

```
module Uart0 implements UartI {
}

def em$generateCode(prefix) {
    |-> #include <driverlib/uart.h>
    ...
}

def put( data ) {
    ^UARTCharPut(^UART0_BASE, data)
}

def get() {
    return ^UARTCharGet(^UART0_BASE)
}
...
```

Figure 5.14: Part of a UART module interacting with C identifiers.

The ^ operator placed in front of an identifier, or an expression bounded by ^^ instructs the Em translator to bypass parsing of the identifier or expression and pass it, unchecked, through to the generated C code. Referencing symbolic names of memory-mapped registers, calling functions defined in existing C code, and inserting inline assembly, for example, can be handled using these mechanisms. Figure 5.14 shows part

of a UART module that references both symbolic register names and calls functions defined in an existing C library. These mechanism in Em make it possible to integrate existing C code into Em modules in a straightforward manner.

5.8 Packages

Finally, every module in Em must reside within a package (e.g. namespace). The `BoardC` composite in Figure 5.8 for example, resides within the `board.arduino` package. Modules can bring another module into their namespace using the `import` statement. Imported modules can be aliased by suffixing the `import` statement with `as` followed by the alias. Modules within the same package can simply import the module by name. Modules that reside in different packages must qualify the module's package by using the `from` clause as seen at the top of Figure 5.3.

Packages are present in Em as formal construct to aid in both organization of code and in its distribution. Modules of related functionality are often placed in packages that reflect their intended purpose. The packaging mechanism in Em resembles Java's packaging semantics. As an example, a hardware abstraction layer (HAL) we have created, which consists of interfaces defined for commonly found hardware peripherals were placed in a packaged called `em.hal`. Distribution of this package is simplified

since all the files of the package are also located within the same physical directory on a storage medium.

5.9 Translation

Em applications are ultimately translated to a single C file that is compiled into binary images for their intended microcontroller targets. In this section, we describe the process of transformation of an Em program from a single source module into a binary image and constitutes what we refer to as *build time*. Figure 5.15 depicts the high-level flow of build time artifacts, starting with the module ModP.em source file and ending with the ModP-prog.out binary image.

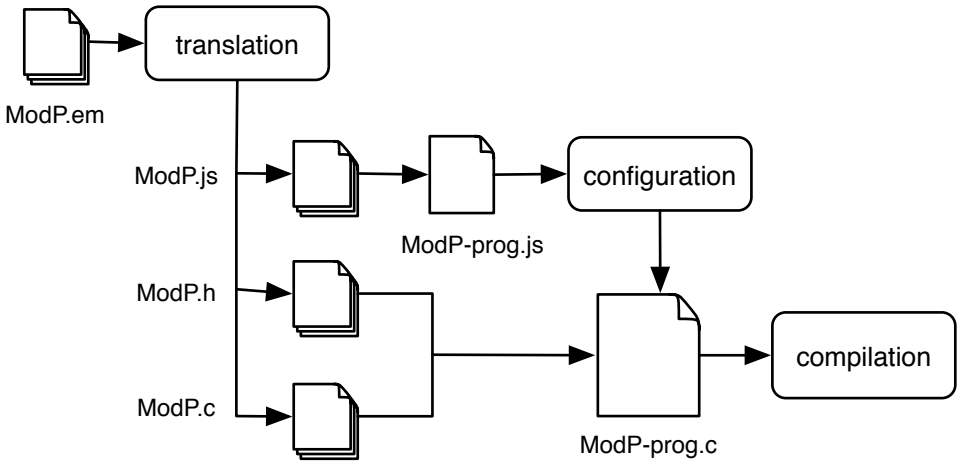

Figure 5.15: Build flow of the Em translator.

Each Em source file (a module, interface, composite, or template) represents an independent unit of translation. From Figure 5.15, starting at the top-level unit, ModP.em, the translator recursively processes an N-element hierarchy of translation units that ModP directly or indirectly imports. Cycles may not occur in the hierarchy and if one is encountered the translator issues an error and stops.

The translation of ModP yields a top-to-bottom partial ordering of dependent units. The translation of modules produces three corresponding output files that are consumed in subsequent phases of the build-process. These are (i) ModP.h, which contains the public and private feature declarations of ModP translated into a C header file; (ii) ModP.c, which contains function definitions within ModP translated into equivalent C code, and (iii) ModP.js, which is a Javascript implementation of ModP that implements the host configuration functionality. All template instantiations encountered in modules trigger the generation of Em modules which are then recursively translated into C header, C implementation, and Javascript implementation files. The subsection below depicts and discusses examples of the generated source files.

Following translation, the configuration phase of an Em program begins. All generated Javascript files are amalgamated and interpreted by a Javascript interpreter. This process makes three top-to-bottom passes over the N-element hierarchy and executes module intrinsic host functions along with any other host function referenced from those functions. The choice of Javascript in the Em translator implementation was

arbitrary and other languages could have been used as the underlying configuration language. The use of Javascript however, does enable host functions to access external Java functionality and leverage the rich amount of existing content for many purposes. For example, accessing a database, potentially on the Internet, of profile metrics that can aid in configuring the application for optimal performance.

Each pass invokes a different set of intrinsic functions, for example, on the first pass, the `em$preconfigure` function within composites is called to bind proxy variables to delegate modules. As a consequence of the first pass of configuration, the original N-element hierarchy becomes pruned to an M-element subset which comprises only those modules that are actually used within the program. The second pass calls each participating module's `em$configure` function. This function configures all public aspects of each module (e.g. public configuration parameters).

The final configuration pass calls module's `em$construct` functions. These functions use the public features of other modules, all of which have already been configured, to initialize the private aspects of the modules (e.g. private configuration parameters). Moreover, the ability to inspect other module's public configuration parameters enables a form of introspection, or reflection, into the state of the application being built, providing opportunity for modules to adapt given the particular hardware/software configuration. The result of configuration is a single .c file that is com-

piled by a target specific compiler, aggressively optimized by that compiler, and linked into a binary executable for the target system.

5.9.1 Source Code Validity

The following rules are currently adhered to when checking for the validity of an Em source file. For all module types - modules, interfaces, composites and templates - only one module may appear in a single source file and both the file and module name must be identical. If a filename does not match a module name, the translator issues an error and translation halts. This behavior is similar to that found in Java. Additionally, multiple opaque types may be declared within a module and all opaque types must be defined within the module that specifies the opaque type's data representation. Not specifying a representation for an opaque type which has been defined is flagged as an error.

Within all modules, if a private specification exists it must follow the public specification of the module. Following the private specification, definitions for all publicly and privately declared module and opaque type functions must appear. The order in which function definitions appear within a file does not matter and can be interleaved or out of order. However, if definitions appear before the public or private specifications of the module, translation halts with an error.

All identifiers not declared within the module most be imported into the module's namespace using the `import` keyword. Each module has a unique namespace defined by its fully qualified name which consists of a dot (.) separated list of packages the module belongs to, followed by the module name. For example, the EventDispatcher module in Figure 5.1 has a fully qualified name of `em.bios.EventDispatcher`. Two modules in the same package with the same name is an error which is caught by the translator when one module attempts to import the other or when a third module attempts to import one of the two similarly named modules.

5.9.2 Generated C Code

Figure 5.16 shows a simple module that makes use of a proxy. In this subsection we walk through the C code generated by this module during the translation process. All modules participating in an application are individually translated to C source and headers as discussed in the previous section. Additionally, a C source file is generated for the whole application, which then includes the other generated C sources in order to present a C compiler a single file representing the entire application.

Figure 5.17 shows the complete C translation of the `Simple` module. The first part of the file provides mappings between functions which have been called on a proxy and the implementation of the functions provided by, or delegated to, the module which was bound to the proxy. The definitions in this proxy section fully specify the dele-

```
from em.hal import BusyWaitI
from em.mcu.avr import BusyWait

module Simple {

    proxy bw implements BusyWaitI
}

private {

    var i: UInt8
}

def em$configure() {
    seal bw as BusyWait
}

def em$run() {

    for (i = 0; i < 10; i++) {
        bw.wait(10000)
    }
}
```

Figure 5.16: A Simple module.

gates of all proxies. Following the proxy definitions come the inclusion of the headers defined by each module involved in the application. The header generated for the Simple module is depicted in Figure 5.18. The header contains `#include` directives for other headers which define types necessary within the module's namespace. The list of headers is defined by the set of imports the original Em module makes. Declarations of structures, constants, opaque types, configuration parameters, module functions, opaque type functions and private variables follow. Each module header has an identical structure.

Following the inclusion of headers in the complete translation of Figure 5.17 are forward declarations of functions and values. These forward declarations may be nec-

```
// PROXY first.Simple.bw DELEGATES em.mcu.avr.BusyWait

#ifndef em_mcu_avr_BusyWait__M
#define em_mcu_avr_BusyWait__M
#include "../../em.mcu.avr/BusyWait/BusyWait.h"
#endif

#define first_Simple_bw_wait__E em_mcu_avr_BusyWait_wait__E

// UNIT HEADERS

#ifndef em_mcu_avr_BusyWait__M
#define em_mcu_avr_BusyWait__M
#include "../../em.mcu.avr/BusyWait/BusyWait.h"
#endif

#ifndef first_Simple__M
#define first_Simple__M
#include "../../first/Simple/Simple.h"
#endif


// FORWARD AGGREGATE DECLATIONS

// FORWARD AGGREGATE VALUES

// em.mcu.avr.BusyWait DEFINITIONS

// first.Simple DEFINITIONS

UInt8 first_Simple_i__V = (UInt8)0x0;

// MODULE FUNCTIONS

#include "../../em.mcu.avr/BusyWait/BusyWait.c"
#include "../../first/Simple/Simple.c"

// RUN FUNCTION

void __run() {
    first_Simple_em__run__E();
    __halt();
}

#include <em.lang/main.c>
```

Figure 5.17: The complete translated C program.

essary due to inclusion dependencies which are resolved in the initial passes of translation. Variable definitions follow next along with inclusion of the C files generated for other modules. Figure 5.19 shows the C file generated for the Simple module which contains the bodies of all functions defined within the module. After the inclusion of the C files in the complete translation, is the definitions of intrinsic functions related to application lifecycle and only the `__run` function appears in the figure. Lastly ap-

pears an inclusion of a file which contains the necessary C `main` function which simply invokes the `__run` function.

```
// IMPORTS

#ifndef em_mcu_avr_BusyWait__M
#define em_mcu_avr_BusyWait__M
#include "../../em.mcu.avr/BusyWait/BusyWait.h"
#endif

// STRUCT NAMES

// CONST/TYPE DEFINITIONS

// CONFIG PARAMETERS

// FUNCTION DECLARATIONS

extern Void first_Simple_em__fail__E( void );
extern Void first_Simple_em__halt__E( void );
extern Void first_Simple_em__reset__E( void );
extern Void first_Simple_em__run__E( void );
extern Void first_Simple_em__shutdown__E( void );
extern Void first_Simple_em__startup__E( void );
extern Void first_Simple_em__startupDone__E( void );

// OPAQUE METHODS

// PRIVATE VARIABLES

extern UInt8 first_Simple_i__V;
```

Figure 5.18: The generated header for the Simple module.

```
// FUNCTION BODIES

Void first_Simple_em__run__E( void ) {
    for (first_Simple_i__V = 0; first_Simple_i__V < 10; first_Simple_i__V++) {
        first_Simple_bw_wait__E(10000);
    }
}
```

Figure 5.19: The generated C code for the Simple module.

5.10 Comparison With Related Systems

The system most closely related to Em in language features, and that also targets resource constrained microcontrollers, is nesC. NesC is an extension of the C programming language and fully supports all features present in C, including ones that lead to unstructured programming such as goto statements. Our design and implementation of Em explicitly removes such C constructs. While nesC depends on common practices that utilize C preprocessor macros and directives for conditional compilation and static variable definitions, Em explicitly does away with such features since they make reading, understanding and maintaining code difficult and error prone. Expressivity in Em is not lost since its build-time execution context and availability of proxies provide similar support.

NesC has constructs for modularity called *components*. Unlike Em, components do not exist within a global namespace requiring that the interaction between modules be specified in what nesC calls *configurations*. NesC requires at least two files to construct an application, a configuration that interconnects collaborating components, and at least one component. Em's notion of modularity is more closely related to the object-oriented model where programmers can explicitly define what functions and parameters modules make public or private. The namespace of other modules in Em is made accessible by directly importing a module into its namespace. The difference in Em is

therefore a reduction in the steps necessary to create applications and a reduction in the need to specify interfaces and behavior in abstract levels at the start of development. Development in Em is also more familiar for experienced object-oriented programmers since it inherets many of these features from existing high-level langauges. Using Em, abstractions and reductions in dependencies between modules come as a refinement of application development, whereas nesC requires the upfront definition of such abstractions and intricate wiring of modules, making development often challenging.

The separation of interface from implementation in nesC is required for all software components. While technically the separation is beneficial to allow software reuse and variability in systems, practically, this requirement poses challenges to development even for experts to use. Wiring interfaces together in nesC configurations is also challenging since components often use multiple fine grained interfaces, applications consist of potentially large quantities of modules, and all module interfaces must be wired together to operate. Em allows the complexity to scale by providing mechanisms for the separation of interface from implementation but does not require it at the onset of development. Em allows an incremental, agile development process where enhancements such as higher levels of abstraction, support for variability, and module reuse can be applied incrementally. NesC, in contrast, in our experience, compels the developer to invest larger amounts of time and effort at the onset of development.

NesC also lacks the build-time execution context for configuration that Em offers. The make UNIX utility [29] and the C preprocessor are the mechanisms used in NesC to support build-time application configuration capabilities. Developers must be knowledgable in both nesC, the C preprocessor, and the Make build system to leverage the configuration capabilities that are integrated directly into Em. Opportunities to offload computation from the target device onto the build host and optimizations that save target resources are therefore not as easily achievable in nesC as they are in Em. Additionally, Em's configuration capabilities are more accessible to less experienced developers since a single language syntax and semantics must be understood to harness both build-time and application operations.

NesC is fundamentally not an object-oriented language, while Em strives to achieve functionality proven to be effective for development found in such languages. Constructs for interface inheritance and packages are therefore not found in nesC. Additionally, while nesC has mechanisms for generic interfaces, the templating constructs available in Em allow for more general purpose usage such as complete module generation.

The Arduino platform is another system that is related to Em. However, Arduino utilizes C/C++ as its implementation language. While C++ is used to afford modularity, many of the language's features are unusable in such constrained environments. We found that a simple Arduino program in C++ that dynamically allocates a variable

consumes nearly 60% of the device's program memory. We measured this by writing a simple C++ class using the Arduino platform, with the main application allocating an instance of it using the `new` keyword. The pitfalls of C as applied in the resource constrained domain are shared by the Ardunio platform as application functionality becomes non-trivial.

5.11 Summary

In summary, this chapter presents the implementation of the language design we articulate in chapter 4. Our language, called Em, supports modules as the fundamental unit of code. Em modules enable encapsulation and enforce information hiding and contribute to the language's abstraction capabilities. To support build-time configuration and standard runtime initialization procedures, modules have intrinsic functions as convenient hooks and points of entry to both processes, respectively. Module host functions and configuration parameters allow Em modules to holistically interface with a build-time configuration context.

Em has the ability to define new data types as opaque structures that can safely be passed and shared between modules. Additionally, we support standard primitive types available in C with the additional requirement of explicitly specifying byte width in variables that can have multiple definitions e.g. integers. While C-style pointers are

currently available in Em, the availability of C++ style references in Em make the direct use of pointers largely unnecessary.

In support of abstraction, software variability, configurability, reusability and portability we implement the constructs of interfaces, proxies, configuration parameters and composites. The definition and use of these constructs are demonstrated and we show their direct applicability in a simplified context. Chapter 8 has a more detailed analysis and complex usage of these features. Additionally, templates are a feature in Em that sidestep copy-and-paste reuse and provide a flexible mechanism to generate C code that may be necessary for applications. Lastly, to preserve modularity at all costs, we implement a limited form of inheritance and demonstrate a simple pattern that leverages the benefits of implementation inheritance without its implementation into the language.

To achieve a level of efficiency, currently available in systems developed in C, Em's high-level language features are translated to C. The translation process proceeds through steps that involve the generation of C header and source files in addition to Javascript configuration code. An execution context at build-time, provided by a Javascript interpreter, executes the configuration code to generate a single C source file that can be aggressively optimized by existing C compilers for numerous microcontroller targets.

Chapter 6

Non-Language Support for Application Development

In this chapter we present the design of additional features to support development of resource constrained systems applications that are enabled by and have not been incorporated into our language design. We see these additional capabilities as essential to fully supporting the creation of device applications in our defined domain. The ability to abstract hardware in our domain of highly variable hardware devices is essential. Support for runtime concurrency is necessary to enable efficient multitasking and straightforward implementation of complex behaviors. Simple development environment features are also important for supporting application development. We discuss the design of each of these aspects in the sections that follow.

6.1 Hardware Abstraction

Abstraction of hardware and low-level operating system details is commonplace in conventional computer systems. If to create a piece of software, every application programmer was required to write device drivers, operating system code and their application, the base of existing programmers, and existing software, would be significantly reduced. The uniform, high-level interfaces to low level device functionality along with a structured model for extending device functionality and adding new hardware support has been pivotal to growing the enormous base of programmers and software that exists today. Those interfaces and structured models do not exist in a standard way in our domain and are key for productivity and expressibility.

In order to let programmers focus on implementing their application functionality without concern for hardware specifics we have designed a hardware abstraction layer (HAL). The HAL has been designed as a package containing a set of module interfaces that specify core functionality for hardware. By abstracting the functionality into interfaces, implementations supporting many devices can exist while components that make use of the functionality can be developed and exist without being aware of implementation specific details. Specifically, we have created a package named `em.hal` in which we have defined the McuI, GlobalInterruptsI, GpioI, and UartI interfaces, among others for basic microcontroller initialization, enabling and disabling global interrupts, inter-

acting with general purpose I/O pins, and sending and receiving data over the UART, respectively. Our definition of these interfaces has been informed by [81].

Through the use of proxies, the defined interfaces allow the authoring of software modules such that they are independent of hardware specifics. The EventDispatcher module depicted in Figure 5.1 utilizes a proxy that implements the GlobalInterruptsI interface which allows necessary atomic operations to be performed by the dispatcher, such as pulling a new event to dispatch off the event queue. The hardware specific part of this operation happens independently of hardware details in the module. Additionally, the LedT template of Figure 5.10 shows the use of a Gpio module to allow for the creation of a module dependent on hardware, yet entirely definable independent of any low-level details. In both the examples given, the availability of interfaces that have implementations for different microcontrollers in conjunction with proxies has allowed code to treat the underlying hardware abstractly and function on any hardware that provides an implementation for the interface.

The abstraction provided by interfaces and the use of proxies also enables straightforward support for new microcontroller platforms, or any hardware device. Support for a new microcontroller unit (MCU) is achieved by simply implementing device-specific modules for all the interfaces the MCU supports. All existing code that has been implemented using proxies and these interfaces may now be usable on the new hardware. Adding software support for entirely new hardware and allowing it to be used abstractly

can be achieved by defining an interface for its functionality and providing a module implementation of it. For existing interfaces and modules that utilize those interfaces with proxies, the hardware has already been abstracted. To adapt an existing module to new hardware, a new implementation for an interface need only be bound to the using module. No other changes or low-level changes are necessary.

6.2 Runtime Concurrency

Cheap, readily available, highly constrained microcontroller devices are primarily used in applications that react to external stimuli whether from the environment, human or machine interaction. These systems must respond, often concurrently, to multiple sources of stimuli in a timely manner and need a form of runtime support to notify an application of the occurrence such events. This runtime support must provide a timely mechanism for concurrency in addition to being mindful of scarce resources such as RAM and power consumption. Since general purpose operating systems, which provide the mechanisms for concurrency in resource rich environments, do not fit the constraints of our domain, developers often must implement a model for concurrency with each application. To save programmers this effort, especially those without knowledge and experience in implementing operating system facilities, we have designed an instance of a model we feel is appropriate.

Two fundamental models of concurrency exist, a thread based model and a reactive model. Each of these models is discussed in more detail in the subsections below. While support for concurrency is essential we felt it would be limiting to incorporate any one model into the design of our language. While threads may be familiar for existing programmers, other models exist and should potentially also be usable. Multiple models can exist concurrently and their usage can be informed by application requirements. Our design puts in place a reactive model of concurrency to handle and process application events as they occur.

6.2.1 Threads

A thread based model for concurrency is most prevalent in resource rich environments. All modern and prevalently used operating systems - Linux, Windows, and MacOS - implement a thread based model for concurrency. In this model a program is allowed to create multiple paths of execution, each of which is called a *thread*, and can execute in parallel relative to other threads. Each thread can carry out computation independently of others although all threads of an application share a single process address space. Sharing and passing data between threads presents many synchronization problems and many mechanisms, such as locks, mutexes, semaphores, monitors, etc., have been created to allow this to take place. The operating system scheduler and

the underlying hardware determine the order of execution for threads and if any true hardware parallelism is performed.

The majority of programming languages used in resource rich environments support the threading model and programmers are familiar with this model. A platform independent standard defining an operating system interface to threads exists (POSIX pthreads) with implementations for all the popular operating system platforms. Some languages utilize the threading library of an operating system while others have incorporated it directly into the language with facilities and constructs for their usage. Programmers developing applications on resource rich systems are familiar with the threading model since its essentially the only model that exists to achieve concurrency. Furthermore, the model is quite simple as it allows a programmer to think in terms of multiple paths of execution and must only consider the intricacies and complexities of sharing data between threads.

With a threading model, each thread requires the use of an execution stack per thread which consumes runtime memory for each stack. The stack of a thread holds its execution context, namely the function's parameters, a pointer to the address the function must return to, and when not in an active state, the state of registers used by the thread. Storage of this runtime data requires memory and to ensure that execution contexts of threads do not intermix, each thread has its own stack.

Applications that use threads must frequently share data and other resources between threads. Synchronization mechanisms mentioned above enable a thread to wait for a particular resource or data to become available if it is occupied by another thread. When a thread must wait for a resource it is put to sleep and either wakes up periodically to check for its availability or it is awoken by another thread that has finished using the resource. When a thread is put to sleep, it makes the processor available for another thread to execute. To run the available thread, a scheduler must remove the execution context of the sleeping thread, save it in all in memory, and then replace the execution context of the thread that is ready so it may pick up from the last place it left off. This process is called a context switch.

A context switch is a RAM intensive operation that lends itself to higher power consumption. Anytime a thread must wait for a resource to become available, or wait for data to be produced by another thread, a context switch is necessary. This is necessary so threads that are ready to execute may have a chance to do so. During the context switch data must be saved into and then read out of RAM and the accessing of RAM is a power consuming operating, not to mention the processing time required to conduct the context switch. On resource rich platforms both the memory and execution time this takes is insignificant. However, in resource constrained environments where power consumption is potentially a critical resource and CPU cycles are not abundant, the cost of a context switch is high.

The model of threads for concurrency may be more familiar to programmers however the relatively inefficient use of resources makes the model less attractive in a resource constrained environment. In resource constrained environments it is often the case that an action only takes place when a specific event has occurred or stimulus received. It is the case then, that in this environment a threading model will frequently have threads that are waiting on a resource. Since each thread consumes already scarce RAM, having multiple inactive threads consuming memory is one significant pitfall of the threading model in our context. Additionally, when threads wait on resources, a context switch must occur and context switches consume some power due to the reading and writing of RAM. If threads are frequently waiting on resources, context switches are also occurring frequently resulting in an overall higher power consumption of the system.

As a result of the above mentioned pitfalls of the threading model, we have decided to not incorporate any constructs into our language design that impose an execution model on the programmer. Moreover, we have decided that a threading model is not the most appropriate model for concurrency in resource constrained systems and have designed instead a method of concurrency based on a reactive model.

6.2.2 Reactive Concurrency

Reactive systems [11, 75] are ones where actions take place in response or in reaction to external or internal stimuli. These systems often remain dormant, or inactive until precisely the moment of stimulus which then causes a precise reaction that may involve single or multiple actions to take place. In these systems, stimuli can arrive at any point in time and potentially simultaneously. Furthermore, some reactions may require non-trivial processing to take place before the inactive state of the system can be reached again. Additional stimuli could enter the system at the same time as a processing of a previous reaction is taking place.

A reactive model of concurrency supports the above behavior. In this model, applications are structured in terms of actions that must take place only once a particular event occurs. The application as a whole then sits in an idle state until an event is encountered. Once it is encountered, an event dispatching scheduler invokes the appropriate handler for that event. This structure makes the system entirely available to respond to any event as it occurs. If multiple events occur simultaneously a queue of events is built up and as event handlers complete, others are invoked, until the system is idle again. The primary source of events is hardware interrupts which triggers an interrupt service routine (ISR). The ISR is invoked asynchronously at any point in time, even if an event handler is currently running, and execute in a different context. The execution context for interrupt service routines is created by the compiler and takes

care saving and replacing any system registers used by any process running before the interrupt occurred. The asynchronous handling of interrupts and the event dispatcher that invokes event handlers as they occur is what provides the concurrency support for the behavior of reactive systems.

The reactive model is better suited for our domain of applications and more resource efficient than the threading model. Applications in our domain continually receive and must respond to stimuli from their environment. Moreover, applications require short, non data-intensive processing that must take place while not compromising the ability of the system to respond to more stimuli. The reactive model in general is therefore more fitting for our domain. Additionally, instead of having threads of execution waiting on resources to become available, the reactive model causes code that services a resource to be invoked only when the resource is ready to be serviced. This removes the need to have threads as the basis for concurrency and with that, removes the additional runtime resources that a threading model requires. If the immediate response to stimuli takes place within an interrupt context (i.e in an interrupt service routine) while further processing occurs outside of interrupt context, a context switch as described in the threading model need not take place. Power savings, necessary by applications in our domain, is better achievable with the reactive model than the threaded one.

The reactive model is less familiar to programmers although it is used extensively in graphical user interface programming (GUI). Since resource rich environments uti-

lize a threading model, fewer programmers have naturally been exposed to a reactive method of programming. However, the execution model used in GUI programming is very similar to the reactive model. Graphical interfaces usually function with an event system where events are triggered in response to user input. Applications are structured and written such that event handlers for user actions are invoked when the user triggers an event. Short, non-blocking, event handlers are invoked by the event system sequentially and as a result, the perception of concurrency is achieved.

We have chosen a reactive model of concurrency for providing concurrency support to users because of its efficient use of resources and congruency with applications in our domain. The fact that applications using a reactive model remain idle until events occur resembles the true nature of resource constrained systems. This congruency influenced our decision to adopt this model in our design. The model furthermore does not require the utilization of multiple function stacks to maintain the multiple execution contexts and saves critical resources. Moreover, not conducting context switches between threads reduces the overall system RAM usage, which contributes to a power savings overall and further influences our decision to adopt the model. Lastly, despite the fact that fewer programmers are familiar with reactive models of concurrency, its presence in GUI programming acknowledges the fact it is within the grasp of programmers in general.

6.3 Development Environment Support

In considering the design of a development environment to support the broad range of devices, applications and developers of our domain a number of important features must exist. Today, the tools for application development are highly fragmented. The available development environments often support only one manufacturer's microcontroller. The environment itself does not have the tools that are used to ultimately download the compiled code onto the microcontroller integrated in; separate software tools are used. Moreover, the environments only directly support a subset of the code downloading utilities. The result is that the typical expert developer's workstation has a collection of environments installed - one for each manufacturer, numerous downloading utilities that may or may not interface to the environment and various hardware downloading tools that ultimately interface with the actual hardware.

Any design of an environment for our domain must support the holistic integration of numerous manufacturer's devices, enable extant downloading utilities to plug into the environment, and support a large number of the download hardware tools. Such support is necessary since the typical application development cycle involves a rapid code-compile-download and test cycle. Namely, software must be incrementally written, built for a specific target, downloaded onto the device and then its execution tested. The fragmentation of the extant tools not only constricts the range of devices usable

in a single environment, it also makes this often repeated cycle unnecessarily slow and complicated. For any suitable design, the development environment must not be specific to one manufacturer's device. Additionally, for each device the environment supports it must also support the integration of the necessary target compiler and code downloading utilities.

The integration should be such that a single user action, such as a button press, can invoke the compilation of an application and seamless downloading of the application onto the hardware. The presence of such a feature in an environment has broadened the range of people capable of getting started with development. The Arduino platform, for example, integrated the compilation and code downloading process, making it accessible from a single click of a button. Users plug their hardware into the computer and can instantly get code working on their devices. The setup and integration of such tools into a development environment is not a trivial task and can be a stumbling block or even a barrier to entry for many without significant technical expertise.

In addition to facilities that ease and expedite a code-compile-download-test development cycle, in future designs, the development environment should also facilitate the discovery of usable software content for applications and a graphical method of development. As development proceeds, a facility within the environment should be available to guide the developer to software content related to what they are developing. Both finding reusable content and accessing API documentation should be simple

and direct since many developers, both novice and expert, depend on such content for functionality, ideas and a rapid development process. An environment designed to support the discovery of such content and allow its rapid integration into applications has the potential to enhance the productivity and development capabilities of a broad audience of users.

The application creation process, especially one utilizing a high degree of reusable software, involves both the creation of new components and integration of existing ones. A graphical representation of such software, with the ability to visually select, interconnect, and configure components as a mode of development is a desirable capability in an environment intended to support a broad audience of developers. Extant visual development environments [65, 42, 67, 51] have shown that domain experts with less programming skills are capable of assembling and creating sophisticated applications. Likewise, experts are capable of creating new features and functionality into such environments that are both necessary for their applications while simultaneously being reusable and accessible to other kinds of developers. The design of a development environment with the above mentioned support can leverage and bring together the language and non-language support mechanisms available to enable the potential for rapid application development in our domain.

Chapter 7

Hardware Abstraction and Concurrency in Em

In this chapter we discuss our implementation in Em of the additional non-language features for supporting device application development presented in the previous chapter. Specifically, we discuss the implementation of a hardware abstraction layer, an event dispatching scheduler to support concurrency in a reactive model, and features added to a development environment to assist the development process.

7.1 Hardware Abstraction

Em's hardware abstraction layer has been implemented as a set of interfaces that define device specific functionality. We currently have modules that implement in low-level, device specific details, the interfaces that match a microcontrollers capabilities

for the Atmel ATmega128 and ATmega168 [8], TI MSP430 [58], NXP LPC2138 [60], LuminaryMicro LM3S811 [55] and LM3S6965 [54] Microcontroller units (MCUs).

The hardware abstraction layer consists of a package of interfaces named `em.hal` which contains definitions for MCU startup initialization, timers, general purpose input/output pins, interrupt sources, serial peripheral interfaces such as UART, SPI and I2C. Other interfaces that do not deal with MCU integrated peripherals have also been defined, for example LEDs, Buttons, and an accelerometer. Our selection of functions for some of these interfaces was informed by [80]. Figure 5.4 shows part of the interface defined for a microcontroller GPIO pin and the other interfaces mentioned are similarly defined for their intended purpose.

The hardware abstraction layer along with the module implementations for the above mentioned microcontrollers has enabled us to create both full applications, as well as simple test applications that are entirely reusable and portable across all the devices. As an example, the BlinkP program in Figure 5.3 is a simple application to blink an LED every 500 milliseconds. The program is fully portable, yet it controls a physical LED wired to a microcontroller pin and utilizes a hardware based timer that is integrated into the microcontroller. Through the use of the hardware abstraction layer we implemented and the use of a composite that represents the hardware module's available on the device's printed circuit board, the program is utilizing hardware abstractly. The more complex applications discussed in a later chapter all use the same

hardware abstraction layer, proxies and composites to abstract away hardware and enable the creation of fully portable, reusable modules and applications.

7.2 Runtime Concurrency

To provide runtime support for concurrency in Em applications, we have implemented modules that represent hardware interrupt sources and a hardware-independent event dispatcher that provides a portable event type, event posting capability, an event queue and a scheduler for event handler invocation. Figure 7.1 shows the interface defined for an interrupt source, Figure 5.1 shows part of the event dispatcher module specification, and Figure 5.2 shows the event dispatcher's `dispatch` function.

```
interface InterruptSourceI {

    type Handler: Void()

    host function setHandlerOnHost( h: Handler ): Void

    function enable(): Void
    function disable(): Void
    function clear(): Void
    function isEnabled(): Bool
}
```

Figure 7.1: Interface for interrupt source modules.

A module representing an interrupt source encapsulates the interrupt service routine which gets called by the microcontroller and has a minimal interface to read, clear,

enable and disable the interrupt source. Every microcontroller has a different mechanism for registering interrupt service routines in its interrupt vector. The interface therefore only defines a function to register the interrupt handler, and leaves it to the implementing module to specify exactly how the handler is registered in the interrupt vector. Users of the module call the `setHandlerOnHost` function to register the interrupt handler with the module. This happens in a host function so the vector can be statically assigned at build time, allowing the implementing module an opportunity to carry out any additional operations it may need to do so, at build time, for example, determining if a handler has already been registered and taking appropriate action if so. This implementation allows modules to treat and interact with interrupts as any other module.

Functions invoked by interrupt handlers, unlike other module functions, execute in interrupt context and are highly sensitive to prolonged computation. Long running operations conducted within an interrupt handler can compromise ability of the system to respond to additional stimuli. As such, only the minimal operation necessary to service the interrupt is conducted at interrupt time. As discussed below, the interrupt handler often sets in motion additional actions by posting an event to the event dispatcher, which will invoke the actions outside of interrupt context.

The event dispatcher in Figure 5.1 defines an `Event` type which other modules use to interact with the event system. By declaring an instance of the event type in

their specification and initializing it with a module function that will be invoked by the event dispatcher, that is the event handler, modules can utilize the event system for concurrency. Modules pass the event type they declared to other modules that will post the event when a particular action of interest has occurred. The BlinkP program in Figure 5.3 declares an event in its private specification and sets its function `blink` as the event handler. The millisecond timer `TimerMilli0` is used to trigger the `blink` event handler every 500 milliseconds. The event is passed to the `TimerMilli0.start` function so that the timer can post the event to the dispatcher by invoking the event's `post` function.

The TimerMilli0 module used in BlinkP is internally using the hardware timer peripheral and its interrupt source to providing the timing. The TimerMilli0 module contains the interrupt handler for the timer interrupt and when it is triggered, it posts the event received by the BlinkP module's call to the `TimerMilli0.start` function. Two things are occurring concurrently in this example, the timer peripheral is counting milliseconds while every 500 milliseconds the BlinkP module toggles its LED. If the MCU of the application's hardware had multiple timers, the BlinkP module could utilize multiple timers, multiple events and multiple event handlers to have more things occur simultaneously.

The event dispatcher is implemented as a statically sized, non-prioritized FIFO queue of events that contain function pointers to event handlers. When modules post an

event it is pushed onto the dispatchers FIFO and the dispatcher will invoke the events handling function at a later time. Since all event handlers are non-preemptable by other handlers, they must be non-blocking and short to allow other handlers to execute in a timely manner. Handlers that need more time for computation, must break up the computation into smaller pieces and advance processing by posting events to themselves. Event handlers, are however preemptable with respect to interrupt handlers. At any point in time an incoming interrupt will stop the currently executing event handler, if one is executing, and jump to the interrupt handling function. Compilers tend to set up an execution context for interrupts that either utilize a different set of processor registers than the previously executing code, or it generates additional code to save and replace the registers used in the interrupt handler. This action by the compiler removes the need to perform a context switch as performed in a threading model.

Prioritized queuing and a scheduler with preemption has also been implemented in Em. The prioritization of events allows some module functions to take precedence relative to others, and potentially preempt an operation, if it should occur before other operations. This prioritization and preemption is necessary when there is a hierarchy of operations that must take place. For example, an averaging filter that uses data from an analog to digital converter is dependent on the availability of data to conduct its operation. If the filtering function is currently executing, and new data is available for buffering, it make sense to preempt the filter in order to conduct the buffering opera-

tion, returning to the filtering once the data has been buffered. If a large amount of data must be buffered, it should not be carried out in an interrupt handler, instead the interrupt handler should post an event to invoke the buffering function outside of interrupt context. With preemption, if the filtering operation is currently executing, it will be stopped and the buffering operation conducted.

7.3 Development Environment Support

The typical developer of a device application incrementally writes code, downloads it to the microcontroller on the device, and tests it. Given the many incompatibilities of downloading utilities with development environments and devices, we have attempted to integrate into an existing environment, namely Eclipse, the ability to write Em code for multiple devices, compile it and download it onto the microcontroller seamlessly.

Our plugin enables a simple code, download, test development cycle. The plugin implements a text editor with syntax highlighting capability and a file browser. A configuration file contains the specification of the target microcontroller, the compiler it requires, and any special compilation flags necessary for building code. By setting the configuration file for the board used the plugin knows which compiler to use and how to invoke it. The file browser provides the ability to create packages for organizing modules and Figure 7.2 shows some packages with microcontroller specific modules

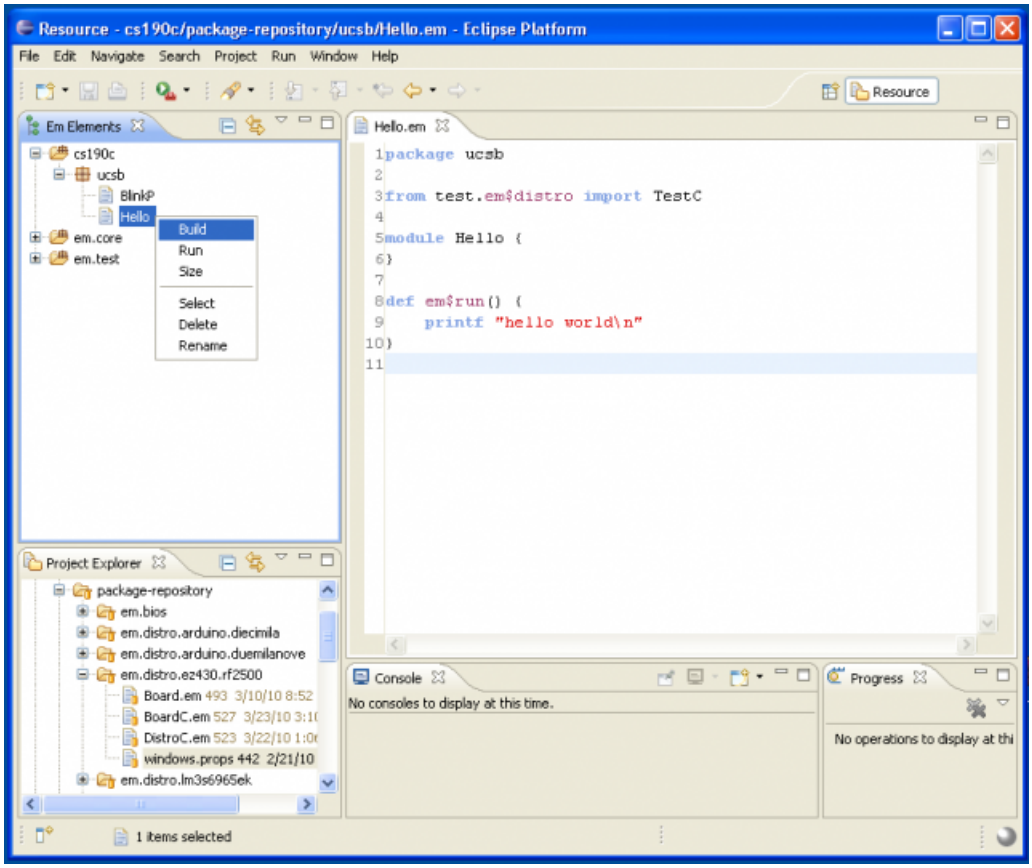

Figure 7.2: Simple capability to build and download applications.

such as `em.distro.arduino.duamilanove`. By selecting a module, a menu enables the building of the code and loading it onto the device. Only modules that are complete applications, distinguished by the presence of the `em$run` function, can be loaded onto the target. With the help of the mentioned configuration file, code can be easily built for and downloaded onto different device targets.

Once the code is on the microcontroller, it is often helpful to see output from the executing process on the host. To enable this, we have incorporated a console into the plugin that can interface with the target device and display characters output by the application. An additional command the plugin contains is to build, load, and open the console such that an application is compiled, downloaded to the target, and its output is immediately viewable on the console. This makes the repetitive actions in the iterations of application development simple, direct and fast.

7.4 Comparison and Contrast

The use of real-time kernels in resource constrained systems is a common approach to incorporating concurrency into applications. The predominant model of concurrency for these systems is a threaded one with FreeRTOS [24] being a frequently used, open-source solution. The system itself only provides a preemptive multi-tasking scheduler. Despite the relative inefficiency of a threading model as compared to a reactive model of concurrency, threading is more familiar to developers and therefore more often used. Integration of existing kernels into applications for specific hardware platforms is left as a task for the developer themselves. This complex and often time-consuming task leaves such kernels within reach of only the most experienced developers. Manipulating

header files to match system resources and incorporating hardware access to global interrupts are some of the steps required to make use of such kernels.

While some kernels provide a model for concurrency, they lack additional support for hardware abstraction and device driver models. Developers must implement their own hardware abstraction mechanisms and put in place a methodology for integrating hardware into designs. Applications that succeed in doing so are usually incompatible with other such systems since their designers do not have a standard to adhere to.

TinyOS along with nesC, support the concurrency and reactivity requirements of Em's intended applications. A major difference between Em and nesC/TinyOS is that Em does not integrate a specific execution model into the language. An asynchronous event based concurrency model features identical to the one present in nesC is implemented in Em allowing other models to be implemented if necessary. Detailed comparison of the nesC and Em concurrency mechanism is discussed in chapter 8.

Finally, TinyOS provides hardware abstraction and a structured model affording support for differing microcontrollers and hardware. While the current offering of microcontrollers of differing architectures is plentiful, the mainline microcontroller support in TinyOS is primarily for the Atmel AVR and Texas Instruments MSP430 families of processors. TinyOS's hardware abstraction layer is well defined in documents, although inspection of application source code shows that programmers do not adhere to the layering defined in the model. Ultimately, with hardware abstraction, nesC and Em

are similar in that the language model affords necessary development support without integrating capabilities into the language itself.

7.5 Summary

Non-language features that are enabled by, but not incorporated into the programming language we have designed, are important to support application development in our domain. In this chapter we presented the implementation of these features, namely, a hardware abstraction layer to raise developers level of abstraction above hardware specific details, a runtime mechanism to support concurrency in a manner suitable for our domain that does not compromise critical resource utilization, and features of an integrated development environment that enable a rapid code, download and test cycle.

All the features mentioned are critical to broadening the audience of people capable of developing resource constrained device applications. At the same time, we have explicitly decided to implement these features in our language, as opposed to implement them into our language. The decision was motivated by the fact that many options exist and could potentially be viable under particular circumstances and we felt the need to allow them to exist without narrowing the scope or capabilities of our designed language.

Chapter 8

Demonstration and Evaluation

In this chapter, we investigate and demonstrate, using real and sophisticated programs written in Em, how well the language features we have chosen facilitate reuse and portability, integration of legacy code, refactoring, and efficient use of device resources. To enable this, we implement and consider applications of varying complexity for a number of off-the-shelf hardware platforms, these include

- A low-power Texas Instruments MSP430-based wireless sensor node [33],
- A LuminaryMicro ARM Cortex-M3 system with Ethernet support [54],
- An Atmel AVR based Arduino hardware [5], and
- An NXP LPC2100 ARM device [60].

The applications we consider range from an embedded web server utilizing a TCP/IP stack, to device drivers for peripherals external to the microcontroller, to a number of frequently used application building blocks. We evaluate our implementations on the

basis of application footprint relative to related systems. Additionally, where applicable, we compare the methodology of development in Em with that of C and other popular systems used for developing resource constrained device applications.

We find that Em provides developers with many of the programming advantages facilitated by languages for more resource-rich systems (C++, Java, Python, Ruby). It does so by leveraging the existing capabilities of C and by further incorporating a subset of features from these languages (modularity, composition, inheritance, separation of concerns, separation of interface from implementation, support for popular design patterns, and others) and combining them with novel support for automatic source generation (legacy and generic), opaque types, and a unified configuration / target development language system. Together, these features enable reuse and portability of code that can be shared and extended by distributed developers for a wide range of devices and components. We show that despite the availability of these features in Em, application resource utilization is not compromised and comparable, if not better, than other systems in our domain.

Lastly, we evaluate the ability of relative novices to learn our language and to implement non-trivial applications. We articulate our experience teaching an undergraduate course for computer science and computer engineering students in modular embedded systems software development utilizing our language. We report results of surveys con-

ducted to gain insight into how well students were able to understand the concepts of the language and realize applications.

8.1 Reusability & Portability

In this section, we conduct an evaluation of the reusability and portability of software written in Em. We investigate the reuse of software supporting a hardware component that is not integrated into the microcontroller itself. Such device drivers are common and the ability to author them in a reusable, portable fashion rarely exists today. We then investigate the reusability of software functionality that is not tied to a particular piece of hardware, yet depends on hardware functionality for its purpose. Specifically, we port an existing TCP/IP networking stack to Em showing the architectural benefits of doing so along with demonstrating the software's reuse on hardware platforms with communication transport mechanisms. We show how our implementation does not decrease the performance of the original code written in C. Lastly, we evaluate the ability to reuse existing C code within Em applications. Such capability can expedite development, where necessary, by not having to rewrite entirely existing functionality in Em.

8.1.1 Reuse of software supporting hardware components

We first investigate the reuse and portability of code that Em enables using an implementation of a device driver for a radio transceiver. For this experiment, we use a development board [33], representative of low-power wireless sensor network nodes, with a low-power microcontroller and the popular ChipCon CC2500 2.4GHz radio transceiver [32]. Since the radio transceiver can be used in many hardware devices and interfaced to microcontrollers from different vendors, the goal of this experiment is to create a reusable, portable driver for the transceiver that can be easily adapted, without modification to the driver, to different board configurations as well as different application configurations.

The CC2500 radio transceiver interfaces to a microcontroller via the serial peripheral interface (SPI) port and two general purpose I/O lines. The SPI port hardware is integrated into the microcontroller itself. The microcontroller configures and interacts with the radio via its SPI port. Multiple SPI ports could be resident on a microcontroller and many microcontroller pins can interface with the I/O lines of the transceiver. Therefore, for the transceiver driver to adapt to different hardware configurations, we must decouple its use of low-level hardware from the driver's implementation.

To achieve this decoupling in Em, we use two interfaces defined in our hardware abstraction later, namely SpiI and GpioI. Within each interface, we define the essential functionality of each peripheral, informed by [81]. Implementations for controlling

each peripheral are written for the development board's microcontroller, in this experiment, a Texas Instruments MSP430.

```
import CC2500Registers as Registers
import CC2500Context as Context
import ICC2500Configuration

module CC2500 {

    proxy BusyWait implements BusyWaitI
    proxy GDO0 implements GpioEdgeDetectI
    proxy GDO2 implements GpioEdgeDetectI
    proxy Spi implements SpiI
    proxy GlobalInterrupts implements GlobalInterruptsI
    proxy Configuration implements CC2500ConfigurationI
    ...

    host function setAddressOnHost( a: UInt8 ): Void
    function setAddress( a: UInt8 ): Void
    function reset(): Void
    function setReceiveMode(): Void
    function setTransmitMode(): Void
    ...
}

def em$construct() {
    Configuration.setContextValuesOnHost()
    ...
}
```

Figure 8.1: Portion of CC2500 radio module.

Figure 8.1 shows an excerpt of the public specification of the CC2500 driver module. Defined at the top, is a collection of proxies the driver depends on, including SPI port and Gpio proxies which implement their corresponding interfaces. Clients of a proxy, in this example, the radio driver, can be sure that at some point before an application is built, a module providing an implementation for the proxy's interface will exist and be bound. Therefore, the radio driver can be implemented without knowledge of

microcontroller-specific details. Furthermore, since Em makes it possible to bind different modules implementing the SPI and Gpio drivers to the driver, different physical configurations of the hardware can be supported with a single driver implementation.

```
import GlobalInterrupts
import Mcu
import SpiUCB0
...
import EdgeDetectP2_6
import EdgeDetectP2_7
...

from em.parts import LedT {activeLow: false} as RedLed
from em.parts.CC2500 import CC2500 as Radio

composite BoardC {
    ...
    export Radio
    export RedLed
    export SpiUCB0
    ...
}

def em$preconfigure() {
    seal Radio.GDO0 as EdgeDetectP2_6
    seal Radio.GDO2 as EdgeDetectP2_7
    seal Radio.GlobalInterrupts as GlobalInterrupts
    seal Radio.Spi as SpiUCB0
    seal RedLed.Pin as P1_0
}
```

Figure 8.2: BoardC composite for wireless device.

Figure 8.2 shows the BoardC composite representing the development board's physical hardware configuration. The radio's SPI proxy is bound to the microcontroller's SPI0 port and the radio's GDO0 and GDO2 I/O lines are bound to pins EdgeDetectP2_6 and P2_7, respectively. The proxies that define the radio driver's hardware interconnection to the microcontroller are now configured. Should a different piece of hardware

using the CC2500 radio be physically wired differently, only the driver's proxy bindings must change and the driver itself need not be modified.

With time and use on different hardware the driver will become stable and trustworthy for use on differing platforms in variable configurations. Hardware vendors, engineers, and others can develop such code and make it available for others to use thereby obviating the need for clients of this code to understand the low-level details of the radio. Less experienced programmers can simply can use the radio within their applications.

Such applications configure the radio according to their needs. Node address, power levels, communication frequencies and other critical settings can vary, and often must, per application. To support this variability without requiring modification to the driver itself, we structured the application such that it uses an interface for configuration modules. In our example we provide an interface called CC2500ConfigurationI. The interface defines one host function: `setContextValuesOnHost`. The module CC2500Context comprises all of the driver's configuration options.

Figure 8.3 depicts the interface and module. The CC2500 module contains a proxy, shown in Figure 8.1, that implements the CC2500ConfigurationI interface and in its `em$construct` intrinsic, uses the proxy to call setContextValuesOnHost. This function sets all of the application-specific configuration values as shown in Figure 8.4.

```
interface CC2500ConfigurationI  {
    host function setContextValuesOnHost(): Void
}

module CC2500Context {

    config addressSize: UInt8
    config radioAddress: UInt8[]
    config broadcastAddress: UInt8[]

    config logicalChannels: UInt8[]
    config numLogicalChannels: UInt8

    config powerLevels: UInt8[]
    config numPowerLevels: UInt8
    ...
}
```

Figure 8.3: Module and interface used for radio configuration.

Each instance of an application can now create a module that implements the CC2500ConfigurationI's interface. Such a module implements the interface to set the application-specific configuration as shown in Figure 8.4. This module can then be bound to the driver's proxy. During its runtime initialization, the CC2500 driver reads and sets configuration values from the CC2500Context module. The module structure above now allows individual applications to define settings as necessary without modification to the driver.

In C, the common way to support both variability in hardware and application configurations is through C-preprocessor macros, `#define` directives, and code wrapped in `#ifdef/#endif` blocks. Figure 8.5 shows a part of the original radio driver code written in C, not shown in the figure is how additional preprocessor directives are inter-

```
from em.parts.CC2500 import CC2500Context as Context

module TSenseCC2500Config implements CC2500ConfigurationI {
}

def setContextValuesH() {

    Context.addressSize = 1
    Context.radioAddress = [0xaa]
    Context.broadcastAddress = [0xff]

    Context.powerLevels = [0x46, 0x97, 0xfe]
    Context.numPowerLevels = 3

    Context.logicalChannels = [0x03, 103, 202, 212]
    Context.numLogicalChannels = 4
    ...
}
```

Figure 8.4: Application-specific radio configuration.

leaved both within and across function boundaries. Em significantly improves readability (amount algorithmic code can be viewed per screen of text), maintainability (ability to understand and correctly extend code), and code stability (localized change) over this alternative by avoiding all use of these mechanisms. The more variation that exists, the more macros and directives one encounters upon reading the code. Understanding what code is ultimately compiled in to a binary and where to make modifications demands close inspection of the code. Modification of these directives, often defined in drivers themselves, changes the overall modification date of the driver effecting code stability in the face of inevitable variation. The CC2500 driver developed in Em has been created in a reusable and portable fashion allowing individual applications to define configurations settings without requiring modification of the original driver source.

```
/* ---------- Radio Abstraction ---------- */
#if (defined MRFI_CC1100)
#define MRFI_RADIO_PARTNUM          0x00
#define MRFI_RADIO_MIN_VERSION      3

#elif (defined MRFI_CC1101)
#define MRFI_RADIO_PARTNUM          0x00
#define MRFI_RADIO_MIN_VERSION      4

#elif (defined MRFI_CC1100E_470)
#define MRFI_RADIO_PARTNUM          0x00
#define MRFI_RADIO_MIN_VERSION      5

#elif (defined MRFI_CC1100E_950)
#define MRFI_RADIO_PARTNUM          0x00
#define MRFI_RADIO_MIN_VERSION      5

#elif (defined MRFI_CC2500)
#define MRFI_RADIO_PARTNUM          0x80
#define MRFI_RADIO_MIN_VERSION      3
#else
#error "ERROR: Missing or unrecognized radio."
#endif
```

Figure 8.5: Part of radio driver abstraction written in C.

8.1.2 Reuse of non-hardware supporting software

We next demonstrate how Em supports the creation of reusable software that is not hardware specific although depends on hardware specific functionality. This demonstration furthermore shows how Em can be used to refactor existing C code to avoid redundancy, separate concerns, and promote code stability by enabling variability without requiring extensive modifications to the code base. Common error prone C constructs, such as goto statements and conditional macros, which also make code difficult to read, debug and maintain were entirely eliminated. For this investigation we ported μIP [21],

a popular RFC compliant TCP/IP stack for resource constrained microcontrollers to Em. On top of the TCP/IP stack we implemented a simple HTTP server which serves a static HTML page.

Since RAM is often a microcontroller's most scarce resource, μIP's RAM usage is configurable at compile time and is determine by various features of the stack such as the size of its protocol buffer, the number of concurrently active TCP connections, the ability for the stack to open connections to remote hosts, and the number of listening TCP connections. μIP contains a single statically allocated packet buffer for both incoming and outgoing data and depends on an underlying device specific transport peripheral, such as an Ethernet controller, to send and receive packets. Some microcontrollers may have sufficient RAM to support a different buffering policy, which may lead to more efficient networking. For such devices, the authors of μIP provide lwIP[20] which has a more sophisticated packet buffer policy in addition to more memory-intensive protocol features. While μIP and lwIP share identical functionality in many places, they have been implemented and are maintained as two separate code bases.

With Em, we have been able to provide a single implementation of the TCP/IP stack that enables variability in transport layers and packet buffer management policies without modification to other aspects of the stack. Furthermore, the implementation in Em modularizes the protocol code, separating each protocol into its own module,

which enables different versions of the protocol to be used in the stack given particular memory constraints. Finally, the stack's implementation completely does away with goto statements and conditional macros, which appear heavily in the C implementation of μIP, making the code easier to read and maintain. Figure 8.11 shows part of the μIP packet processing function illustrating frequent use of goto statements and conditional macros.

```
interface TransportI {

    function disable(): Void
    function enable(): Bool
    function getHostAddress( addr: UInt8[] ): Void
    function getHostAddressSize(): UInt8
    function getHeaderLength(): UInt8
    function getPacket( buf: BufferManager.Buffer& ): IP.Packet&
    function receive( buf: BufferManager.Buffer& ): UInt16
    function send( buf: BufferManager.Buffer& ): Int16
    ...
}
```

Figure 8.6: A transport layer interface.

To enable variability in the transport layer we created an interface, TransportI Figure 8.6, containing the features required by transport layer modules. A module called TransportManager, Figure 8.7 bottom, contains a proxy of type TransportI and is responsible for coordinating the reception and transmission of data between the transport layer and the stack. By binding the proxy in TransportManager to different implementations of transport modules the upper layers of the stack can send and receive data on different mediums such as wireless radios, serial ports etc. without modification.

```
module TransportManager {

    proxy Transport implements TransportI

    function enableTransport(): Void
    function disableTransport(): Void
    function getTransportHeaderLength(): UInt8
    ...
}
```

Figure 8.7: Part of a transport manager module specification.

The TransportManager also handles the buffers for data that is sent and received by the application. To facilitate different buffer management policies to exist and to provide flexible configuration of the stack given a specific policy, we employ the following design. We create a BufferManagerI interface (Figure 8.8) to define a Buffer type and the functions of the manager. Only one buffer manager must exist in any application yet we want to allow different representations of the buffer type. To enable this, we use the Provider pattern [64].

A module named BufferManager which we present in Figure 8.9, defines all the functions of the BufferManagerI interface and a representation for the Buffer opaque type. A proxy within BufferManager implements the BufferManagerProviderI interface. This interface inherits the BufferManagerI interface and the BufferManager forwards all calls to its functions to the provider proxy in Figure 8.9. We can now create a concrete BufferManagerProvider module with the implementation of the management

policy an buffer representation. In our implementation in Figure 8.10, we create a SingleBufferManager that statically allocates a single buffer of configurable length.

```
package em.net.transport

interface BufferManagerI {

    type Buffer: opaque {
        function getBuffer(): UInt8[]
        function getSize(): UInt16
        function getDataLength(): UInt16
        host function initH( size: UInt16 ): Void
        function setDataLength( len: UInt16 ): Void
    }

    function freeBuffer( buf: Buffer& ): Void
    function getEmptyBuffer(): Buffer&
    function hasEmptyBuffer(): Bool
}
```

Figure 8.8: Packet buffer manager interface and opaque type definition.

Our stack implementation now supports variability in transport mediums and buffer management policies in a single code base without requiring modification of the upper layers of the stack. In contrast, the μIP implementation in C would need to be radically modified to support multiple packet buffers since it relies entirely on a single, globally accessible buffer, with global length variables modified in various points of the code. Regarding transport mechanisms, μIP itself does not contain any transport specific code, however it does integrate ARP into the main packet processing function which assumes an Ethernet transport. The implementation in Em removes ARP from the stack functionality and abstracts that functionality into the transport layer.

```
module BufferManager implements BufferManagerI {

    proxy Provider implements BufferManagerProviderI
}

private {

    def type Buffer: struct {
        base: Provider.Buffer
    }
}

def Buffer.getBuffer() {
    return this.base.getBuffer()
}

def Buffer.getSize() {
    return this.base.getSize()
}
...
```

Figure 8.9: Packet buffer manager modules.

μIP's packet processing algorithm is contained in its entirety in a single function. The function is heavily dependent on conditional macros to selectively include or omit parts of the code and as a result compromises the readability of the code. `goto` statements are heavily used in the function and it is claimed by the author of μIP that this is for optimization purposes however both readability and maintainability of such code is laborious and error prone. A sample of the μIP `process` function is shown in Figure 8.11. Furthermore in the name of optimization, the code relies on global variables for temporary variables, connection states and packet buffer lengths so multiple local variable declarations and parameter passing to functions can be avoided. Finally the

```
module SingleBufferManager implements
    BufferManagerProviderI { }

private {

    def type Buffer: struct {
        dlen: UInt16
        buffer: UInt8[]
    }

    var buf: Buffer
    var bufUsed: Bool
}

def Buffer.getBuffer() {
    return this.buffer
}

def Buffer.getDataLength() {
    return this.dlen
}
...
```

Figure 8.10: Packet buffer manager implementation with single buffer policy.

packet processing for various protocols are all interleaved into the one function with parts related to UDP processing interleaved with that for TCP and ICMP.

The implementation in Em separates the concerns of each protocol into individual modules where, for example, all TCP processing and connection state management only appears in the TCP module. The stack's packet processing function no longer interleaves different protocol processing code. Protocol and connection state is not shared between any modules in the form of global variables, and large functions with goto statements were replaced by smaller, more manageable functions. Overall, Em

```
void uip_process(u8_t flag)
{
#if UIP_TCP
  register struct uip_conn *uip_connr = uip_conn;
#endif /* UIP_TCP */

#if UIP_UDP
  if(flag == UIP_UDP_SEND_CONN) {
    goto udp_send;
  }
#endif /* UIP_UDP */

  uip_sappdata = uip_appdata = &uip_buf[UIP_IPTCPH_LEN + UIP_LLH_LEN];

  /* Check if we were invoked because of a poll request for a
     particular connection. */
#if UIP_TCP
  if(flag == UIP_POLL_REQUEST) {
    if((uip_connr->tcpstateflags & UIP_TS_MASK) == UIP_ESTABLISHED &&
       !uip_outstanding(uip_connr)) {
     uip_flags = UIP_POLL;
     UIP_APPCALL();
     goto appsend;
    }
    goto drop;

  ...
}
```

Figure 8.11: A sample of the μIP process function.

enabled the μIP functionality to be written in a way that avoids many maintainability pitfalls and promotes more readable, understandable code.

We now compare the resource consumption and performance of our web server application implemented using the ported μIP stack in Em with the same application implemented in C, using the original unmodified μIP stack. Both applications were implemented for the Luminary Micro LM3S6965-EK evaluation board and compiled

µIP HTTP Server	**Flash**	**RAM**	**Request**
Optimized for size (-Os)	(bytes)	(bytes)	(ms)
Em	9600	2456	4.0
C	9836	2444	4.5
Optimized for speed (-O3)			
Em	13880	2456	4.0
C	11440	2444	4.0

Figure 8.12: μIP web server footprint and request duration.

using the same gcc compiler (version 4.4.1) for the ARM Cortex-M3 architecture. We report RAM and Flash memory footprints when the code is optimized for performance (using -O3) and optimized for size (using -Os). The performance of each application we report is the average duration of an HTTP request from a sample of 100,000 requests. In our measurements both application implementations had the same stack features selected. Specifically, the stacks had the same protocols enabled, used a statically assigned IP address, did not allow active TCP connections from the device to remote hosts, had a single packet buffer of identical size, and the same number of maximum connections and listening ports.

When we build the application and optimize for size, the Em application memory footprint is 2.4% smaller than the C implementation, uses 12 bytes more RAM, and performs HTTP requests 500 microseconds faster on average. When build the application and optimize for performance, the C program memory footprint is smaller, there was no change in data memory size, and the two implementations perform similarly. The C

implementation makes heavy use of goto statements whereas the Em implementation divides the single packet processing function into multiple functions. The inlining of these functions in several call locations causes the larger program memory footprint in the Em implementation for the performance-optimized case. Additionally, the use of local temporary variables and additional parameters to functions in the Em implementation results larger data memory size in the performance-optimized case. Despite its use of modularity, design patterns, and configurability, an Em implementation does not compromise performance and resource utilization.

Reuse of improved code

With the TCP/IP stack ported to Em and usable in one application, we now demonstrate how the entire stack was reused on a different device having a different transport mechanism. For this demonstration, we use the MSP430 wireless development kit mentioned in the above section, created an Echo [23] client/server and reused the TCP/IP protocol stack we implemented in Em. The development kit includes two identical boards each containing a 2.4GHz wireless transceiver. The data memory (RAM) capacity of these devices is 1KB. To utilize the existing stack in this application, we implement a new transport layer module and configure the packet buffer according to the availability of memory.

```
package em.distro.ez430.rf2500

from em.net.transport import BufferManager
from em.net.transport import RadioTransport
from em.net.transport import SingleBufferManager

from em.net import Stack
from em.net import Utils

from em.parts.CC2500 import CC2500 as Radio
import CC2500Config

composite NetC {

    export Stack
}

def em$preconfigure() {

    seal Stack.Transport as RadioTransport
    seal BufferManager.Provider as SingleBufferManager
    seal Radio.Configuration as CC2500Config
    seal Utils.cpuLittleEndian as Stack.cpuLittleEndian

    BufferManager.Provider.maxBufferSize = 62
}
```

Figure 8.13: Composite for configuring the network stack.

To support the wireless radio transport in this application we create a new module that implements the TransportI interface. The module, `RadioTransport`, encapsulates all the data and functionality required to send and receive data over the wireless radio. `RadioTransport` implements the interface functionality to get data in and out of the TCP/IP stack and uses our radio driver from, shown in Figure 8.1, for sending and receiving data over the radio. We implement all transport layer address resolution protocols and facilities in this module.

To utilize the new transport module, and to demonstrate its use without modification to the stack, we encapsulate the configuration of the network stack within a composite called NetC shown in Figure 8.13. The composite imports all the modules necessary to set various configuration parameters and bind proxies present in the imported modules. Specifically, the figure shows the CC2500 radio module being bound to the stack transport proxy. The previous section's evaluation, which used an Ethernet enabled device, bound this proxy to an Ethernet transport module. Additionally, the configuration parameter `maxBufferSize` of the BufferManager Provider is set according to the memory constraints of this system, namely the size of the packet buffer is set to 62 bytes in length.

No other modification to the TCP/IP stack or its protocols were necessary for this demonstration. A new transport layer module to support the hardware's transport layer and a configuration of the packet buffer size for system memory constraints were the only actions taken. The entire TCP/IP stack, a piece of software with over a thousand lines of code which depends on hardware functionality, was reused without modification. Em's support for modularity and the language's ability to decouple interface from implementation were essential to enabling this software to be reused on different hardware.

8.1.3 Reuse of existing C code

We next investigate how to use Em to reuse legacy C source code within microcontroller applications. For this demonstrations, we employ the popular Luminary Micro library of drivers. This software package includes the driver code for all peripherals integrated into their ARM Cortex-M3 devices [56]. The library is written in C and is available as a collection of source files from Luminary Micro. With Em, we can reuse the entire driver library without modification within applications using Luminary Micro's devices.

In the library, each peripheral's functionality is defined in a source file named for the peripheral. For example, the file Timer.c contains the peripheral driver for timers. In Em, we create a template to instantiate modules that each incorporates a peripheral's C source. Figure 8.14 shows the template. Upon instantiation of a module from the template, clients of the module can access any C symbol - functions, data types, register definitions.

We also create a composite to instantiate modules from the template and to export the modules under descriptive names. Figure 8.15 shows part of the composite and its exports for modules that represent peripheral drivers from the library.

To enable applications to be written independently of hardware peripheral implementations, we write a collection of interfaces that includes GpioI, UartI, TimerMilliI and others. We create modules that implement these interfaces, each of which imports

```
template StellarisWareT { }

def em$generateUnit() {

        |-> package `StellarisWareT.em$packageName`
        |->
        |-> module `StellarisWareT.em$unitName` { }
        |->
        |-> def em$generateCode( prefix ) {
        |->     |-> #include <driverlib/`em$unitName`.c>
        |-> }
}
```

Figure 8.14: Template for modules wrapping library functionality.

from the composite the particular peripheral module that it requires. Figure 8.16 shows a millisecond timer module, `TimerMilli32BitTimer0`. This module imports from StellarisWareC, the modules `Sysctl` and `Timer`. Any symbol defined in the source files of these two peripherals is accessible from within `TimerMilli32Bit...` The `start` function of `TimerMilli32Bit..` calls functions and references symbols defined in Timer.c.

This process enables any part of the Luminary Micro driver source library to be reused without modification. The Em translation integrates automatically only the source used by the application. The code becomes part of the resulting C file generated by the Em translator and as such, undergoes whole program analysis for aggressive optimization and tight coupling of application and library code. Finally, if source files for a legacy library are not available (i.e., they are only available as header and object files), we can incorporate them into an Em application by having the template

```
import StellarisWareT {} as Cpu
import StellarisWareT {} as Interrupt
import StellarisWareT {} as Sysctl
import StellarisWareT {} as Gpio
import StellarisWareT {} as Timer
...

composite StellarisWareC {

    export Cpu
    export Interrupt
    export Sysctl
    export Gpio
    export Timer
    ...
}
```

Figure 8.15: Composite to instantiate modules from template.

import the libraries header files and linking the application against the library during build-time.

8.2 Building Block Applications

In this section we evaluate the differences between programs written in nesC/-TinyOS, the Arduino platform, and Em, through a comparison of four building blocks applications – simple functionality used in the majority of applications in this domain – that are compatible across these development platforms. Our goal is to understand the differences in performance (memory footprint and cycles executed) produced by each platform and to evaluate whether Em is able to offer similar or better performance given the design decisions we employ to improve ease-of-development through intu-

```
from em.hal import TimerMilliI

from StellarisWareC import Timer
from StellarisWareC import Sysctl

module TimerMilli32BitTimer0 implements TimerMilliI {}

...

def start() {

    ^TimerDisable(^TIMER0_BASE, ^TIMER_A)
    ^TimerLoadSet(^TIMER0_BASE, ^TIMER_A, cyclesPerMs)
    ^TimerIntEnable(^TIMER0_BASE, ^TIMER_TIMA_TIMEOUT)
    ^TimerEnable(^TIMER0_BASE, ^TIMER_A)
    running = true
}
...
```

Figure 8.16: Millisecond timer using driver library functions.

itive and familiar software engineering support (module-focused, global address space, component model, composition support, etc.).

The null or empty application demonstrates the bare minimum program and reflects the lowest overhead of each system. The blink application blinks an LED periodically and uses hardware interrupts to dispatch a single event whose handler accesses the hardware synchronously. The sense application reads a sensor value from an analog to digital (A/D) converter asynchronously and displays three bits of the value read on digital I/O lines. This application uses multiple interrupt sources, exercises asynchronous interaction with hardware and manages multiple events. The sensemod application reads an A/D value asynchronously and uses the value to modulate the blinking rate of an LED. This application also uses multiple interrupt sources, manages multiple con-

current events from asynchronous processes that interact with hardware. We evaluate each application by building it for the equivalent Atmel AVR5 family of processors, specifically the ATmega128 and ATmega168 processors, using the gcc v4 toolchain and inspecting the resulting binaries using the standard binutils package. We present our results in Figure 8.17.

Program (Bold)	Program	Data		
Null	Memory (bytes)	Memory (bytes)	Files	LOC
NesC/TinyOS	616	4	2	8
Arduino	436	9	1	2
Em	346	4	1	4
Blink				
NesC/TinyOS	1898	33	2	17
Arduino	1026	13	1	10
Em	754	23	1	14
Sense				
NesC/TinyOS	2850	47	2	34
Arduino	936	21	1	23
Em	1082	35	1	34
SenseMod				
NesC/TinyOS	3128	56	3	37
Arduino	1020	14	1	19
Em	1572	44	1	28

Figure 8.17: Resource usage for NesC/TinyOS, Arduino, and Em applications.

With respect to memory usage by each sample application, our results show that the Em implementations consume 44% - 62% less program memory than those employ-

ing the other systems. Em offers the same or less (30%) data memory consumption than TinyOS for equivalent functionality. The savings in program and data memory is partially due to the use of build-time computations to determine configuration parameters that are used at runtime. The build-time calculation of these values is key because it saves program memory, data memory, and runtime resources as instructions for the computation need not be generated or executed on the device. Moreover, the resulting values are stored as constants in program memory instead of data memory and employed for constant-propagation by the compiler. The use of gcc's whole-program analysis and optimizations provide further savings by optimizing across module boundaries inlining code and eliminating function call overhead. The savings in memory enables more complex functionality to fit into the limited program space of the microcontrollers. For applications with equivalent functionality, the savings in data memory can provide power savings over other systems since RAM usage is typically a primary consumer of overall system power.

NesC/TinyOS components use fine-grained interfaces in an effort to reduce resource consumption in applications. By using narrow interfaces, components can use and provide the minimal amount of code to enable their functionality [4, 47]. Em components are specified with more coarse-grain functionality (that we believe is more intuitive – i.e. related functions are grouped together). We find in our experimentation however, that using course-grain interfaces achieves equal or better utilization of resources by

relying upon whole-program optimization to remove dead code and provide optimizations mentioned above.

Arduino does not provide any runtime support for concurrency. Thus, the lack of an event system and scheduler is reflected in the numbers that show lower resource consumption for some applications. We implement the sample applications in Arduino using only the functionality the platform provides to its user [7] so that the program semantics are as similar as possible for the intended purpose. This means that we use strictly sequential, synchronous, code for these applications (all but sensemod – which we discuss below). While the implementations may have achieved a similarity in behavior with the Em and TinyOS applications, they are not equivalent. We include them only as a reference. Without support for event handling and scheduling, users must write/rewrite their own versions leading to significant redundancy and a lack of reusability. As such, Arduino is not appropriate for our context (application development for heterogeneous devices and platforms).

Next, we attempt to evaluate the complexity of writing these applications in each system by looking at the number of source files and number of lines of code required for an implementation. Arduino applications are all implemented in a single file and have the fewest number of lines of code implying ease-of-development. However, the event handling capabilities of both TinyOS and Em are not available and implementing the

same functionality for Arduino applications is a non-trivial task requiring significantly more code.

TinyOS applications all required at least two source files since modules have a strictly local namespace and modules that provide functionality cannot be used without being configured, or wired, to at least the main application configuration. In the Em applications, since modules can be directly used from within other modules, only a single source file is necessary. In all applications evaluated, the Em versions used equal or fewer lines of code to express the same functionality as TinyOS. Reducing the number of source files and lines of code a programmer must write and maintain reduces programming complexity and enables a more direct, intuitive mapping between the abstract structure of an application and the physical form it takes in code.

8.3 Event Model Comparison

In this section, we evaluate the performance of the concurrency mechanism we implemented in Em and compare it to a similar mechanism found in nesC/TinyOS. We have provided this concurrency mechanism as non-language support for application development; we did not develop the mechanism into the language itself. NesC/TinyOS, on the other hand, has implemented its mechanism directly into the language, with keywords and constructs that are necessary to utilize the feature. The basis of the event

dispatching in our implementation for this evaluation was identical to that found in the NesC implementation.

The NesC/TinyOS concurrency model supports tasks, split-phase functionality, and uses bi-directional interfaces. Split-phase functionality means that a request for action returns immediately and completion of the request is acknowledged by a call-back. This functionality is necessary in reactive systems so that long-running operations do not block the requesting client, potentially interfering with system reactivity. Tasks are the mechanism through which split-phase functionality is implemented in NesC/TinyOS. Bi-directional interfaces refer to the pairing of functions called `commands` and `events`. A command requests functionality and is implemented by interface providers and events signal the completion of the request and are implemented by interface users. While some clients that issue commands need notification of their completion, it is often the case that many do not. Unfortunately, even in such cases, clients must still provide implementations (empty implementations in this case) for all events of used interfaces.

The left side of Figure 8.18 shows a TinyOS interface, Send, providing a bidirectional pair of functions, the command `send` and the event `sendDone`. Below the interface are the parts of the split-phase interaction. The client call to the command `send` starts the operation and returns immediately with a return value indicating if the request succeeded or failed. Upon completion of the requested action, the provider

```
interface Send {

  command error_t send(...);
  event void sendDone(error_t err);
  ...
}

// start phase
send();

//completion phase
void sendDone(error_t err) {
  if (err == SUCCESS) {
    ...
  }
}
```

```
component Sender {

  provides interface Send;
}

implementation {
  ...

  task sendTask() {
    ...
    signal sendDone;
  }

  command error_t send(...) {
    if (not_sending) {
      post sendTask();
      return SUCCESS;
    } else {
      return FAIL;
    }
  }
```

Figure 8.18: Split-phase client and implementing component in TinyOS.

of the functionality will signal requesting client that the operation completed invoking their sendDone function. Regardless of whether the client wishes to be informed the command's completion, they must provide an implementation of sendDone in order to use the send interface.

The right side of Figure 8.18 shows how the split phase functionality is implemented by the provider of the Send interface. When the `send` command is invoked, if the component can carry out the operation it will post a task, which it implements itself and defer completion of the operation until a later time so that the call can return to the client. At a later time, the TinyOS scheduler will dispatch the posted task (sendTask)

which will carry out the send operation and when it completes it will raise the signal sendDone in effect invoking the clients sendDone event function.

The Em concurrency model supports the same behavior as TinyOS only in a more general manner. Events in Em are simple structures, associated with the module they are defined within and hold a pointer to a function within the same module that will handle the event when it occurs. If a module's function may block a calling client, it must be implemented such that it takes a reference to its client's event which will be posted when the called function's action completes. Module's can post events to themselves and use this as a way to return from a clients call and defer completion of the call's action until a later time, notifying the client by posting their event. This concurrency model provides the same split-phase and event notification behavior as NesC/TinyOSs while not requiring that developers implement bi-directional interfaces.

On the left side of Figure 8.19, the module MyModule defines a private event called `sent` and a function `hasSent`. The implementation of MyModule defines the `hasSent` function and within the `my_func` function makes use of the Sender module (imported at the top). The send operation of the Sender module, defined in the right side of Figure 8.19, is potentially a long running operation. To not block its caller, Sender.send has been implemented to take a reference to a client's event which Sender will use to notify the client when the operation has completed. Thus far, MyModule interacts with module Sender using split-phase behavior equivalent to TinyOS. Notice

```
import Sender
from em.bios import EventDispatcher

module MyModule { }

private {

    sent: EventDispatcher.Event
    function hasSent(): Void
}
...

# completion phase
def hasSent() {
 ...
}

def my_func() {
    sent.init(hasSent)

    # start phase
    Sender.send(sent, ...)
    ...
}
```

```
module Sender {

    send( e: Event&, ... ): Bool
}

private {
    ...
    sendEvent: EventDispatcher.Event
    clientEvent: EventDispatcher.Event&

    function doSend: Void
}

def doSend() {
    if (clientEvent) {
        Scheduler.post(clientEvent)
    }
}

def send( e, ... ) {
    if (buffer_full) {
        return false
    } else {
        clientEvent = e
        Scheduler.post(sendEvent)
        return true
    }
}
```

Figure 8.19: Split-phase behavior implemented in Em.

The right side of Figure 8.19 shows how the Sender module implements the split-phase behavior. The Sender module defines an event `sendData` and holds a reference to another event `clientEvent`. It defines its public function `send` and a private function `doSend`. When a client invokes send, if Sender is capable of carrying out the send operation, it stores a reference to an event, passed in by the client, that will be posted when the operation completes. Sender then posts its own event `sendData` to the scheduler, deferring the work of the send operation, and allowing `send` to return to the client. When the `sendData` event is dispatched the `doSend` function will be invoked to carry out the sending operation and ultimately, if the `clientEvent`

reference is valid, post the `clientEvent` event. If the `clientEvent` event was valid and dispatched, its handler, hasSent will be invoked, signaling completion of the split-phase operation.

Our evaluation shows that the event model we have implemented in Em can achieve equivalent split-phase behavior as TinyOS. Furthermore, in our implementation, we do not require bi-directional interfaces and we leave the decision of being informed of split-phase operation-completion up to the client. By relaxing the requirement to have bi-directional interfaces and always implement functionality in a split-phase manner the programmer has to write less code and when bi-directional functionality is required, it is implementable.

8.4 Learning Em & Developing Applications

To investigate the ease of learning and developing non-trivial applications in Em, we taught a ten-week course on modular embedded systems programming to computer science and computer engineering undergraduate students. The course comprised 18 students in a technical discipline, however, their knowledge of programming and technical skills in general varied as there were both freshmen and seniors. We asked students enrolling in the course to have rudimentary experience with two programming languages. The majority of students had been exposed to programming in Java and

C++ and had no prior experience programming a microcontroller. A few students had only been exposed to C in an introductory course.

The first four weeks of the course introduced students to embedded systems concepts such as device drivers, interrupt handling and concurrency through lectures and demonstration of implementations in Em. Exercises were given that tested both the student's understanding of the concepts they were working with as well as their implementation in Em. An emphasis was placed on thinking about embedded software in terms of concepts prevalent in high-level languages such as encapsulation, abstraction, software reuse and portability. In the remaining six weeks of the course, students implemented a project of their choosing using one of three off-the-shelf development boards, namely the Arduino Duemilanove [5], the Texas Instruments EZ430-RF2500 kit [33], or the LuminaryMicro LM3S811 evaluation board [55].

In the following subsections, we present the qualitative results of surveys given to students on the first and last days of the course and describe some non-trivial projects created. With the initial survey we sought to gauge the students' level of experience with programming in general and programming microcontrollers in specific. The final survey was given to understand the students experience learning and using Em. With respect to the student projects, we present several successful implementations notable for either their design, complexity or the student's level of skill. For each project we also present the application's footprint and number of lines of code.

8.4.1 Student Survey Results

In the course we taught we gave our students a survey on the first day of class to gauge their level of technical knowledge in general, and knowledge of concepts related to embedded systems in specific. Our course was an elective for computer science and computer engineering students and thus attracted mostly junior and senior students. The course was comprised of eighteen students; one freshman, three sophomores, five juniors, eight seniors and one graduate student.

The programming languages most familiar to students were Java and C++, with general exposure to C. Other languages students frequently mentioned having exposure to were Python, Ruby and C#. Despite the majority exposure to object oriented languages, 56% of students failed to accurately describe the main concepts of object oriented languages such as encapsulation, information hiding, inheritance and polymorphism. The majority of students understood the concept of software modularity. More than 72% of the students had never programmed for or had been exposed to microcontrollers. With respect to systems level understanding, 67% of the students had not been exposed to operating systems concepts and while a majority of students described the concept of concurrency accurately, a large majority of students were not familiar with how it was implemented. Similarly, 67% of students understood what a device driver was while 94% of students had never been exposed to the implementation of one. Finally, it was rare that a student had been concerned with software reuse and

portability in the past and many noted their biggest struggle with C and C++ was understanding and working with pointers and memory management. Our first survey showed that the students in our course were members of the audience our domain targets.

Upon completion of the course we conducted a second survey to gain an understanding of the student's experience learning and develop applications in Em. When asked what was most challenging in working with Em the most frequent responses noted that it was learning the syntax and understanding the modularity of the language. Regarding the modularity, Em modules are singletons and are not instantiated like objects in an object oriented language. This notion was not familiar to many students. Regarding learning of syntax, the majority of students did not have proficiency in multiple languages and this could be related to the process of learning a new language. Other responses by multiple students on the challenges of working with Em related to understanding parameter passing semantics and several reported that it was a lack of reference documentation and example code. In our current implementation of Em, function signatures that accept aggregate types such as opaque types and structs must explicitly be denoted as being passed by reference. Many students overlooked this requirement and encountered difficulties.

When students were asked to list the concepts that were new to them when learning Em, the most frequent responses, in order of frequency, were: working with an event driven programming model, utilizing proxies, and working with a build-time execution

context. Other than the event programming model, the responses did not note that these concepts were overwhelmingly challenging to understand or difficult to work with. With respect to student's perceived difficulty utilizing an event driven model for concurrency, on a scale of one to five, 67% of students rated it as a three implying it was moderately difficult for them. 22% rated the perceived difficulty level at a two. Overall 50% reported having been exposed to event driven programming prior to the course.

The remaining survey questions pertained to the student's perception of Em as a high-level language and given their experience with Em, how they might improve the language. 63% of students reported they perceived Em as a high-level language. When asked what they would have liked to spend more time on in class, the majority of responses mentioned further explanation of the use of interfaces, proxies and the build-time execution context. The emphasis on these language features, as opposed to low level hardware topics, we assume implies the students were more attuned to the high-level aspects of the language. Regarding improvements to the language, the responses were quite varied. The most common responses had to do with aspects related to the maturity of the language itself, such as more helpful errors from the translator, the availability of documentation, code examples, and more diverse libraries.

Overall, the students in the course were successful in learning Em and were capable of developing non-trivial applications in the language. The perception of Em as a high-level language, we feel, helped students employ their past programming experi-

ence without too much concern for low-level programming in a resource constrained environment. The challenges many students encountered could have been related to the task of learning a new language, while the repeated difficulty with topics such as parameter passing by value verses by reference have lent themselves to informing us how to reduce such difficulties in the future. The following subsection details four applications created by students in the six weeks they had to execute an idea of their own choosing.

8.4.2 ViaCar

ViaCar [86] is an undergraduate design competition where teams of students design, build and race an autonomous car which must follow a track marked by 1-inch tape on a dark carpet. As a course project, a second year computer engineering student, having only a rudimentary understanding of C and minimal exposure to object oriented programming, developed the software for his teams entry to the competition in the course. The developed vehicle with the microcontroller development board mounted on top is pictured in Figure 8.20.

The student's design used an ARM-Cortex development board, and an array of twelve infrared (IR) LED and phototransistor pairs arranged perpendicularly to the track being followed. The application used periodic timer interrupts to actuate the IR LEDs and read the phototransistors to determine the location of the tape across the

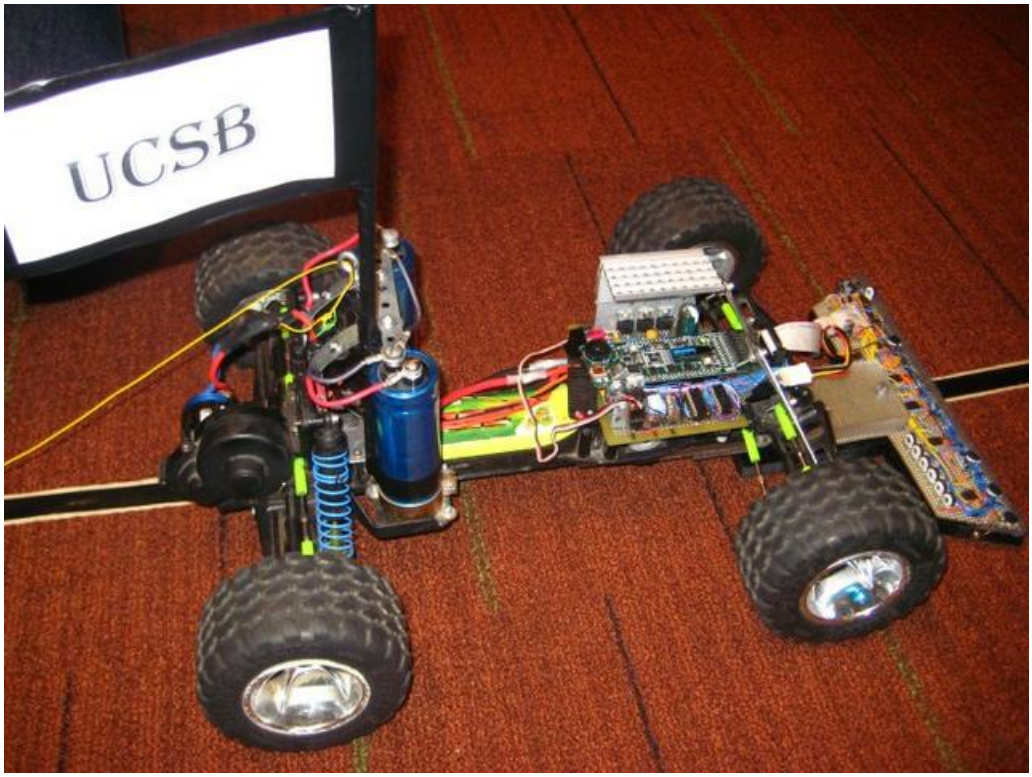

Figure 8.20: Student created autonomous line following car.

array. Given the location, an error factor would be computed and passed to the PID controller to determine a correction factor which would in turn be used to adjust the PWM duty cycle of the servo steering motors.

Modules developed by the student included the PID controller, motor control logic using a PWM generator integrated in the microcontroller, IR LED actuation and phototransistor input. The development board came with a library of drivers for the microcontroller peripherals written in C, which was entirely reusable by the student. The car finished in 4th place out of 17 contestants in the competition. The application consisted

of 1072 lines of code and had a footprint of program and data memory of 12KBytes and 1080 bytes, respectively.

8.4.3 Persistence of Vision Display

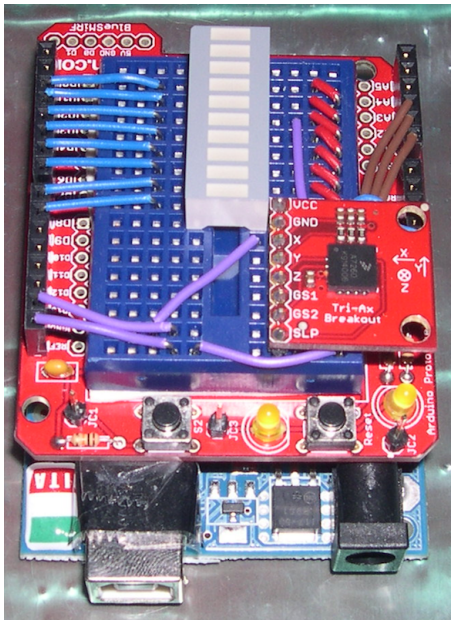

Figure 8.21: Student developed persistence of vision display.

Persistence of vision (POV) is the phenomenon of the eye by which an after-image is though to persist for approximately 1/25th of a second on the retina. A third year computer science student with no previous microcontroller programming experience developed an LED-based POV display using the Arduino Duemilanove development board, an accelerometer and a strip of 8 LEDs. Critical to this application is the continuous sensing of motion along with timely coordination of the blinking of LEDs such

that as the device is moved the appropriate LEDs are illuminated to give the POV effect - the appearance of textual messages in the air. Using a timer module, the application samples the accelerometer data at a frequency of 100Hz. When motion is detected beyond a particular threshold, an event is posted to a module responsible for blinking out the sequence of LEDs to display the message. The message blinker module uses another virtual timer to update the strip of LEDs at 1ms intervals.

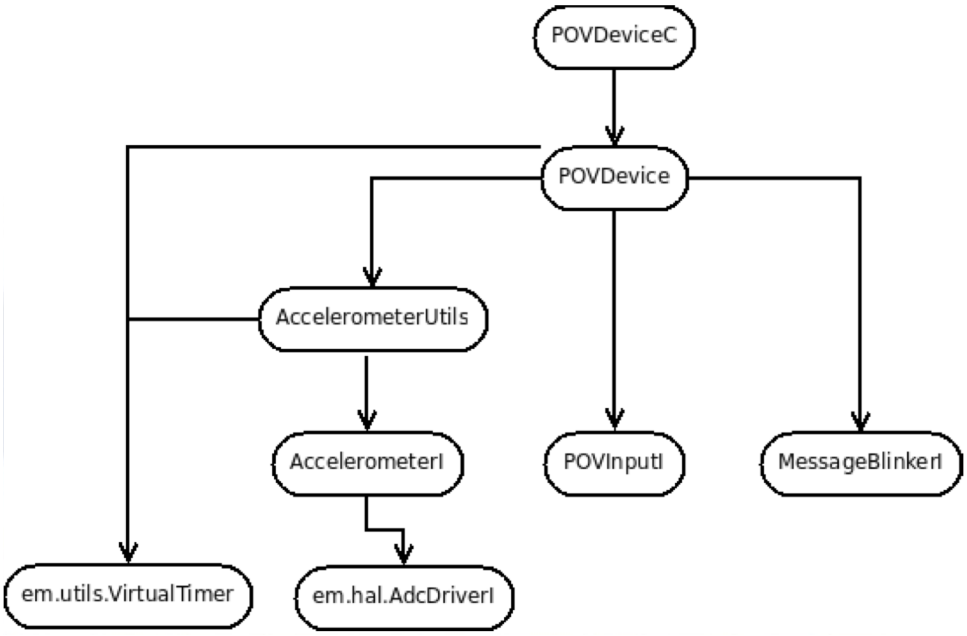

Figure 8.22: Student's application design of persistence of vision display.

Particularly noteworthy of this student's application design was their use of Em's abstraction, reuse and portability mechanisms, specifically interfaces, proxies and composites. A diagram of the high-level design of the application is depicted in Figure 8.22. Interfaces, such as `AccelerometerI`, `MessageBlinkerI`, and

`POVInputI` were created to define the abstract functionality of the accelerometer, the message blinker and the input source for messages to display respectively. Since each of these components can vary, yet are independent of the core functionality of the application, the student used proxies to enable ease of variation. To allow the hardware to be configured easily and enable the main application to be entirely portable across hardware, the student created the `POVDeviceC` composite. The application software contained approximately 930 lines of code and had a program and data memory footprint of 5154 bytes and 670 bytes respectively.

8.4.4 Ocarina Instructor

A student interested in learning to play the ocarina, a small flute-like wind instrument, developed an application to instruct himself how to play the instrument. The student was in his senior year of studies and had never developed and embedded systems application. The application, consisting of about 659 lines of code, was developed using the Arduino Duemilanove development board, an 8x8 LED matrix display and a push button for user input.

Each LED on a row of the display corresponded to a hole on the flute. The Illuminated LEDs on a row would instruct the user to cover those holes in order to play a particular note. To teach the user a song, the application would display notes in sequence, illuminated for the note's duration. If the note was to be held for particular

duration, it would ”fall” from the top of the display to the bottom in the appropriate amount of time. The device's button allowed a user to interact with the application to restart a particular song from the beginning or to skip ahead to another song stored in the microcontroller's memory.

The existence of several modules to control hardware, specifically the SPI port and simple button input, helped the student focus on developing application functionality. The LED display used in the application was controlled via the SPI port. Having a module for driving the SPI port allowed the student to control the display and start implementing his application functionality within the first hour of obtaining the display. Along with a module for receiving input from a button connected to the microcontroller, the student was able to start physically interacting with his project on the first day of development. The remaining weeks of the course, the student focused entirely on the high-level software functionality of his application.

8.4.5 The Game of Simon

Another project created by a student team with no prior microcontroller programming experience was the popular memory game Simon. Using the Arduino Duemilanove development board, four buttons, a collection of LEDs and a buzzer, the students implemented the game which flashes out progressively more complicated sequence of

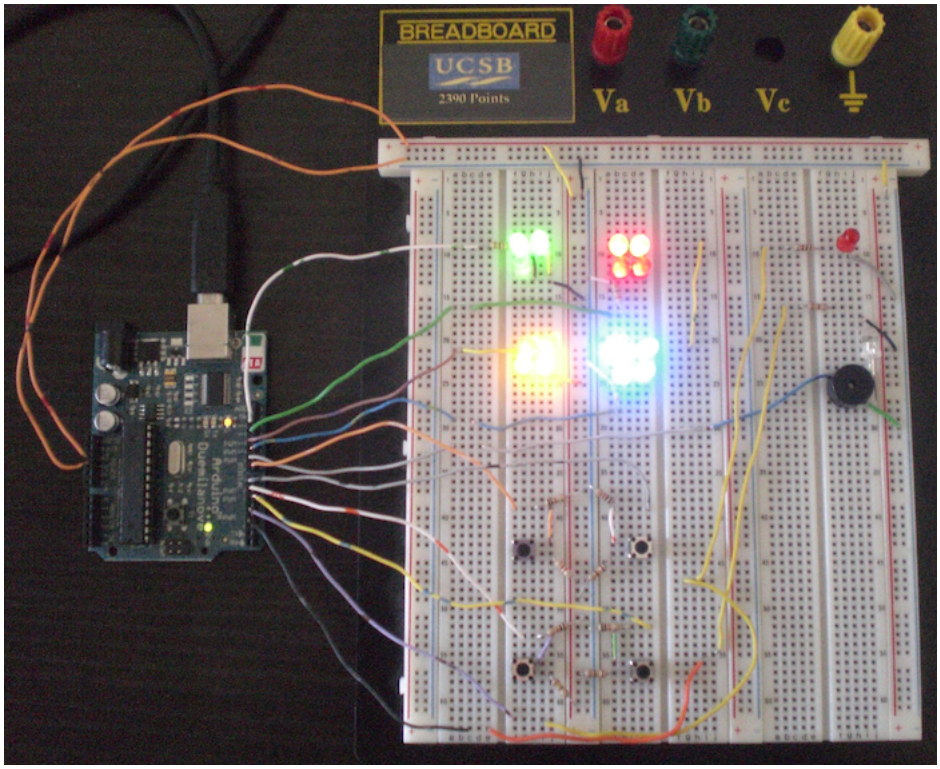

Figure 8.23: The game of simon developed by students.

light patterns for the game player to emulate. The implementation is shown in Figure 8.23.

Having interest and familiarity with software design patterns, the students modeled their application design around the Model-View-Controller (MVC) [25] pattern. Three modules (model, view and controller) were created containing approximately 423 lines of code with a program and data memory footprint of 5236 bytes and 765 bytes respectively.

The controller module polled the game buttons for any input. On detecting a button press an event would be posted to the model module which would determine what action needed to be taken. Program state would be updated and as a response events would be generated and posted to the view module which would display the color patterns on the LEDs. Each module in the MVC pattern was decoupled from the other, communicating strictly though events. This decoupling made the design easily modifiable for different display or input options without effecting or requiring the main game model to be modified.

In this application, the students were able to apply software engineering methodology used with high-level programming languages, namely design patterns. The students did not have trouble learning Em and were able to apply their previous knowledge of high-level application design, and implementation directly in this domain.

8.5 Summary

In this chapter we presented a demonstration and evaluation of the Em programming language. We first demonstrated the ability to write reusable and portable software in Em. Software that supports hardware functionality can be written such that it is reusable across hardware platforms with variation in physical and application configurations. This was articulated through an example of the design and implementation

of a device driver for a wireless transceiver component that was not integrated into a microcontroller. We then demonstrated the ability to reuse non-hardware supporting software in Em through the porting of a popular RFC-compliant TCP/IP protocol stack implemented in C. The modular structure of the protocol stack implementation in Em affords an interchangeable transport layer and protocol buffer management policy without requiring modifications to the stack itself. We implemented a web server using the stack and evaluated its implementation in terms of runtime performance and application footprint relative to an implementation in C. Our results show that the modular implementation in Em did not decrease the performance relative to C. The application footprint was only slightly larger due to the presence of more function parameters as a result of separating concerns given the lack of such separation in the original C implementation. It was also demonstrated how the entire TCP/IP stack was reused unmodified on a different hardware platform with tighter memory constraints using a different transport layer. To conclude the reuse and portability section we articulated how existing C code can be reused and integrated into an Em application.

We then compared on the resource consumption of simple building block applications whose functionality is found in most device applications. The comparison was conducted against identical applications written in Em, nesC/TinyOS, and the Arduino platform. Our results show that Em implementations consume 44% - 62% less program memory than those employing the other systems and offers the same or less (30%) data

memory consumption than TinyOS for equivalent functionality. The Arduino platform lacked the ability to interact with hardware asynchronously as well as a mechanism for concurrency. The comparison was conducted with Arduino because it is currently the most popular development platform used by non-technical experts in our domain. Additionally, we compared the concurrency mechanism implemented in Em with the one present in NesC/TinyOS. The mechanisms achieve the same functional purpose, however our implementation was not integrated into the language, where NesC's was. It was demonstrated that the split-phase functionality found in NesC/TinyOS is achievable with Em, however we have relaxed the requirement for bi-directional interfaces which require the programmer to always implement both sides of the split-phase operation, even when one is less frequently utilized.

As a demonstration of the ability for non-technical experts to learn Em and implement non-trivial applications, we presented four projects created in a 10 week course. The students in the course ranged from freshman to senior. The majority of students had never been exposed to microcontroller programming and a number of students had only minor exposure to programming in general. Prior knowledge of high-level programming language concepts and development technique familiar to student were usable in Em. Survey results from the student's experience using the language were presented.

Chapter 9

Conclusion

In this dissertation we investigate language and runtime support for software development in resource constrained microcontroller-based device applications. Traditionally, only highly skilled engineers and developers with significant financial resources had access to hardware and tools necessary for developing applications. Today, the low cost and availability of development kits, application specific hardware components, and development tools has significantly broadened the audience interested in and attempting to create applications.

Due to constraints on the availability of program and data memory, CPU processing speed and power consumption, application development has proceeded differently than in resource rich environments such as our desktop and server computers. High-level programming languages with dynamic features, general purpose operating systems for hardware abstraction and resource management, and uniform development environments that support development of reusable, portable systems all lack from resource

constrained environments. A tight coupling between hardware functionality and application logic exists, low-level languages such as assembly and C are predominantly used, and general purpose operating systems typically require more resources that are available on these devices.

The goal of our work is to bring high-level language constructs and development practices that exist in resource rich environments down to resource constrained environments without compromising efficient use of resources. Moreover, our goal is to support a broad audience - spanning from novice programmers through to expert embedded systems developers - that is able to access and utilize such language support. In addition to language support, our work investigates non-language features essential for effective, productive development by such a broad audience.

We investigate the design, implementation and evaluation of a high-level programming language for developing resource constrained microcontroller software. We select features from existing high-level languages that have been proven to support abstraction, modularity and the separation of concerns, a decoupling of interface from implementation, configurability and variability of software for differing hardware configurations and availability of runtime resources. We adopt a sparse syntactic style common in scripting languages popular among non-programmers and relative novices in order to appeal to their taste and interest. We additionally design, implement and evaluate non-language support for resource conscious models of concurrency, hardware abstraction,

and development environment features to support uniform development across hardware platforms.

The design and implementation of our domain-specific programming language is described in chapters 4 and 5 and the design and implementation of non-language support is described is described in chapters 6 and 7. Our empirical evaluation and demonstration of the efficacy of our work is presented in chapter 8. We seek to enable the existence of an ecosystem of software such that it is easily distributable and reusable across a wide range of heterogenous hardware platforms and application domains while being usable by non-technical experts and uncompromising for the most experienced developers. As such, our work supports:

- **High-level language constructs.** We incorporate successful features from existing, high-level languages that have proven to enable the expression of complex ideas in software systems. We incorporate modules as the fundamental unit of code in our language. Encapsulation and data-hiding, in addition to enabling the separation of concerns, lend themselves to enabling software abstraction. We employ interfaces to further extend abstraction capabilities. We find that a popular design pattern called the Proxy pattern, employed in the language itself, provides a lightweight and robust mechanism to decouple interface from implementation. The use of interfaces and proxies together enable the creation of software that can adapt to differing physical hardware configuration and application require-

ments without modification to code. To further extend configuration capabilities, and provide a degree of dynamism that is too expensive for runtime we introduce a build-time execution context that can leverage the rich resources of the build host. Holistically integrated into the language with identical syntax and semantics, the build-time context enables introspection, dynamic memory allocation, and offloading runtime computation from the target microcontroller onto the build host.

- **Efficiency.** Given microcontrollers with as little as several hundred bytes of memory and a few kilobytes of program memory that potentially utilize severely limited power sources, efficiency in the use of scarce resources is of utmost importance. Languages and tools in use today achieve a desirable level of efficiency and must not be compromised with the addition of high-level language constructs. Our design and implementation leverages existing compilers with their refined optimization capabilities by translating our high-level language into portable C code. This code in turn is compiled and optimized for a specific microcontroller target. We find that despite the high-level constructs in our language, we can achieve at least equivalent performance to the languages used in this domain with respect to resource utilization.

- **Modern Software Development Practices.** Modern software development practices depend on the availability of large amounts of readily available, reusable, portable software content. To enable systematic reuse of existing code we limit language constructs that may violate modularity such as implementation inheritance. We introduce configuration parameters and late binding of proxies to enable content to be written such that it can be flexibly reused in differing contexts. We introduce a construct called a composite which enables a pattern for portability and high-level configurability. We incorporate templates which save copy-and-paste reuse of code in modules with closely related functionality. We incorporate lightweight packaging semantics to easily distribute collections of related functionality. The collection of constructs and features we employ enable modern development practices to be utilized in a domain that largely lacks such practices.

- **Hardware diversity.** The diversity of available microcontrollers and hardware components for applications is extensive. Existing tools for development are fragmented and do not support a wide range of hardware. To support a diversity of hardware we leverage our language's abstraction capabilities to implement a hardware abstraction layer. Definitions for common functionality of microcontroller peripherals and other components enable higher level software to utilize hardware resources strictly based on defined functionality, not implementation.

In conjunction with the use of proxies, the decoupling of interface from implementation is achieved and enables software supporting hardware functionality to be written independent of low-level device specific implementation details. The result, as articulated in our evaluation, is an ability to support a wide range of microcontrollers and application hardware.

- **Concurrency.** The reactive nature of resource constrained systems demand timely, simultaneous, response to external stimuli. The resource consumption of general purpose operating systems preclude their use in the resource constrained domain. Developers, therefore, are left to implement mechanism for concurrency individually. We demonstrate the implementation of a reactive model of concurrency in our language which relieves the programmer of implementing such complex functionality. We compare the reactive model to the more common threading model and articulate why we have chosen one model over the other. We additionally compare our implementation to a similar model in an extant system. We find that our implementation relaxes some requirements imposed by the other system without losing the ability to express all its intended functionality.

Our empirical evaluation and demonstration of applications shows that our language and non-language (runtime) support, with its claimed benefits, enable development of sophisticated functionality without compromising scarce resources. Additionally,

through surveys and evaluation of an undergraduate course taught using our language, we find that non-embedded systems experts can successfully learn the language and develop non-trivial applications on resource constrained microcontroller devices.

9.1 Contributions and Impact

In this section we summarize our main contributions and discuss their impact. Our primary contribution is the design and implementation of a high-level programming language for our specified domain. Specifically, a language that supports modern programming constructs and practices for the development of resource constrained microcontroller-based device applications by a broad audience with diverse technical skills. The contributions made in this dissertation are as follows:

- A new high-level, general purpose, embedded systems programming language called Em that incorporates successful object oriented language and engineering techniques for the development of highly resource constrained microcontroller applications

- Language support, enforced by translation to C, for modularity, encapsulation, data hiding, and code reuse using techniques from modern high-level object-oriented languages including C++ and Java.

- Unified programming support for single file, similar syntax, application and configuration coding.
- Support for separation of concerns in code and code reuse via interfaces, independent implementation, proxies for substitution, opaque types, interface inheritance, and automatic source generation (templates).
- A reduction in the verbosity of the C language specifically to make it accessible to new/non-programmers (similar to that done for modern scripting languages).
- Demonstration of code reuse, portability, interchangeability, integration of legacy source, using real applications and sophisticated embedded systems components.
- An extensive empirical evaluation of the footprint and resource consumption for a number of program building blocks and real applications. In addition, we provide an evaluation of ability for non-expert embedded systems programmers to learn and develop non-trivial applications through the use of Em in an undergraduate college course.

The general impact of our work is an advancement in the state-of-the-art of programming languages and development practices in the specific domain of resource constrained microcontroller-based application development. Languages in this domain are predominantly low-level with C being the most prevalently used language. While suitable for development of efficient systems level software, the development practices of

expert developers in our domain continues to limit the scope of who can develop applications and the rate at which applications can take advantage of new hardware offerings. Our work puts in place the ability to create modular, configurable, reusable and portable software that functions across application domains and hardware configurations.

The fact that our work has introduced high-level language constructs into resource constrained systems without compromising resources shows that existing practices can evolve to the level we enjoy in application development for resource rich environments. Specifically, we can create device drivers for hardware components that are reusable on differing hardware platforms, in various configurations and adaptable to available resources. Developers can write software independent of hardware that can be easily distributed over the Internet and quickly integrable into diverse applications. Programmers can take more advantage of the capabilities of devices, which must remain cheap and efficient, since complexity from hard-to-understand practices have been reduced. High-level abstractions, design patterns, and other efficiencies of modern development practices are feasible.

A significant consequence of our work is the potential to enable a future generation of innovative applications to be developed. Artists, designers, students, hobbyists and people who are specialized in non-technical disciplines are currently limited in what they can accomplish using todays tools and technologies in this domain. Ideas for how microcontrollers can be applied by non-technical experts are growing with new attempts

being highlighted in newspapers, magazines, blogs and expositions internationally. It is clear that as a tool being applied in diverse application domains, microcontrollers are powerful and effective. It is inadequate to demand that these people, experts in their own domain, must master the tools and intricacies of multiple highly-technical disciplines in order to realize their ideas. Our work can continue to lower the barrier to entry for this growing audience of people and furthermore, enable them to get further in realizing their ideas than they are able to today.

Our work has shown that Em provides the necessary support to create and sustain an ecosystem of software for resource constrained environments, as exists for resource rich computing environments. Such an ecosystem expedites the application development process which is important for both reducing time-to-market of professional products, as well as the prototyping of new ideas. It is counterintuitive that the most prevalent computing systems on earth - the resource constrained systems - lag behind the technologies available for resource rich environments. Our work has made advances to bridge the gap in novel and interesting ways.

9.2 Future Research Directions

In this section we identify several directions for future research. Our work in this dissertation motivates further refinement of the language itself, the exploration of ef-

ficient and intuitive models for concurrency, environments for graphical development of applications, stream-oriented software frameworks, and new teaching methods that enable less-experienced programmers and novices of various ages to develop microcontroller applications.

With respect to language features, we believe Em can be refined to eliminate the use of pointers by programmers. Pointers are a major stumbling block for novices learning how to program and a major source of software errors. Eliminating pointers from our language, we feel, can make it more accessible to novices and non-programmers without losing efficiency or expressivity required by experts.

While a reactive concurrency model was chosen in our design and implementation, it is unclear how successful this model will be or how easy it is to learn for developers with varying technical skills. The most familiar model for existing programmers is a threaded model. Event driven programming may be familiar to some although further exploration into the usability of our chosen model is a possible future direction.

Reactive programming, as common in our domain, is data driven. Data sources and sinks represent stimuli to and reaction of the system and data that enters it is processed as it flows through various computational steps. A framework for stream-oriented application development is an interesting direction for future research. The stream-oriented approach has been successfully utilized in an embedded systems context in the past however its application remains within a highly specialized area of

digital signal processing. Expansion of the stream-oriented method to a broader domain which is accessible to a wider audience of developers could potentially provide an expressive methodology of development.

The modular constructs in our programming language, the use of proxies capable of binding differing implementations, and composites as used for application configuration lend themselves well to a graphical representation. In Chapter 3 we discuss graphical programming environments for both experts working in an embedded systems domain and non-experts working in an entirely different application domain. The use of such environments has been successful in enabling more people, specialized in their particular domain yet lacking programming skills, to be productive and expressive. Since the language constructs present in our language lend themselves well to a graphical representation, it is a potentially fruitful avenue to pursue creation of a development environment that affords a graphical process of application development.

Lastly, the understanding of technology and systems in general is of utmost importance to society. Future generations should be taught, at a high-level, the inner workings of the technical infrastructure they depend on and utilize daily. Putting programmable microcontroller systems that can react and interact with their environment in the hands of high-school students around the world could profoundly impact children's understanding of the technical world around them. Continuing to explore how to make microcontroller programming more accessible, expressive and robust for young

and non-technical audiences is a future research direction with profound implications for society.

In summary, our contributions open up various avenues with promising opportunities for continued research in the domain of resource constrained microcontroller-based application development. Advancements in this domain not only impact the ability to realize novel and potentially innovative applications, it can impact society's ability to understand and work with technology.

Appendix A

Grammar

feature-declaration:
constant-declaration
config-declaration
variable-declaration
function-declaration
type-declaration
struct-type-declaration
opaque-type-declaration
opaque-type-representation
proxy-declaration

feature-declaration-list:
feature-declaration feature-declaration-list$_{opt}$

feature-name:
identifier

constant-declaration:
host$_{opt}$ **const** *feature-name* **:** *type-specification initializer* **;**$_{opt}$

config-declaration:
host$_{opt}$ **config** *feature-name* **:** *type-specification initializer*$_{opt}$ **;**$_{opt}$

variable-declaration:
host$_{opt}$ **var** *feature-name* **:** *type-specification initializer*$_{opt}$ **;**$_{opt}$

function-declaration:
host$_{opt}$ **function** *function-name* **(***argument-declaration-list*$_{opt}$**):***type-specification* **;**$_{opt}$
template *feature-name* **(** *argument-declaration-list*$_{opt}$ **)**

function-name:
feature-name

argument-declaration:
argument-name **:** *type-specification initializer*$_{opt}$

argument-declaration-list:
argument-declaration
argument-declaration **,** *argument-declaration-list*

argument-name:
identifier

argument-name-list:
argument-name
argument-name **,** *argument-name-list*

type-declaration:
host$_{opt}$ **type** *type-name* **:** *type-specification* **;**$_{opt}$

enum-type-declaration:
host$_{opt}$ **type** *type-name* **: enum {** *enum-value-list* **,**$_{opt}$ **} ;**$_{opt}$

struct-type-declaration:
host$_{opt}$ **type** *type-name* **: struct {** *field-declaration-list* **} ;**$_{opt}$

opaque-type-declaration:
host$_{opt}$ **type** *type-name* **: opaque {** *function-declaration-list* **} ;**$_{opt}$

opaque-type-representation:

def type *type-name* **: struct** **{** *field-declaration-list* **}** **;**$_{opt}$

type-name:
feature-name

enum-value-list:
enum-value
enum-value **,** *enum-value-list*

enum-value:
identifier

field-declaration:
field-name **:** *type-specification* **;**$_{opt}$

field-name:
identifier

field-declaration-list:
field-declaration field-declaration-list$_{opt}$

field-declaration:
field-name **:** *type-specification* **;**$_{opt}$

field-name:
identifier

function-declaration-list:
function-declaration function-declaration-list$_{opt}$

proxy-declaration:
proxy *feature-name* **implements** *interface-name* **;**$_{opt}$

type-specification:
base-type type-operator-list$_{opt}$ **&**$_{opt}$

base-type:
intrinsic-type
declared-type

declared-type:
type-name
unit-name **.** *type-name*

type-operator-list:
***** *type-operator-list*$_{opt}$
[*expression*$_{opt}$ **]** *type-operator-list*$_{opt}$
[length] *type-operator-list*$_{opt}$
(*argument-type-list*$_{opt}$ **)** *type-specification type-operator-list*$_{opt}$

argument-type-list:
type-specification
type-specification **,** *argument-type-list*

intrinsic-type:
Bool | Char | IArg | Int8 | Int16 | Int32 | Ptr |
Ref | String | UArg | UInt8 | UInt16 | UInt32 | Void

initializer:
= *expression*

expression:

basic-expression
binary-expression
call-expression
cast-expression
conditional-expression
escape-expression
hash-expression
index-expression
selector-expression
size-expression
vector-expression

expression-list:

expression
expression `,` *expression-list*

basic-expression

intrinsic-name
identifier
literal
`(` *expression* `)`

binary-expression

expression binary-operator expression
expression assignment-operator expression

binary-operator:

one of: `+ - * /% ==!= < <= > >= & | ^ && ||`

assignment-operator:

one of: `= =+ =- =* =/ =% =<< =>> =& =| =^`

call-expression:

expression `(` *expresssion-list*$_{opt}$ `)`

cast-expression:

`<` *type-specification* `>` *expression*

conditional-expression:

expression `?` *expression* `:` *expression*

escape-expression:

`^` *identifier*
`^^` arbitrary text `^^`

hash-expression:

`{ }`
`{` *hash-initializer-list* `,`$_{opt}$ `}`

hash-initializer-list:

hash-initializer
hash-initializer `,` *hash-initializer-list*

hash-initializer:

feature-name `:` *expression*
field-name `:` *expression*

index-expression:

expression `[` *expression* `]`

selector-expression:

expression **.** *feature-name*
expression **.** *field-name*
expression **. length**

size-expression:

sizeof < *type-specification* >
alignof < *type-specification* >
offsetof < *type-specification* **.** *field-name* >

unary-expression:

prefix-operator expression
postfix-operator expression

prefix-operator:

one of: **+ - * ! & ++ --**

postfix-operator:

one of: **++ --**

vector-expression:

[ ]
[*expresssion-list* **,**$_{opt}$ **]**

statement:

assert-statement
basic-statement
break-or-continue-statement
do-statement
for-statement
if-statement
local-declaration-statement
print-statement
program-termination-statement
return-statement
seal-statement
switch-statement
text-generation-statement
while-statement

statement-list:

statement statement-list$_{opt}$

assert-statement:

assert *expression* **;**$_{opt}$

basic-statement:

expression **;**$_{opt}$

break-or-continue-statement:

break ;$_{opt}$
continue ;$_{opt}$

do-statement:

do { *statement-list*$_{opt}$ } **while** (*expression*) **;**$_{opt}$

for-statement:

for (*expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$) { *statement-list*$_{opt}$ }
for (*local-declaration-statement* **;** *expression*$_{opt}$ **;** *expression*$_{opt}$) { *statement-list*$_{opt}$ }

if-statement:
if (*expression*) { *statement-list*$_{opt}$ } *elif-clause-list*$_{opt}$ *else-clause*$_{opt}$

elif-clause-list:
elif (*expression*) { *statement-list*$_{opt}$ } *elif-clause-list*$_{opt}$

else-clause:
else { *statement-list*$_{opt}$ }

local-declaration-statement:
var *local-name* **:** *type-specification initializer*$_{opt}$ **;**$_{opt}$

local-name:
identifier

print-statement:
printf *print-format* **;**$_{opt}$
printf *print-format* **,** *print-value-list* **;**$_{opt}$

print-format:
string-literal

print-value-list:
expression **,** *print-value-list*

program-termination-statement:
fail ;$_{opt}$
halt ;$_{opt}$

return-statement:
return *expression*$_{opt}$ **;**$_{opt}$

seal-statement:
seal *expression* **as** *expression* **;**$_{opt}$

switch-statement:
switch (*expression*) { *case-clause-list*$_{opt}$ *default-clause-list*$_{opt}$ }

case-clause-list:
case *expression* **:** *statement-list*$_{opt}$ *case-clause-list*$_{opt}$

default-clause:
default : *statement-list*$_{opt}$

while-statement:
while (*expression*) { *statement-list*$_{opt}$ }

translation-unit:
module-unit
interface-unit
composite-unit
template-unit

unit-name:
identifier

module-unit:
package-designation import-directive-list$_{opt}$ *module-header public-features private-implementation*

module-header:
host$_{opt}$ **module** *module-name module-implements*$_{opt}$

module-name:
unit-name

module-implements:
implements *interface-name*
implements *interface-name* **using** *module-name*

interface-unit:
package-designation import-directive-list$_{opt}$ *interface-header public-features*

interface-header:
host$_{opt}$ **interface** *interface-name interface-extends*$_{opt}$

interface-name:
unit-name

interface-extends:
extends *interface-name*

composite-unit:
package-designation import-directive-list$_{opt}$ *composite-header public-features private-implementation*

composite-header:
composite *composite-name composite-extends*$_{opt}$

composite-name:
unit-name

composite-extends:
extends *composite-name*

template-unit:
package-designation import-directive-list$_{opt}$ *template-header public-features private-implementation*

template-header:
template *template-name*

template-name:
unit-name

package-designation:
package *package-name* **;**$_{opt}$

package-name:
identifier
package-name **.** *identifier*

import-directive:
import-from$_{opt}$ **import** *module-name import-as*$_{opt}$ **;**$_{opt}$
import-from$_{opt}$ **import** *interface-name import-as*$_{opt}$ **;**$_{opt}$
import-from$_{opt}$ **import** *composite-name import-as*$_{opt}$ **;**$_{opt}$
import-from$_{opt}$ **import** *template-name import-template-parameters import-as* **;**$_{opt}$

import-directive-list:
import-directive import-directive-list$_{opt}$

import-from:

from *package-name*
from *composite-name*

import-template-parameters:
hash-expression

import-as:
as *unit-name*

public-features:
{ *export-directive-list*$_{opt}$ *feature-declaration-list*$_{opt}$ }

export-directive-list:
export-directive export-directive-list$_{opt}$

export-directive:
export *module-name* ;$_{opt}$
export *module-name* **implements** *interface-name* ;$_{opt}$

private-implementation:
private-features$_{opt}$ *function-definition-list*$_{opt}$

private-features:
private { *feature-declaration-list*$_{opt}$ }

function-definition-list:
function-definition function-definition-list$_{opt}$

function-definition:
def *declared-function-name* (*argument-name-list*$_{opt}$) { *statement-list*$_{opt}$ }

declared-function-name:
intrinsic-name
function-name
type-name . *function-name*

lexical-element:
intrinsic-name
identifier
literal
comment

intrinsic-name:
identifier prefixed with em$

identifier:
identifier-first-letter
identifier identifier-letter

identifier-first-letter:
one of: A..Zora..zor_

identifier-letter:
one of: A..Zora..zor0..9or_

literal:
boolean-literal
character-literal
integer-literal
string-literal

boolean-literal:

`true`
`false`

character-literal:

`'` *single-character* `'`
`'` *escape-sequence* `'`

single-character:

one of: `!`..`~` but not `'` or `\`

integer-literal:

decimal-numeral integer-literal-suffix$_{opt}$
hexadecimal-numeral integer-literal-suffix$_{opt}$
octal-numeral integer-literal-suffix$_{opt}$

integer-literal-suffix:

`l` or `L`

decimal-numeral:

`0`
non-zero-digit digits$_{opt}$

non-zero-digit:

one of: `1`..`9`

digits:

digit
digits digit

digit:

`0`
non-zero-digit

hexadecimal-numeral:

`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-numeral hexadecimal-digit

hexadecimal-digit:

one of: `0`..`9` or `A`..`z` or `a`..`z`

octal-numeral:

`0` *octal-digit*
octal-numeral octal-digit

octal-digit:

one of: `0`..`7`

string-literal:

`"` *string-characters*$_{opt}$ `"`

string-characters:

string-character
string-characters string-character

string-character:

one of: `!`..`~` but not `"` or `\`
escape-sequence

escape-sequence:

\ b
\ t
\ n
\ f
\ r
\ "
\ '
octal-escape

octal-escape:

\ *octal-digit*
octal-escape octal-digit

comment:

single-line-comment
documentation-comment
multi-line-comment

single-line-comment:

plus all remaining text on this line

documentation-comment:

#! plus all remaining text on this line

multi-line-comment:

multi-line-comment-begin intervening text *multi-line-comment-end*

multi-line-comment-begin:

#/* in column 1

multi-line-comment-end:

#*/ in column 1

Bibliography

[1] Adafruit Industries. http://www.adafruit.com.

[2] X. Amatriain. A domain-specific metamodel for multimedia processing systems. *IEEE Transactions on Multimedia*, 9(6):1284–1298, 2007.

[3] X. Amatriain, M. de Boer, E. Robledo, and D. Garcia. Clam: An object oriented framework for developing audio and music applications. In *Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 1284–1298. ACM Press, Nov. 2002.

[4] W. Archer, P. Levis, and J. Regehr. Interface contracts for tinyos. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 158–165, New York, NY, USA, 2007. ACM.

[5] The Arduino Project. http://www.arduino.cc.

[6] Imagining a World of Hardware Mashups, February 8 2009. http://tinyurl.com/22mygnb.

[7] Arduino API Reference. http://arduino.cc/en/Reference/HomePage.

[8] Atmel AVR 8-Bit RISC processor. http://www.atmel.com/products/AVR/.

[9] R. Bannatyne. Microcontrollers for the automobile. http://www.mcjournal.com/articles/arc105/arc105.htm.

[10] M. Bar. Real men program in C. http://www.embedded.com/columns/barrcode/218600142.

[11] A. Benveniste and G. Berry. Readings in hardware/software co-design. In G. De Micheli, R. Ernst, and W. Wolf, editors, *Proceedings of the IEEE*, chapter The synchronous approach to reactive and real-time systems, pages 147–159. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[12] J. Bresson. Sound processing in openmusic. In *Proceedings of the International Conference on Digital Audio Effects*, Sept. 2006.

[13] J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 429–432, Minneapolis, Apr. 1993. IEEE Press.

[14] J. T. Buck and E. A. Lee. *The Token Flow Model*, pages 267–290. IEEE Press, 1995.

[15] Burning Man Festival. http://www.burningman.com/.

[16] A. Chaudhary, A. Freed, and M. Wright. An open architecture for real-time audio processing software. In *Audio Engineering Society 107th Convention*. Audio Engineering Society, 1999.

[17] D. W. Cho, H. S. Kim, and S. Oh. A new approach to detecting memory access errors in c programs. *Computer and Information Science, ACIS International Conference on*, 0:885–890, 2007.

[18] CNN News Network. http://cnn.com/.

[19] Texas Instruments DSP/BIOS Real-Time OS. http://focus.ti.com.

[20] A. Dunkels. Lightweight implementation of tcp/ip protocol suite. http://www.sics.se/~adam/lwip/.

[21] A. Dunkels. Full tcp/ip for 8-bit architectures. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM.

[22] The Eclipse Foundation. http://eclipse.org/.

[23] Echo protocol rfc. http://www.faqs.org/rfcs/rfc862.html.

[24] FreeRTOS. http://www.freertos.org.

[25] E. Gamma, R. Johnson, R. Helm, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[26] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[27] GNU Compiler Collection. http://gcc.gnu.org/.

[28] Gizmodo. http://gizmodo.com/.

[29] GNU Make. http://www.gnu.org/software/make/.

[30] D. Harel and E. Grey. Executable object modeling with statecharts. In *Proceedings, 18th International Conference on Software Engineering*, pages 246–256. IEEE Press, Mar. 1996.

[31] B. Hartmann, L. Abdulla, M. Mittal, and S. R. Klemmer. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 145–154, New York, NY, USA, 2007. ACM.

[32] T. Instruments. Chipcon cc2500. http://focus.ti.com/docs/prod/folders/print/cc2500.html.

[33] T. Instruments. ez430-rf2500 development board. http://focus.ti.com/docs/toolsw/folders/print/ez430-rf2500.html.

[34] Isadora. http://www.troikatronix.com/isadora.html.

[35] H. Ishii and B. Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, pages 234–241, New York, NY, USA, 1997. ACM.

[36] Jameco Electronics. http://www.jameco.com.

[37] The Javelin Stamp Module. http://www.parallax.com/.

[38] Real-time Native Java Module. http://jstamp.systronix.com/Resource/jstamp_datasheet.pdf.

[39] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475, 1974.

[40] A. Kalavade and E. A. Lee. Hardware/software co-design. In *Proceedings of the IFIP International Workshop on Hardware/Software Co-Design*. IEEE Press, May 1992.

[41] T. Kiriyama and M. Sato. Analyzing human behaviors in an interactive art installation. In *Proceedings of the 13th International Conference on Human-Computer Interaction. Part IV: Interacting in Various Application Domains*, pages 345–352, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] National Instruments LabVIEW. http://www.ni.com/labview/.

[43] J. Lahart. Taking an open-source approach to hardware. *Wall Street Journal*, page B8, November 27 2007.

[44] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan. 1987.

[45] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245. IEEE Press, Sept. 1987.

[46] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–799. IEEE Press, 1995.

[47] P. Levis and D. Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[48] Maker Faire. http://makerfaire.com/.

[49] Make: technology on your time. http://blog.makezine.com/.

[50] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 367–371, New York, NY, USA, 2008. ACM.

[51] The Max Programming Environment. http://www.cycling74.com/products/maxoverview.

[52] History of Max. http://freesoftware.ircam.fr/article.php3?id_article=5.

[53] S. McDirmid, M. Flatt, and W. C. Hsieh. Java component development in jiazzi, 2001.

[54] L. Micro. Ek-lm3s6965 microcontroller development kit. http://www.luminarymicro.com/products/lm3s6965_ethernet_evaluation_kit.html.

[55] L. Micro. Ek-lm3s811 microcontroller development kit. http://www.luminarymicro.com/products/stellaris_811_evaluation_kits.html.

[56] L. Micro. Stellarisware peripheral library. http://www.luminarymicro.com/products/software.html.

[57] L. Mikhajlov, E. Sekerinski, and T. C. F. C. Science. A study of the fragile base class problem. In *In European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998.

[58] Texas Instruments MSP430 processor. http://www.ti.com/msp430.

[59] WIRED NextFest Festival. http://www.wired.com/wiredscience/tag/nextfest-2008/.

[60] NXP Semiconductors Microcontrollers. http://www.nxp.com/.

[61] D. Overholt. The musical interface technology design space. *Org. Sound*, 14:217–226, August 2009.

[62] Parallax Basic Stamp. http://www.parallax.com/tabid/295/Default.aspx.

[63] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Universitat Darmstadt, Germany, 1962.

[64] Provider design pattern. http://msdn.microsoft.com/en-us/library/ms972319.aspx.

[65] The Ptolemy Project. http://ptolemy.eecs.berkeley.edu.

[66] M. Puckette. Combining event and signal processing in the max graphical programming environment. *Computer Music Journal*, 15(3):68–77, 1991.

[67] M. Puckette. Pure data: another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[68] Python Programming Language. http://python.org/.

[69] Quartz Composer. http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html.

[70] Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[71] Realtime Software Components. http://eclipse.org/dsdp/rtsc/.

[72] XDCspec Language Reference. http://rtsc.eclipse.org/.

[73] Ruby on Rails. http://rubyonrails.org/.

[74] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. Compiling java for low-end embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 42–50, New York, NY, USA, 2003. ACM.

[75] J. N. Seizovic. The reactive kernel. Technical report, Caltech, Pasadena, CA, USA, 1988.

[76] Sparkfun Electronics. http://www.sparkfun.com.

[77] V. Subramonian, L.-J. Shen, C. Gill, and N. Wang. The design and performance of configurable component middleware for distributed real-time and embedded systems. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[78] C. Szyperski. Independently extensible systems - software engineering potential and challenges -. In *In Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[79] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2002.

[80] TinyOS 2: Hardware Abstraction Architecture. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep2.html.

[81] TinyOS. http://www.tinyos.net.

[82] TinyOS Programming Manual. http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf.

[83] TIOBE Index. http://www.tiobe.com.

[84] B. L. Titzer, J. Auerbach, D. F. Bacon, and J. Palsberg. The exovm system for automatic vm and application reduction. *SIGPLAN Not.*, 42(6):352–362, 2007.

[85] H. W. van Dijk, H. J. Sips, and E. Deprettere. Conext-aware process networks. In *Proceedings of the Application-Specific Systems, Architectures, and Processors (ASAP'03)*, pages 6–16. IEEE Press, June 2003.

[86] Viacar competition. http://ieee.ucsd.edu/viacar.

[87] G. Wang. *The ChucK Audio Programming Language*. PhD thesis, Princeton University, 2008.

[88] B. Weiss, G. Gridling, and M. Proske. A case study in efficient microcontroller education. *SIGBED Rev.*, 2:40–47, October 2005.

[89] M. C. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications*. PhD thesis, University of California at Berkeley, 1998.

[90] Wiring. http://wiring.org.co/.

[91] G. Zhou, M.-K. Leung, and E. A. Lee. A code generation framework for actor-oriented models with partial evaluation. In *Proceedings of International Conference on Embedded Software and Systems*, pages 786–799. Springer-Verlag, May 2007.