Language Support for Highly Resource-Constrained Microcontroller Applications

UCSB Technical Report 2010-16
August 1, 2010

Amichi Amar Chandra Krintz

Media Arts and Technology and Computer Science Departments
Univ. of California, Santa Barbara

Abstract

Modern embedded systems (i.e. highly resource-constrained, microcontroller-based devices) are increasingly available at very low cost and consist of a variety of hardware components. As a result, these systems have become increasingly ubiquitous and are emerging as an important computing platform. Advances in support for easy program development across heterogeneous devices, by a broad developer base (with a range of expertise and backgrounds) for such devices is vital, but unfortunately has not kept pace. Currently, only very skilled developers are able to develop even simple applications or extant software development frameworks only support a small set of similar devices or a particular application domain.

Toward this end, we present a new programming language, called Em, for the development of highly resource-constrained device applications. Em is an extension to the C language that integrates high-level language and software engineering features that include modularity, encapsulation and data hiding, interface separation from implementation, inheritance, support for key design patterns, and reduced syntax verbosity, among others. Em facilitates code reuse, portability across platforms and device components, interchangeability (substitutability) while still achieving the footprint and code efficiency of C. Em also integrates novel features such as unifying the configuration (build time execution) and target (run time execution) code development. We demonstrate the efficacy of Em using multiple embedded systems building blocks and applications.

1. Introduction

Currently, there is a proliferation of microcontroller-based, resource-constrained, digital devices in the market place. 98% of processors produced globally are embedded [4] – 55% of which are 8-bit, microcontrollers with as little as 1KB of RAM and 16KB of program memory. These devices are programmable and offer a wide variety of sophisticated components including accelerometers, gyroscopes, atmospheric sensors, wireless radios, and tiny displays – at very low cost (e.g. see Sparkfun[25], Jameco[13], Adafruit[1]).

These devices are ideal platforms for applications in digital media [5], human interactivity [24], consumer electronics [30], home automation, health and environmental monitoring [9], sensing, control/response systems, etc., and offer compact and esthetically pleasing form-factors, off the shelf and at very low cost. As a result, these systems are increasingly ubiquitous and there is a significant and growing demand for applications for these devices as well as support for tools that aid development of such applications by a wide range of users (designers, artists, hobbyists, students, engineers, etc.).

Unfortunately, development of microcontroller applications is difficult for novice and expert alike. There is a vast diversity of devices from a variety of manufacturers and often, device families from the same manufacturer can differ in architecture, internal memory maps, and operation of their integrated peripherals. Code for one device typically cannot execute on another, and manufacturer-specific development environments make it challenging to write code that is reusable, and functional across platforms. Moreover, applications for these devices typically must be integrated with systems code as opposed to separated via a well-defined interface to an operating system, as in more resource-rich systems. To complicate the matter further, the dominant language for writing microcontroller applications is C.

The low-level nature of C and its compiler maturity and availability enables code to be generated that has a very small footprint and that can operate within severe resource constraints. However, interest in learning and using C is dwindling because it provides little abstraction and many fewer tools to aid the pedagogical and development process compared to more modern, high-level languages. For example, the C preprocessor and build tools such as Make, and vendor-specific environments (e.g. non-portable libraries, tool-chains) constitute the state of the art for embedded systems application development.

The goal of our work is to design and implement a new language that addresses these issues to lower the barrier to entry on development of highly resource-constrained, microcontroller applications. To enable this, we extend C with a small amount of new syntax that implements successful features from modern software engineering and object-oriented computing for more resource-rich systems and languages. We employ C as the base language since it is familiar to many and so that we may exploit mature and highly optimizing (and freely available) compilers.

Our language, called *Em*, employs a modularity-focused constitution and integrates techniques for abstraction, reuse (by a single developer as well as a distributed developer community), portability, and interchangeability. However, we do so in a way that still facilitates efficient code generation, application-system integration, and operation under severe resource constraints. In addition, key to Em's novelty and efficiency is its integration of compile-time configuration within the application program. We have used Em to develop a number of applications for a wide range of devices and device components, from Atmel's AVR and Texas Instrument's MSP430, to NXP's LPC ARM series and LuminaryMicro's line of ARM Cortex-M3 devices. We evaluate Em using embedded systems building blocks and find that, despite the higher-level of abstraction that Em provides, the compilation system is able to generate very efficient code. We also employ Em for more sophisticated applications and discuss how Em enables reuse, portability, and interchangeability for these systems.

In summary, with this paper, we contribute

- a new programming language called Em that brings successful language and software engineering techniques to the familiarity of C for the development of highly resource-constrained microcontroller-based applications,
- language and translation (enforcement) support for modularity, encapsulation, data hiding, and code reuse using techniques from modern high-level object-oriented languages including C++ and Java,
- a reduction in the verbosity of the C language (similar to that done in modern scripting languages such as Python and Javascript),
- 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 (called templates),
- a demonstration of code reuse, portability, interchangeability, and integration of legacy source by Em applications for sophisticated embedded systems components (device drivers, TCP/IP stack),
- an evaluation of the footprint of Em applications and a comparison to related systems.

In the sections that follow, we describe Em and provide examples of its key features. Em is currently in use for an introduction to embedded systems programming class for undergraduates (with a range of programming skills) in computer science at a university (details omitted for anonymity reasons).

2. The Em Language

Em is a high-level programming language for writing embedded systems software. It embodies concepts from other high-level languages that enable straightforward reusability of code, interchangeability of software components to cope with change and variation, and development of code that is usable across microcontroller devices, i.e. portability. In particular, we have designed Em with the notion that modularity, or the separation of concerns in code, should be preserved at all cost in order to enable an ecosystem of software components to exist that are developed independently by people all around the world and can be easily shared through the Internet.

While Em provides high-level programming features it does not add significant runtime overhead. Em code is translated to standard ANSI C and the Em translator enforces the semantics of the language. The translation process results in C code that can be compiled by any standards-compliant C compiler. Since C is currently the dominant language of embedded systems development [4], most microcontroller manufacturers support an optimizing C compiler for their devices. Furthermore, most C compilers, if fed an entire application in a single source file, can perform whole program analysis and aggressively optimize the code by inlining functions, folding away constants etc. The resulting binaries, therefore, can be more optimized than handwritten C code compiled piecewise and linked together. Em takes full takes advantage of modern C compiler technology to achieve its efficiency of generated code.

Since software written in Em must be built for, fit into, and run efficiently on a wide range of highly resource constrained hardware, the ability to configure, or tune, software components at build time, given knowledge of other components present in the application and the hardware it will run on is essential. To this end, unlike any other language in this domain, we holistically integrate into the Em language, the application build process. Code residing in an Em file can be target code, which provides runtime functionality on the device itself, or it can be configuration code which executes on the build host and serves to configure the module at build time for a particular hardware/software configuration. This enables modules to adapt themselves to a given configuration and potentially off-load tasks that would normally take place at runtime on the target, onto the build host that has significantly more resources.

In addition, our Em design employs a usage model that is similar to some degree to scripting languages such as

Javascript, Python and Ruby and also to object-oriented languages such as Java, C++ and others. As per the former, Em encourages development of code that largely depends on existing functional content to realize its task. As per the latter, Em provides rich abstractions that are suitable for complex architectural patterns.

To implement Em, we extend the C language with approximately 30 key words and new syntax patterns. Our Em grammar specified using Antlr [19] is available online at *(url omitted for anonymity purposes)*; we omit it here for brevity.

In the subsections that follow, we describe each of the key features of Em and provide examples. Across all examples, we employ most of the new syntax that we extend the C language with to specify Em. That is, Em adds very little to C to enable the benefits we are after (modularity, reuse, and portability with highly compact code generation). The key features that Em implements are modules, interfaces and proxies, composites, templates, and a restricted form of inheritance. We detail these features and discuss the build and translation process.

2.1 Modules

The fundamental unit of code in Em is the module. Figure 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 access the public features of other modules. We refer to modules that do so as *clients* of the module they access. Modules in Em are singletons 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 is a data type defined in the public specification of the module BasicListManager).

Function declarations take the form `function fnName` $([paramType]^*)$ `: returnType`, where `fnName` is the function name, $[paramType]^*$ are 0 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.

Function definitions follow the specification blocks in a module. Function definitions take the form `def fnName` $([paramName]^*)$ $\{\ldots\}$. 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, semicolon is not needed to end a line or declaration; types are omitted in 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.

```
package em.bios

module EventDispatcher {

    proxy GlobalIntr implements GlobalInterruptsI
    type EventHandler: Void( e: Void* )

    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 1. An Em module with public and private specification.

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

Figure 2. Dispatch function from the EventDispatcher module.

Module data is represented by variables with primitive and composite data types. Module variables must be declared within the public/private specification blocks. The only other types of variables are local variables within functions. Em has all data types present in C, although we require that the widths of primitive types be specified explicitly where they can vary. Integers, for example, are specified with their width and sign, such as Int8 or UInt8; Int does not exist. 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 [6] 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* semantics of C++ using the & operator (following the type name, e.g. `var event : Event&`) for primitive and composite data types. Em references can be assigned a null value at declaration and may be reassigned after declara-

tion unlike C++ references. Reference types provide developers with an option of a safer alternative to pointers. Pointer types are allowed and have the same semantics as in C and C++. 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.

Em uses the keyword `type` to rename a type, e.g. `typedef` in C. Em also use this keyword to introduce a new composite data type not present in C, called an *opaque* type. An opaque type is similar to a struct in C but contains function as well as data declarations. The data within an opaque type can only be accessed and operated on via these functions. The function definitions are located with the module functions.

The EventDispatcher module in Figure 1 is an asynchronous function dispatching scheduler. It declares an opaque type named `Event` and lists the functions the module contains. The data associated with the Event type is defined in the private section of the module and reflects the private nature of the opaque type representation and may consist of primitive, or composite data types (including another opaque type). The module defining the opaque type may also access the representation. Figure 2 shows the definition of the `dispatch()` function in this module. This function accesses the `handler` property of the `Event` opaque type (which is employed as a reference type).

Figure 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`.

2.1.1 Module Configuration

The transformation of one or more Em modules into a binary image takes place on a resource-rich host such as a personal computer. We refer to this process as build time. During build time, every module that is part of an application is capable of being configured for that application's needs, for a microcontroller target, and a specific hardware configuration. The Em language has two mechanisms for modules to configure themselves and other modules during build time: configuration parameters and host functions.

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 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 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.

```
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 ) {
    Led.toggle()
}
```

Figure 3. Using EventDispatcher to Blink an LED

The *host* keyword also applies to functions. Host functions implement code that executes on the 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 [20]. 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. Since both the host and target code are implemented in Em, it unifies the syntax and combines the development and con-

figuration. That is, developers need not learn multiple languages (e.g. as for 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 1) is an example of a host function.

Finally, every module in Em must reside within a package (e.g. namespace). The `EventDispatcher` module for example, resides within the `em.bios` package. Modules can bring another module into their namespace using the `import` statement. 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 3.

2.1.2 Module Intrinsics

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

The intrinsic functions relevant at build time are implicitly `host` functions and include `em$configure`, `em$construct`, and `em$generateCode`. `em$configure` and `em$construct` are used to initialize the state of a module's public and private features respectively. For example, proxies - Em's mechanism for decoupling a module's interface from implementation (more on proxies below) - that are public, are often bound to concrete implementations in `em$configure`. The `em$construct` function is used to initialize a module's private state, that is, to construct the module itself. Figure 3 shows the BlinkP module's `em$configure` and `em$construct` functions which we use to initialize the module's public and private state respectively.

`em$generateCode` is an intrinsic that can be used to inject code into a module's C representation during translation from Em into C. A simple use of `em$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.

The intrinsic functions associated with a module's run time life-cycle 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. Upon powering up the microcontroller, the first function called, if it exists is `em$reset`, followed by all occurrences of `em$startup`, proceeded by `em$startupDone` and finally `em$run` which is Em's equivalent of main() in C. 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 run time.

```
interface InterruptSourceI {

    type Handler: Void()

    host function setHandlerOnHost( h: Handler ): Void

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

Figure 4. An interrupt source interface.

```
module Led {

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

    proxy Pin implements GpioI
}

private {
    config activeLow: Bool
}

...

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

Figure 5. Part of an LED module using a proxy.

2.2 Interfaces & Proxies

To support software variability and change, Em provides interfaces and proxies, which together, decouple a module's interface from its implementation. An Em interface, depicted in Figure 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 opaque types declared in the interface they implement. Interfaces in Em resemble and function much like interfaces in Java.

Em introduces a language construct, based on a common software design pattern, known as a proxy [8]. 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 declare proxies

```
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 Uart

from em.mcu.atmega168 import PD0 as D0
from em.mcu.atmega168 import PD1 as D1
...
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 Uart
    ...
    export D0
    export D1
}

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

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

Figure 6. A composite representing a board configuration.

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.

Before a module that declares proxies can be utilized, its proxies must be bound to modules that implement the same interface as the proxy. The binding of proxies occurs automatically at build time. Figure 5 shows an Led module that makes use of a general purpose I/O pin (GPIO) proxy (called `Pin`) for its implementation.

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 substituted 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.

2.3 Composites

A composite in Em is a special module that is used to configure proxies and configuration parameters of other modules. Composites are special in that they contain no target code, i.e. they are host-only modules. Figure 6 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.

In Figure 3, BlinkP imports the Led and TimerMilli0 modules from BoardC of Figure 6. 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 3 imports an LED and millisecond timer from the BoardC composite shown in Figure 6. 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. Applications of all complexities can follow this pattern to decouple their dependencies on low-level hardware specific implementations and achieve portability.

2.3.1 Composite Intrinsics

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 in the build process, change the configurations that take place here.

`em$preconfigure` is where immutable configurations occur. Since the BoardC composite in Figure 6 reflects some hard-wired aspects of an application's hardware, such as its oscillator's frequency and a GPIO pin connected to an Led, these aspects are configured such that no other module may change them. Our translation system ensures that no other module changes immutable configurations of any composite.

2.4 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 7 shows an Em template.

A template intrinsic, `em$generateUnit`, is defined and is invoked at build time for each template instantiation that is encountered. Figure 6 shows an instantiation of an Led mod-

```
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 7. A template to generate GPIO modules.

ule 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.

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. The public specification of a template specifies parameters that are necessary for its instantiation. A private specification can exist to define host configuration parameters and functions the template may need to carry out its operation.

As an example, the operation of a general purpose IO pin is identical for all pins although each pin has a distinct bit in a specific register it must operate on. In Figure 7, 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 statement `import GpioT {port: "P1", pin: 0} as P1_0` will create a GPIO module with the name P1_0.

2.4.1 Template Intrinsics

In addition to the intrinsic function `em$generateUnit`, templates have two often used intrinsics, `em$packageName` and `em$unitName`. Since every module resides within a package the value of `em$packageName` is always defined

```
interface GpioI {

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

Figure 8. Part of an interface for an I/O pin.

to be the name of the package the template is instantiated in. For the Led module instantiated from template LedT in Figure 6, `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 6.

2.5 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 9. 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.

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 [18, 27]. Such issues can lead to a breach in modularity, which with Em, we attempt to maintain strictly.

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 8 and another module N implements the EdgeDetectGpioI interface which extends the interface of GpioI in Figure 9. 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 N, as shown in Figure 10. 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.

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

```
interface GpioEdgeDetectI extends GpioI {

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

Figure 9. An interface that extends GpioI

```
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 10. Example of implementation inheritance by forwarding.

to automate the definition of functions that forward calls to save the programmer from typing them.

2.6 Interaction with C from Em

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.

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 11 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.

2.7 From Module to Binary Image

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 12 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.

```
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 11. Part of a UART module interacting with C identifiers.

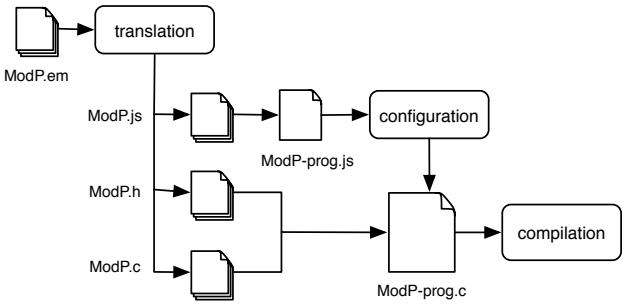

Figure 12. Build flow of the Em translator

Each Em source file (a module, interface, composite, or template) represents an independent unit of translation. From Figure 12, 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.

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 imple-

mentation was arbitrary and other languages could have been used as the underlying configuration language.

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

3. Demonstrations of the Use and Efficacy of Em

In this section, 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 for a low-power wireless sensor node, a Luminary Micro ARM Cortex-M3 system, and μIP, a popular TCP/IP stack for resource constrained microcontrollers. We also consider a number of different embedded systems building blocks to evaluate memory footprint relative to related systems.

3.1 Reuse & Portability

We first investigate the reuse and portability of code that Em enables using an implementation of a radio transceiver module. For this experiment, we use a development board [12], representative of low-power wireless sensor network nodes, with a low-power microcontroller and the popular ChipCon CC2500 2.4GHz radio transceiver [11].

Since the radio transceiver is 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 microcontroller configures and interacts with the device via its SPI port. Multiple SPI ports could be resident on a microcontroller and many 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 from its implementation of low-level hardware interface.

```
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 13. Portion of CC2500 radio module.

To achieve this decoupling in Em, we create two interfaces, SpiI and GpioI. Within each, we define the essential functionality of each peripheral, informed by [28]. Implementations for controlling each peripheral are written for the development board's microcontroller, a Texas Instruments MSP430.

Figure 13 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, Em makes it possible to bind different SPI ports and Gpio pins according to different physical configurations of the hardware.

Figure 14 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 interface to the microcontroller are now configured. Should a different piece of hardware using the CC2500 radio be physically wired differently,

```
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 14. BoardC composite for wireless device

```
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 15. Module and interface used for radio configuration.

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. They 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. To support this variability without requiring modification to the driver itself, we provide an interface to it called CC2500ConfigurationI. The interface defines one host function: setContextValuesOnHost. The module CC2500Context comprises the driver's configuration options.

```
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 16. Application-specific radio configuration.

Figure 15 depicts the interface and module. The CC2500 module contains a proxy, shown in Figure 13, 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 16.

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 16. 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. 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.

3.2 Reuse of an existing C library

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

```
template StellarisWareT { }

def em$generateUnit() {

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

Figure 17. Template for modules wrapping library functionality.

```
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 18. Composite to instantiate modules from template.

of drivers. This software package includes the driver code for all peripherals integrated into their ARM Cortex-M3 devices [16]. 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 17 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 18 shows part of the composite and its exports for modules that represent peripheral drivers from the library.

To enable applications to be written independent 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 from the composite the particular peripheral module that it requires. Figure 19 shows a millisecond timer module, `TimerMilli32BitTimer0`. This module imports from StellarisWareC, the modules `Sysctl` and `Timer`. Any

```
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 19. Millisecond timer using driver library functions.

symbol defined in the source files of these two peripherals is accessible from within `TimerMilli32BitTimer0`. The `start` function of `TimerMilli32BitTimer0` 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 the libraries header files and linking the application against the library during build-time.

3.3 Improvement upon an existing C library

We next demonstrate how Em can be used to refactor existing code to avoid redundancy and to separate concerns. For this investigation, we employ μIP [7], a popular TCP/IP stack for resource constrained microcontrollers.

For most microcontrollers, RAM is the most scarce resource, μIP has been designed to utilize a single statically allocated packet buffer for both incoming and outgoing packets. Other microcontrollers may have sufficient RAM to support a separate buffering policy, which may lead to more efficient networking. For such devices, the authors of μIP, also provide lwIP (light-weight IP) which provides separate buffer support of incoming and outgoing packets.

With Em, we have been able to provide a single implementation of this TCP/IP stack, the buffer implementation for which can be configured by the application. Given that much of the code in the two original implementations was duplicated (except for the buffer implmentation), an extensible Em implementation has the potential for reducing lines of codes, programmer errors (that are potentially propagated by the copy), and code stability (the base implementation

```
interface BufferManagerI {

    type Buffer: opaque {
        function getBuffer(): UInt8*
        function getSize(): UInt16
        host function initH( size: UInt16 ): Void
    }

    function freeBuffer(): Void
    function getEmptyBuffer(): Buffer*
    function hasEmptyBuffer(): Bool
    host function setMaxBufferSizeH( size: UInt16 ): Void
}
```

Figure 20. A packet buffer manager interface.

```
import BufferManagerI
import BufferManagerProviderI

module BufferManager implements BufferManagerI {

    proxy Provider implements BufferManagerProviderI
}

private {

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

def freeBuffer() {
    Provider.freeBuffer()
}

def getEmptyBuffer() {
    return Provider.getEmptyBuffer()
}
```

Figure 21. A packet buffer manager interface.

can remain unchanged while new buffer implementations and management algorithms can be implemented and evaluated. At the same time, since the Em build process only integrates the code that the application uses, an Em implementation uses no more code than the original.

Figure 20 shows the packet buffer manager interface that defines a packet buffer data type and the functions to be implemented by buffer managers. To enable the network stack implementation to use one buffer manager module with a consistent buffer type definition, we create the module in Figure 21. The `BufferManager` implements the definition of the buffer type and uses a proxy that implements the `BufferManagerProviderI` interface. All `BufferManager` functions then call the proxy's functions of the same name. `BufferManagerProviderI` in Figure 22 extends `BufferManagerI` and currently only serves to qualify the implementing modules as providers.

Implementations of `BufferManagerProviderI` are now responsible for defining the actual policy for buffer management. Many implementations can exist. One, for example, with a single statically allocated buffer, or another with multiple buffers. Binding of an implementation to the `Provider` proxy of `BufferManager` identifies the implementation that is to be used. This process requires no modification to the network stack itself. Applications, based on their hardware resources, can easily modify buffer policies to their requirements. In conjunction with host functions and the ability to introspect on module's configuration parameters at build time, the stack could even be written such that it automatically selects the ideal buffer configuration given specified availability of resources.

```
import BufferManagerI

interface BufferManagerProviderI extends BufferManagerI {
}
```

Figure 22. A buffer manager provider interface.

3.4 Empirical Measurements

In this section, we investigate the efficiency of the code generated from Em applications. The key metric for our device domain is memory footprint: code and data size, as many of our devices have only a few kilobytes of memory within which all application and system code must run.

We first evaluate the memory footprint for four building block components typically implemented within a wide range of embedded systems applications. The programs we consider are Null, Blink, Sense, and SenseMod. The Null or empty application demonstrates the bare minimum program and reflects the lowest overhead of each system. Blink toggles an LED periodically using an deferred function invocation dispatcher and hardware interrupts to dispatch a single event whose handler accesses the hardware synchronously.

Sense 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. SenseMod reads an A/D value asynchronously and uses the value to modulate the blinking rate of an LED. This application uses multiple interrupt sources and manages multiple concurrent events from asynchronous processes that interact with hardware.

We compare Em code to that produced by NesC/TinyOS and Arduino systems. These toolchains are those that are the most similar to Em (NesC/TinyOS) or which target the same application domain and users (Arduino). We compare and contrast these systems with Em in Section 4. We have implemented each of our building block applications using each of these three systems (NesC/TinyOS, Arduino, and Em).

For the Em runtime, we implement a simple event handler and scheduler using Em itself (that we reuse for each application). We made our best effort to implement the same semantics of these components as those that are released as part of the NesC/TinyOS system. We distribute this asynchronous event-handling code and our building block applications as part of the Em development environment.

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 23. Resource usage for embedded systems building blocks.

Arduino does not provide runtime support for concurrency. We thus, implemented the sample applications in Arduino using only the functionality the platform provides its users [3] so that the program semantics are as similar as possible. We use strictly sequential, synchronous, code for these applications. While the implementations behave similarly to 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 which leads to significant redundancy and a lack of reusability.

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 report memory footprint (program and data), the number of source files required for development, and the number of lines of code in the source files. We present our results in Figure 23.

The results show that implementations in Em consume 44% - 62% less program memory than those of the other systems. Em offers the same or less (30%) data memory consumption as TinyOS for equivalent functionality. The savings in program and data memory is in part 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 savings in memory enables more complex functionality to fit into the limited resources of the microcontroller. 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.

We also include file count and lines of code as part of this table for each of the applications. TinyOS programs require 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 (located in a separate file). In the Em applications, since modules are 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. Arduino has fewer lines of code and smaller footprint in some cases because it does not provide event handling support.

We next present various metrics (footprint and in some cases cycles executed) for the various applications and test cases that we distribute with the Em development environment.

- *Temperature Sensing Application*. This application is written for the Texas Instruments ez430-RF2500 development board. The board contains an MSP430 low-power microcontroller and the ChipCon CC2500 low power 2.4GHz radio transceiver. The `end point` application reads the microcontroller's temperature from an internal thermometer and sends it to the `access point`. The access point listens to its radio for incoming data and when data arrives, the access point communicates the value to a host computer to which it is connected via USB. The host prints the value to its screen.
  - Temperature Sense Endpoint Application:
    Program Size: 2565 bytes
    Data + BSS: 150
  - Temperature Access Point Application:
    Program Size: 2547 bytes
    Data + BSS: 160
- *Network Stack Application*. This application is the μIP TCP/IP stack to which we refer in our demonstration of code reuse above. We implement this application for the Luminary Micro LM3S6965-EK evaluation board that has an ARM Cortex-M3 based LM3S6965 microcontroller with integrated ethernet MAC. These results employ IP/ICMP protocols only and implement the the buffer manager with a single statically allocated buffer.
  - Basic (IP/ICMP) Network Stack with BufferManager
    Program Size: 8828 bytes
    Data + BSS: 1436

Our test suite consists of a number of different applications that we have developed to test and evaluate the functionality of the Em implementation for different platforms. We present data for three different microcontrollers each running three different test applications. The devices are an

8-bit AVR ATMega168 device with 1KB of RAM and 16KB of flash, a 16-bit MSP430 with 1K of RAM and 16KB of Flash, and a 32-bit ARM Cortex-M3 LM3S811 with 8K of RAM and 32K of Flash.

Our benchmarks are BenchmarkP, LatencyP, and TimerP. BenchmarkP counts the CPU cycles to execute three empty for-loops 1, 10, 100, 1000 iterations. LatencyP measures the CPU cycles to get from an interrupt service routine to the event handler of an event that was posted through the EventDispatcher. The last program tests the functionality of a virtual timer module that can have N different instances (virtual timers) for one hardware timer.

Table 1 shows the results. Overall, Em is able to generate compact code and data sizes for each of the devices we investigate.

In summary, we find that Em provides developers with many of the programming advantages facilitated by languages for more resource-rich systems (C++, Python, Ruby). It does so by incorporating into C, a subset of features from these languages (modularity, composition, inheretance, separation of concerns, separation of interface from implementations, 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. At the same time, Em does so in a way that enables the resulting applications to operate under the severe resource constraints of modern microcontroller-based systems.

BenchmarkP				
MSP430 (cycles)	8	72	702	7002
Program Size (bytes)	1046			
Data+BSS Size (bytes)	34			
ATR ATmega168 (cycles)	4	40	1001	11001
Program Size (bytes)	1616			
Data+BSS Size (bytes)	70			
LM3S811 (cycles)	10	21	611	6012
Program Size (bytes)	3356			
Data+BSS Size (bytes)	520			
LatencyP				
MSP430 (cycles)	104			
Program Size (bytes)	1615			
Data+BSS Size (bytes)	99			
ATR ATmega168 (cycles)	132			
Program Size (bytes)	1646			
Data+BSS Size (bytes)	64			
LM3S811 (cycles)	79			
Program Size (bytes)	3220			
Data+BSS Size (bytes)	545			
TimerP				
MSP430				
Program Size (bytes)	1783			
Data+BSS Size (bytes)	67			
ATR ATmega168				
Program Size (bytes)	2112			
Data+BSS Size (bytes)	84			
LM3S811				
Program Size (bytes)	3552			
Data+BSS Size (bytes)	588			

Table 1. Benchmark performance results for three development boards containing different microcontrollers (MSP430, AVR ATmega168, and LM3S811). The data is for three of the Em test programs. BenchmarkP shows cycle times for 1, 10, 100, and 1000 iterations of a for loop as the first entry for each device. LatencyP shows the number of cycles between an interrupt service routine and an event handler as the first entry. The other entries for each device show program footprint: number of bytes for the program and data/BSS, respectively.

4. Related Work

Em is the first language to integrate modern object-oriented and high-level language techniques into C to facilitate development of reusable, portable, and interchangeable code that executes on a wide range of severely resource-constrained microcontroller-based devices. The languages and systems related to this include RTSC/XDC [21], NesC [10], Arduino [2]/Wiring [32], and other environments for higher-level languages. Code generation for embedded devices exists in some tools [33] and embedded systems projects have been realized with these systems [14, 31], however they are targeted for use by researchers and highly specialized experts with embedded systems engineering experience.

The RTSC/XDC toolset provides component-oriented microcontroller programming facilities. This toolset however targets devices with significantly more resources than are available in our domain. Moreover, the toolset requires the use of three different programming languages to develop a single application; an interface specification language for describing component interfaces (xdcspec [23]), a configuration language for configuring components (xdcscript [22], and target language for device code (C).

NesC is a programming language for wireless sensor networks that extends the C language. NesC was designed to embody the structuring concepts and execution model of TinyOS [28]. TinyOS is an event-driven operating system designed for sensor network nodes that have very limited resources. There are many similarities between NesC and Em that include the module-oriented approach, separation of implementation and interface, and event-oriented and highly resource-constrained target devices. We could have used NesC as a basis for Em. We investigated doing so and found that NesC's requirements of creating configurations to wire together modules to be very complex and verbose. Many NesC/TinyOS developers (including ourselves) find the NesC/TinyOS wiring, fine-grained interfaces, and the split-phase model non-intuitive and very challenging to learn even for those with experience and expertise with the target devices and the C language. We thus started from scratch to reduce the verbosity required by Em and to introduce a small number of new keywords to the language in an effort to reduce complexity of both the translator, scheduling system, and application development.

Other differences between Em and NesC/TinyOS is that Em modules and composites are available for a much more diverse set of devices, Em is not domain specific (NesC/TinyOS are only used for wireless sensor network devices even though it has been available since early 2000), existing C source code can be integrated easily into Em applications, and Em unifies configuration and target code development using a single language (no complex Make files are employed by the build process or require modification by developers). In addition, since the event model for Em is implemented in Em, any model can be developed and integrated into an application. NesC/TinyOS specify a particular asynchronous event model that is very difficult to change as the design of the language and OS assume that the model is in use. We show in our experimental evaluation that even though Em provides these additional high-level features to developers, the system is able to generate code that is similar (or smaller) in footprint to NesC/TinyOS code.

Other related work has been to extend higher level languages with constructs that better enable modularity, componentization, or support for construction of embedded systems applications. For example, Jiazzi [17] 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 [29] 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 [26]. Both of these works provide insight into optimization techniques applicable to the build process in Em. However, they target more resource-rich environments than our domain of resource-constrained embedded systems.

Higher level programming environments such as LabView [15] provide a component-oriented usage model. Unfortunately, the use of resource-constrained devices is via a tethered, remote-control method of operation where the main application code executes on resource rich hosts as opposed to on the microcontroller.

Wiring and Arduino evolved out of a need to enable a specific, non-technical, user community to develop applications on highly resource constrained microcontrollers based on the Arduino device. Wiring and Arduino lower the barrier to entry to a wide audience of developers, however, lack a structured modular programming model to enable reuse and a runtime system to support concurrency in applications. 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 Arduino hardware. Beyond this API, the Arduino community has code supporting different device hardware 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.

5. Conclusions

We present Em, a high-level, modularity-focused, programming language for development of the next-generation of embedded systems applications for microcontroller-based devices. Em is an extension to the C language that incorporates features from popular high-level languages for more resource-rich environments (C++, Python, Javascript) and successful techniques from modern software engineering. In particular, in Em everything is a module with no cross-module data dependencies, the syntax reduces verbosity, Em enforces support for data hiding, encapsulation within types (via the opaque type) and modules, and Em enables reuse through template (automatic module generation from extant code) and interface inheritance. In addition, Em also unifies the programming of configuration (executed on the host at build time) and target (executed on the device at run time) tasks.

The Em translator implements checks to enforce these concepts and to identify errors during the build process. It

then converts Em to C, integrates extant C code if any, and employs modern available C compilation for aggressive optimization and efficient code generation. We demonstrate the efficacy of Em for code reuse and portability, for integration of existing source and library code into Em applications, and for code stability through the separation of concerns, using sophisticated applications (device drivers, TCP/IP stack). We show that Em, despite its integration of the high-level programming features is able to generate code that has very small footprint and that operates on a wide variety of different highly resource constrained microcontroller-based systems.

References

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

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

[3] ArduinoReference. Arduino API Reference. `http://arduino.cc/en/Reference/HomePage`.

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

[5] L. Buechley and M. Eisenberg. The lilypad arduino: Toward wearable engineering for everyone. *IEEE Pervasive Computing*, 7(2):12–15, 2008. ISSN 1536-1268. doi: http://dx.doi.org/10.1109/MPRV.2008.38.

[6] 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. doi: http://doi.ieeecomputersociety.org/10.1109/ICIS.2007.33.

[7] 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. doi: http://doi.acm.org/10.1145/1066116.1066118.

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

[9] M. Gao, F. Zhang, and J. Tian. Environmental monitoring system with wireless mesh network based on embedded system. In *SEC '08: Proceedings of the 2008 Fifth IEEE International Symposium on Embedded Computing*, pages 174–179, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3348-3. doi: http://dx.doi.org/10.1109/SEC.2008.28.

[10] 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. ISBN 1-58113-662-5. doi: http://doi.acm.org/10.1145/781131.781133.

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

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

[13] Jameco. Jameco Electronics. `http://www.jameco.com`.

[14] 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.

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

[16] I. Luminary Micro. Stellarisware peripheral library. `http://www.luminarymicro.com/products/software.html`.

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

[18] 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.

[19] T. Parr. Antlr. `http://www.antlr.org/`.

[20] RhinoJS. Rhino: JavaScript for Java. `http://www.mozilla.org/rhino/`.

[21] RTSC. Realtime Software Components. `http://www.eclipse.org/dsdp/rtsc/`.

[22] RTSCxdcscript. The XDCscript Language. `http://rtsc.eclipse.org/docs-tip/The_XDCscript_Language`.

[23] RTSCxdcspec. XDCspec Language Reference. `http://rtsc.eclipse.org/docs-tip/XDCspec_Language_Reference`.

[24] R. Sankaran, B. Ullmer, J. Ramanujam, K. Kallakuri, S. Jandhyala, C. Toole, and C. Laan. Decoupling interaction hardware design using libraries of reusable electronics. In *TEI '09: Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, pages 331–337, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-493-5. doi: http://doi.acm.org/10.1145/1517664.1517732.

[25] Sparkfun. Sparkfun Electronics. `http://www.sparkfun.com`.

[26] 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. ISBN 0-7695-2247-5. doi: http://dx.doi.org/10.1109/REAL.2004.53.

[27] C. Szyperski. Independently extensible systems - software engineering potential and challenges -. In *In Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[28] TinyOS. TinyOS: An open-source OS for the wireless embedded sensor networks. `http://www.tinyos.net`.

[29] 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. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250775.

[30] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.825699.

[31] M. C. Williamson. *Synthesis of Parallel Hardware Implementations from Synchronous Dataow Graph Specications*. PhD thesis, University of California at Berkeley, 1998. URL `http://ptolemy.eecs.berkeley.edu/publications/papers/98/SDFToParallelVHDL/mwilliamsonThesis.pdf`.

[32] wiring. Wiring. `http://wiring.org.co/`.

[33] 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.