An Operating System in Java for the Lego Mindstorms RCX Microcontroller

Pekka Nikander

Helsinki University of Technology
Pekka.Nikander@hut.fi
http://www.tcm.hut.fi/~pnr/rcx/

Abstract

The Lego Mindstorms is a Lego bricks based robotics toy series produced by the Lego Group, based on the ideas developed at the Massachusetts Institute of Technology in the Programmable Brick project. The heart of a Lego robot, the RCX microcontroller, hosts a Hitachi H8 microcontroller with 28 kilobytes of memory available for downloadable firmware and applications. In addition to the GUI based programming environment provided by Lego, a number of alternative programming environments have been developed for the RCX. However, these alternative programming environments are written in C, tightly bound to the hardware, and provide only relatively low level services. The strong hardware dependency makes it hard to debug programs; in practice, a hardware simulator is needed, and such a simulator does not yet exist in an open source form.

In this paper we present a new type of operating system and new programming environments for the Lego RCX brick. The operating system is written almost completely in Java, and currently provides runtime support for Java, C and C++ programs. In the case of Java applications, simulation and debugging is relatively simple as it can be performed on a standard Java Virtual Machine with just a small hardware simulation package.

1 Introduction

Introduction of an affordable robotics development kit in the form of the Lego Mindstorms Robotics Invention System (RIS) [1], and its predecessor, the MIT programmable brick [2], has fostered a number of very different efforts for both teaching robotics and experimenting with non-traditional applications of robotic equipment. These approaches include, for example, the idea of using a hoard of Lego robots to collectively perform a larger task.

A Lego Mindstorms Robot consists of a programmable Lego brick, called the RCX, which contains three sensor inputs, three actuator outputs, four user buttons, a simple LCD display, an IR transceiver, and a Hitachi H8 microcontroller with 32 kilobytes of RAM, 4 kilobytes of which is used for interrupt vectors and other low level data. Normally, the RAM is used to host a firmware program, provided by Lego, which is used to interpret the actual user program. The user program is represented in the form of byte code [3]. The byte code itself, in turn, is generated on a Windows PC running a graphical programming environment, which is tightly bound to the Windows operating system. Currently, the byte code offers fairly limited view to the power of the RCX; for example, it allows only 32 variables to be used.

Since the two programming environments [4][5] provided by the Lego group are severely limited in their ability to fully utilize the computational power of the RCX brick, a number of independent programming environments have been developed for the RCX, including Not Quite C (NQC) [6], LegOS [7], and `librcx`, a minimal runtime library [8]. The latter two of these use the GNU C Compiler, part of the GNU Compiler Collection (GCC) [9], to compile C source code to the machine code of the Hitachi H8 processor.

In this paper, we present *RCX Java Operating System*, which is an experimental operating system for the RCX microcontroller. To compile our OS, we have used a modified GNU Java compiler, GCJ, from the GCC suite of compilers. The GCJ compiler compiles Java source code and byte code into the native code of the target machine. We used a cross compiler running on a FreeBSD PC with the Hitachi H8 as the target environment. In contrast with the other existing Java runtime environments, ours is almost completely written in Java, which was possible since all the Java code is being compiled into native code. In the project, getting the compiler and the runtime to function together was one of the

most interesting tasks. Furthermore, a number of interesting tricks were needed to represent non-object data structures in a beautiful way in Java code. Some of the design choices are explained in Sections 3 and 6.

In addition to being a neat way of making it possible to use Java to write programs for the RCX, our approach has a number of potential benefits. First, because the operating system is written in Java, it is fairly independent of the hardware, making it relatively easy to port to other microcontroller based systems such as the uClinux SIMM [10]. Second, as the operating system is written in an object oriented language, it is possible to extend the operating system using the object paradigm. Third, in Java the thread and synchronization models are tightly integrated into the language, making it natural to use threads when developing applications for the RCX. And fourth, the compiler and runtime system can be extended to support additional Java based technologies such as Jini [11]. Some of these aspects are further explored later in this paper, while others are left for future work.

The rest of this paper is organized as follows. In Section 2 we describe the hardware and ROM structure of the RCX microcontroller in more detail. Next, in Section 3, we briefly outline the structure of the GCC compilers, focusing on the GCJ native Java compiler, and describe the modifications we have made to it. Section 4 describes the minimal Java runtime that we developed to support Java based native code on the RCX, and in Section 5 we explain the RCX specific operating system services implemented as well as the interface from the Java environment both to the routines implemented in the RCX ROM and to the actual hardware. Finally, Section 6 summarizes the benefits and lessons learned so far, while Section 7 outlines some possibilities for future work.

2 Lego RCX microcontroller

The Lego RCX (see Figure 1) is a large Lego brick hosting a battery case, a Hitachi H8 microcontroller, an IR transceiver, a simple LCD panel, a few control buttons, sensor and actuator connectors shaped into the form of Lego brick connectors, and some auxiliary circuitry.

The standard programming environments provided by Lego [4][5] allow only a very limited access to the resources of the microcontroller. Basically, they are aimed to enable high school students to apply their knowledge of using Lego blocks in building physical structures on the domain of building logical program structures that

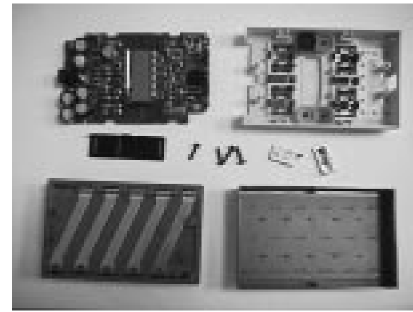

Figure 1: The Lego RCX, opened [12]

control the robot. However, our goal was to enable full access to the microcontroller resources. Therefore, only the low level programmer's view of the RCX is described. The following information is largely based on the reverse engineering work performed by Kekoa Proudfoot and others [12].

When programming at the machine language level (either using an assembler or through a high level language), a programmer has direct access both to the hardware and to the ROM routines. Hitachi H8 uses memory mapped I/O, and the actual hardware ports are mapped to the highest part of the memory, at a so called eight bit area. The I/O port map, as used for RCX, is partially illustrated in Table 1. [12]

Memory Range	Function
`F000 - F0FF`	Motor control
`FFB7`	IR transceiver range, button input
`FFBA`	IR control, external RAM power save mode
`FFBB`	Sensor power, timer, LCD I/O
`FFBE`	Sensor input, button input
`FFC3`	Serial/Timer control
`FFE0 - FFE4`	Sensor A/D input

Table 1: Some RCX I/O addresses

The ROM contains a fairly large number of routines, and it is beyond the scope of this paper to describe them all. However, the routines include the following functionality:

- Initialization functions and a simple main loop
- Default interrupt handlers
- Memory move, copy, clear, etc. auxiliary functions
- Battery power management
- Sensor I/O taking care of interpreting raw data
- Motor control
- LCD and sound output
- IR transceiver I/O

The firmware code, either as loaded from internal storage or over IR, is stored in a memory area starting at hex 8000. The default firmware is meant to interpret user programs expressed in the form of byte code [3]. Since we do not use the byte code in any way, the structure and functionality of the default firmware are beyond the scope of this paper.

3 GNU Java compiler and libgcj runtime

The GNU Compiler Collection (GCC) is a suite of compilers, based on the original GNU C Compiler architecture as created by Richard Stallman and others, including C, Fortran, C++, Objective C, and Java compilers. These compilers share the same basic structure and a common set of back ends, including a back end for the Hitachi H8 series of processors. The C and C++ versions of these compilers had earlier been adapted for the RCX environment by Markus L. Noga and others [7] [8].

Utilizing the C and even C++ compilers of the GCC suite is pretty straightforward, even for an embedded stand-alone environment such as the RCX. Basically, the C compiler requires little runtime support while the C++ runtime support is relatively modest. However, the runtime support required by the GNU Java compiler, GCJ, is both much more complex and currently less mature than its C and C++ counterparts. Furthermore, the Java runtime support is provided as a separate package, called `libgcj`, and its connections with the actual compiler are largely undocumented.

When we started our work with the system, a new version of GCC, GCC 2.95.1, and a corresponding version of the runtime, `libgcj` 2.95.1, were just released. However, in that version the compiler had a number of immaturities and restrictions, some of which have later been relieved while others have not been. In our work, we have attempted to generalize away some of the immature features and restrictions, basically aiming for a compiler that would be more runtime independent.

3.1 Restrictions in GCJ 2.95.2 and -current

In order to facilitate understanding of the current status of the compiler and the way the compiler restrictions made our work a little bit harder, a brief outline of the GCC compiler structure and the definition of the Java programming language is needed. They are presented next.

GNU compiler structure. The compilers in the GCC suite are structured around a three pass architecture. First, a file to be compiled is read in, along with any explicitly or implicitly included files. While parsing, the compiler front end forms recursive data structures, called *trees*, of any global declarations encountered. In the case of Java, all classes and most methods and fields are considered global; a Java compilation unit may only contain classes belonging to a single package, and classes within a package have access to all non-private fields and methods of each other.

Intermixed with the first pass, whenever the parsing of a compilable entity such as a method or a class is finished, the second pass creates an intermediate RTL representation of the entity. RTL, short for *Register Transfer Language*, is a kind of an abstract machine language.

In the third pass, a compiler back end optimizes the RTL representations through a number of passes into assembler language. Target specific transformations are taken into consideration and used by the optimization routines.

While building the Java runtime we experienced a number of times the relative newness of the Java front end, and to a lesser degree the fact that the COFF (Common Object File Format [14]) back end had apparently not been used before with the Java front end. However, the back end problems were basically inabilities to handle the names of Java specific data types, i.e., arrays, as first class objects. More specifically, the back end needed a fix to mangle any symbols containing brackets.

The Java front end. The front end restrictions were more severe but also more subtle. To understand these, we have to dig deeper into the structure of the Java front end.

In contrast to the C programming language, which was designed to be independent from any runtime library, the Java language specification explicitly defines a number of runtime classes that belong to the `java.lang` package. Of these, especially `Object`, `Class`, and `Throwable` are fundamental. Additionally, the default semantics of a number of runtime checks expect certain subclasses of the `Throwable` classes, e.g., `NullPointerException`, which is thrown whenever a null reference is followed. Furthermore, if a native compiler is to provide information needed by the Java reflection API (the `java.lang.refect` package), the compiler needs to supply information about the fields and methods of classes.

Field name	Field type and contents
vtable	pointer to a table of function pointers; initialized to point to the dispatch table of the class
sync_info	void pointer; initialized to null

Table 2: Instance fields silently inserted
to the Object class by the unmodified GCJ

The GCJ compiler addresses these needs by handling a number of runtime classes specially, including Object, Class, Throwable, Error, Exception, and Thread. Of these, the compiler treated the classes Object and Class significantly differently from others, thereby creating unnecessary limitations for the runtime. More specifically, the out-of-the-box GCJ silently inserts a number of fields (see Tables 2, above, and 4, on the next page) into these classes, and does not allow any new fields to be defined in the corresponding Java source code. Furthermore, even if some of the silently added fields can be accessed by Java code to be compiled, classes containing such references cannot be compiled with any other Java compiler. Finally, some of the internally generated fields have types that cannot be represented in Java. The restriction of not allowing any other fields to be declared, along with the other two above mentioned features, made it relatively hard to build the runtime in Java. Lifting the restriction and modifying the compiler, as described next in Section 3.2, alleviated the situation.

3.2 Modifications to the Java compiler

The problems encountered were mostly related to the Java front end and not to the rest of the compiler; this is a strong indication of the very high quality of the GCC compiler suite in general. Unfortunately, apparently due to early design decisions, the GCJ compiler is built around the idea that the necessary Java runtime would be mostly written in some other language than Java, e.g., in C++. Considering the fact that the GCJ compiler is able to produce native code in addition to byte code, we considered that approach limiting. Furthermore, as our goal was to implement as much as possible of the RCX operating system in Java itself, we decided to lift these restrictions and modify the compiler so that the runtime could be written in Java to the greatest extent. These modifications are explained next.

Making the compiler-inserted fields visible. In addition to disallowing any new fields from being added to the fundamental classes, the unmodified GCJ rejects (re)definitions of any fields with a name matching any of the compiler generated fields. That is, when we naively tried to add a field named interfaces to java.lang.Class, the compiler complained about field redefinition. On the other hand, if we tried to use the compiler generated field interfaces without explicitly declaring it in the source, Sun javac refused to compile the file.

Field name	Field type and contents
vtable	No changes
monitorSema	byte; id of the thread holding the monitor corresponding to the object
monitorQueue	byte; id of a thread waiting for entry to the monitor; other threads waiting for entry to the monitor are linked to the first thread
monitorCount	byte; the number of times the holding thread has recursively entered the monitor
waitQueue	byte; id of a thread waiting for a notify on this object, any other waiting threads are linked to the first thread

Table 3: Instance fields inserted
to the Object class by the modified GCJ

To resolve the dilemma, we modified the compiler so that if the user defines a field with a name clashing with a compiler generated field, and if the types of the compiler generated field and the user defined field are assignment compatible in both directions, the compiler only issues an warning, not an error. By using a new compiler flag, even the warning can be silenced.

Since making the compiler generated fields accessible is only meant to be used in implementing the runtime itself, we went still a little bit further and loosened slightly the assignment compatibility rules. One effect of this was that the compiler still could declare some of the integer fields unsigned (which is *not* representable in Java) while the corresponding user defined fields are signed. Additionally, we allowed the type java.lang.Void to function as an unnamed pointer, i.e., something like void pointer in C or C++. Thus, in that way we were able to declare in Java even those compiler generated fields whose types could not be represented as Java types.

In addition to these loosened type compatibility rules, a redefinition in the modified compiler resets the field's visibility to that defined by the user, thereby allowing

Field name	Field type (and contents)
next	a pointer to java.lang.Class
name	a pointer to an UTF8 constant string
accflags	unsigned short, access bits
superclass	a pointer to java.lang.Class
constants	a record enclosing constants info
methods	a pointer to the first element in an array of internal method records
method_count	short, size of the array above
vtable_method_count	short, number of virtual functions
fields	a pointer to the first element in an array of internal field records
size_in_bytes	int, size of instance objects
field_count	short, number of fields
static_field_count	short, number of static fields
vtable	a pointer to the table referred at the Object's vtable
interfaces	a pointer to the first element of an array of class pointers
loader	void pointer, initialized to null
interface_count	short, number of interfaces
state	byte, initialized to zero
thread	void pointer, initialized to null

Table 4: Instance fields silently inserted to the Class class

wider access within the runtime. (By default the compiler generated fields are considered private.)

Changing the types of the compiler-inserted fields. As was explained in Section 3.1, the compiler silently inserts a number of instance fields to the Object and Class classes, among others. However, the types of many of these fields are defined so that they cannot be expressed in Java. Fortunately, modifying the compiler to use a corresponding Java compatible type was relatively easy in most cases. The new types for the changed inserted fields are given in Table 5. (For the original field types, see Table 4, above.)

Field name	New field type
methods	java.lang.reflect.Method[]
method_count	*removed*
fields	java.lang.reflect.Field[]
field_count	*removed*
interfaces	java.lang.Class[]
interface_count	*removed*
loader	java.lang.Loader
thread	java.lang.Thread

Table 5: Modifications to the fields installed to a Class

Changing the types of some of these fields required considerable changes, some of which were not quite obvious. For example, the old methods field was a pointer to a C struct array. Each element in the array described a method, and the field method_count provided the length of the array. In our Java friendly version, the methods field is a pointer to a Java array object. The generic memory layout of a Java array is described in Figure 2, below. In the standard case, the array object would contain pointers to Method objects stored elsewhere in memory. However, since we are here having the compiler generate the array and have full control

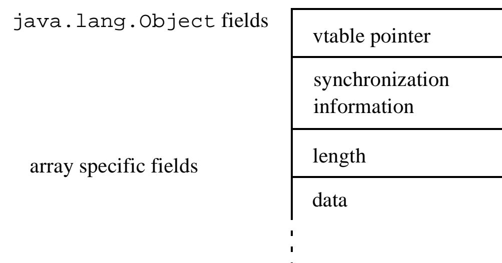

Figure 2: Java array structure

Original function name	The corresponding method after modification	Enclosing class
`_Jv_InitClass`	`void initClass()`	`Class`
`_Jv_Register_class`	`void registerClass()`	`Class`
`_Jv_AllocObject`	`Object allocObject()`	`Class`
`_Jv_CheckCast`	`Object checkCast(Object)`	`Class`
`_Jv_IsInstanceOf`	`boolean isInstace(Object)`[a]	`Class`
`_Jv_LookupInterfaceMethod`	`Method lookupInterfaceMethod(String, String)`	`Class`
`_Jv_MonitorEnter`	`static void monitorEnter(Object)`	`Thread`
`_Jv_MonitorExit`	`static void monitorExit(Object)`	`Thread`
`_Jv_Throw`	`static void throwException(Object)`	`Thread`
`_Jv_ThrowBadArrayIndex`	`static void throwBadArrayIndex(int)`	`Thread`
`_Jv_exception_info`	`static Throwable exceptionInfo()`	`Thread`
`_Jv_NewArray`	`static Object newArray(int, int)`	`Runtime`
`_Jv_NewObjectArray`	`static Object newObjectArray(int, Class, Object)`	`Runtime`
`_Jv_NewMultiArray`	*currently unsupported*	
`_Jv_CheckArrayStore`	`static void checkArrayStore(Object, Object)`	`Runtime`

Table 6: Runtime functions internally called by the compiler

a. Note that `isInstance(Object)` is a method that is called not only by the runtime, but that is also available as a part of the public API; in order for this to work, the order of the parameters to the method was changed.

over its internal structure, a more compact representation is possible. Thus, instead of storing pointers to the `Method` objects in the Java array, we store the `Method` objects themselves. Since the compiler knows the actual type of the array, it generates correct code when accessed from Java. However, care must be taken when declaring and accessing these kinds of arrays from C++.

Redirecting compiler-generated support functions to Java. Any Java runtime includes a number of relatively high level functions, including memory management, threads, thread synchronization, a few of type management operations, and runtime class handling. To implement these, the GCJ compiler inserts calls to runtime support functions in the generated object code (see Table 6, above). In the unmodified compiler, these functions are C functions, and some of them take arguments whose type cannot be described in Java. Now, in order to be able to write as much of the runtime in Java as possible, we modified the compiler so that the inserted functions are calls to Java methods and the arguments are representable as Java types. The resulting methods are also given in Table 6.

Since these methods are directly inserted to the object code without any access checking, we declared the corresponding methods in the Java classes as using default (package) or `private` visibility, thereby making it impossible to directly use them from outside of the runtime classes. (`Class.isInstance` is an example of deviation from this scheme, as it happens to be a `public` method defined in J2ME.) In a few cases where the functionality could not be implemented in Java (yet) the corresponding Java methods were declared `native`, and the implementation was made in C++.

Other front end modifications. The other modifications include the following:

- The semantics of `synchronized native` methods and `static transient volatile` variables were changed, as described in Sections 3.3 and 5.4.
- The generation of metadata for Java methods, fields, and class names was made optional. The method and field information is only needed for the reflection API while class names are also needed for the `Class.forName` method. Since most

embedded programs do not use the reflection API nor the `forName` method, leaving that information out reduces the code size about 30%.
- The automatic inclusion of compiler generated fields in the `Object` and `Class` classes was made more generic, or less hardwired, thereby making it easier to specify runtime specific fields in these classes.
- In a number of locations, the front end created code that made computations using the Java `int` type variables. For example, the array `length` field and any computations applied to it, including array index checking, used `ints` internally. However, since the RCX is a 16 bit processor, and has only 64 kB of total address space, the resulting assembly code is both inefficient and unnecessarily voluminous. To alleviate this problem, we created a new compiler symbol for Java array lengths, and used that in the calculations. Selecting an 16 bit integer for this type produced much better assembler code.
- According to the Java specification, the JVM handles integral data types that have less than 32 bits as 32 bit integers during computation of expressions. However, GCJ attempts to optimize this issue and uses only the native machine word size whenever it is sufficient. Now, when GCJ reads in Java byte code, it cannot immediately determine the "real" types of JVM stack variables just by inspecting the byte code instructions. Handling of this had a couple of bugs that appeared only in machines with a small native word size. The fixes were contributed back to the GCJ project. However, more could be done in this respect, and new optimizations would help produce even better code.

3.3 H8/300 back end optimizations

The backend optimizations we made were mostly related to two issues. First, we changed the calling conventions to produce slightly better code and to be compatible with the RCX ROM calls. Second, we changed the register usage directives so that the resulting code uses fewer memory references than with the default directives, when compiling typical Java code. These changes required us to modify the register usage of the runtime system, including `setjmp` and `longjmp` as well as exception handling.

Changes in register usage. The standard GCC back end for the Hitachi H8/300 microcontroller uses register `r7` as the stack pointer, `r6` as a frame pointer, and registers `r0`...`r2` to pass the first three arguments to functions. The rest of arguments, if any, are passed on the stack. While a standard practice, this has a drawback in the case of Lego RCX, since the RCX ROM calling conventions are different. That is, the first argument to a ROM routine is passed in register `r6` while the rest are passed on the stack. LegOS and librcx have solved this problem by adding a small assembler wrapper which is used when calling the ROM routines. However, in order to gain efficiency and to minimize the amount of code not written in Java, we solved the problem differently.

Basically, GCC calling conventions are defined using relatively simple macros and functions in the target specific back end specification. Thus, in order to support direct ROM calls from Java we created a new back end variation, called `h8300-hitachi-rcx` (instead of default `h8300-hitatchi-hms`), that has a different call convention. First, register `r3` is used as a frame pointer instead of register `r6`. Second, the first parameters to a function are passed in registers `r6`, `r5` and `r4`, *unless* the call is declared as a ROM call, in which case only `r6` is used to pass parameters, and the rest of parameters are passed on the stack. As other register usage related optimizations, we declared that frame pointers may be omitted when not needed and that register `r4` is saved over function calls. Together these reduced both code size and the number of memory stores and loads.

Calling ROM functions directly. The changed register usage made it easy to call ROM routines directly from C by declaring a ROM routine as an external function and declaring the actual address in the linker configuration file. To ensure correct parameter passing, the external function declaration must be announced as a ROM call by using a C or C++ `__attribute__`, or a corresponding preprocessor `#pragma`, which signals the back end to use the alternative calling conventions.

As an example, let us consider the ROM routine "set LCD segment," which is available at `0x1b62`. The corresponding C declaration and a relevant fragment of the linker configuration file are given in Figure 3, on the next page.

Making a ROM routine callable directly from Java required a little bit more. First, Java does not include any feature corresponding to the C/C++ usage of attributes in method or field declarations. However, Java allows a native method to be declared synchronized, but does not associate any specific semantics for this. (Synchronization is implemented by a method itself; the caller of a method does not need such information.) Thus, we hacked the compiler so that it flags any method declared as `native synchronized` as a ROM entry point.

```
extern char set_lcd_segment(
  short code
) __attribute__((ROM));
```

```
SECTIONS {
 ...
 .rom : {
   _set_lcd_segment = 0x1b62;
   ...
 } > rom
 ...
```

Figure 3: A C declaration for an RCX ROM routine, and a corresponding linker directive

Second, the link time naming conventions for Java methods are different from those of C functions. In order to overcome this, the entry points must be declared as *mangled names* in the linker configuration file. The result of converting the C example is shown in Figure 4, below.

4 Minimal Java runtime

Originally, the Java 1.0 and Java 1.1 specifications defined a single language and runtime structure. However, along with the success of Java, Sun Microsystems started to define a number of versions of the runtime, targeted for different purposes. Currently, *Java 2 Micro Edition* (J2ME) [15] is the smallest runtime specification still supporting the full language. On the other hand, the *Java Card Runtime Environment* (JCRE) [16] provides a still smaller runtime, but due to the restrictions it imposes on the language it is arguable whether JCRE utilizes Java at all in the sense that the rest of the Java runtime environments do.

```
class Display {
  static native synchronized void
    setSegment(short code);
```

```
SECTIONS {
 ...
 .rom : {
  _setSegment__Q17Displays = 0x1b62;
  ...
 } > rom
 ...
```

Figure 4: A Java declaration for an RCX ROM routine, and a corresponding linker directive

In our project, the goal is to be as compatible as possible with J2ME. However, the J2ME specification is based on the assumption that the runtime environment is capable of executing Java class files, i.e., has an interpreter, JIT, or hardware support for byte code. Due to space restrictions, this is not possible in the RCX, and therefore we cannot be fully compatible with J2ME.

Below, we describe the basic API provided by our environment, the language features currently not supported, static constructor, destructor and class initialization routines, and thread support. The description of the operating system level features are deferred to Section 5.

4.1 `java.lang` APIs

The `java.lang` classes included in our runtime are enumerated in Table 7. The methods provided are more restricted than they are in the Java standard edition, but mostly aimed to be compatible with J2ME. However, Sun has not produced any public specification of the actual J2ME API. Therefore, we have used Embedded Java [17], which is available, as our comparison point. The most important differences from the Embedded Java API are outlined in the table. Most notably, the `String` class is heavily restricted, providing only minimal support. Furthermore, some exceptions and many errors associated with runtime checks are eliminated due to the static nature of our runtime environment.

When compiled into class files, the total size of the class files is about 32 kilobytes. As a binary library (including symbols), the runtime fits into 170 kilobytes. In binary, with all the symbolic and metadata information eliminated, a typical runtime size is 15 kilobytes.

4.2 Language level restrictions

Our current implementation has a number of restrictions that affect the typical programmer. These include the following.

- The runtime library does not currently support `floats` or `doubles`. Source code attempting to use them compiles but does not link.
- Multidimensional arrays are unsupported. Code attempting to create such arrays calls `newMultiArray`, which has not been implemented.
- Java interface classes are not supported either, since their current implementation in GCJ relies on the method and field metadata, and we want to omit them from the binaries. If the metadata information is included, the interface support should work, but that has not been tested.

Classes	Differences to Embedded Java
`Object`	Method `toString` omitted; otherwise full API.
`Class`	Methods `getResource`, `getResources`, and `getSigners` omitted; otherwise full API.
`ClassLoader`	Only available as a placeholder. All methods omitted.
`System, Runtime`	I/O channel, library, process, property, security manager, and trace releated fields and methods omitted. Some of the other methods are currently placeholders only.
`Throwable, Error, Exception`	No messages or stack trace supported. All errors and exceptions are assumed to be immutable singletons. Each class has a constant static field `instance`.
`Void`	Full API. Also used as an opaque type for data whose type is not available in Java.
`Boolean`, `Byte`, `Integer`, `Number`, `Long`, `Short`	String related parsing and printing omitted. Conversions to floating point numbers currently unsupported. Otherwise full API.
`Double`, `Float`, `Math`	Currently unsupported.
`Character`	Only rudimentary support. Most methods omitted.
`String`	Only rudimentary support. No constructors nor any methods creating new strings are available. All strings must be created at the compile time.
`Thread`	String, security manager, stack trace, thread group, and deamon threading methods omitted. Priorities are currently unsupported, but the API is available.
`Compiler`, `Process`, `StringBuffer`, `Security-Manager`, `ThreadGroup`	Not available at all. Thread groups might be added later; no need for others.

Table 7: Differences between Embedded Java API and the API of our implementation
(Some classes, such as the exception and error classes, are left out for brevity.)

4.3 Static constructors and destructors

When compiling Java code, the GCJ compiler produces stubs for class registration. That is, for each compiled class, the GCJ produces a piece of code that is not called anywhere from the code, but a pointer to it is placed into a so called `.ctors` loader area. The initialization system in the runtime loops over the pointers in the `.ctors` area, calling each function in turn. The same mechanism is used for running static constructors in a C++ program. In the case of GCJ generated code, however, the stub code only calls the `registerClass` routine, allowing us to create a list of all classes.

4.4 Class initialization

By default, the GCJ compiler behaves strictly according to the JVM specification, and generates lots and lots of calls to `initClass`. That is, whenever a Java class is accessed from the Java source code, the code generator generates a piece of code that first calls `initClass`, and then performs the actual class access. For example, consider the following code fragment.

```
class Ex1 { static short x; }
...
  Ex1.x = 0;
...
```

The code generated from the assignment looks like the following.

```
mov.w   r6,#_CL_Q13Ex1
jsr     initClass
sub.w   r0,r0
mov.w   r0,__Q13Ex1$x
```

This approach complicates slightly the writing of the code for the `initClass` method. That is, care must be taken that all classes that are accessed from `initClass`, either directly or directly, must be separately initialized by explicitly marking them initialized and calling their class initializers *before* any other Java code is called. Furthermore, the class initializers of these classes must not cause invoking of the `initClass` method for any other classes. If these rules are not strictly followed, a call to `initClass` easily results in an unterminated recursion or other disastrous complications.

4.5 Memory management

As already mentioned, most of the runtime environment and operating system was written in Java. However, explicit C, C++ and even assembler support was needed for the memory management, exception handling, thread management, and thread synchronization. Of these, exceptions, thread management and thread synchronization are discussed in more detail later in Section 5, while memory management is considered next.

In Java, the memory is managed at the granularity of objects. Each object consists of a few compiler generated fields (see Table 3) along with any user defined instance variables. Each instance variable contains either a reference to some object, or a primary data item. For the variable fields, GCJ uses natural field sizes, and aligns the fields according to the C++ alignment rules (this is different from what most JVMs do). In order to allocate objects, the compiler arranges calls to allocation routines `allocObject` and various versions of `newArray`.

Due to the limited amount of memory available and the nature of programs running in a typical robotics application, we assume that most applications will create a small number of relatively long living objects. Furthermore, temporary circular data structures will be more likely the exception than the rule.

Our current garbage collection method is very simple, and similar to the Java Card Runtime Environment. That is, garbage is not collected, and allocated objects stay around until the next firmware or hardware reset. However, to better support dynamic data structures we are planning to support garbage collection. The current options include a simple reference counting scheme that would be implementable in the compiler back end, and a simple mark-and-sweep collector utilizing object layout information provided by the compiler. However, both of these schemes require considerable support from the compiler, and are presently left for further study.

5 Operating system services

Due to the simple nature of the RCX hardware, the border between the Java runtime environment and the operating system is ambiguous. However, while object level memory management along with garbage collection is more or less hardware independent, exceptions, threads, low level hardware access, and events are more tightly bound to the underlying hardware than to the language. Therefore, we decided to classify the latter issues as operating system services, and consider them next.

5.1 Exceptions

The GCC collection of compilers have two different possibilities for exception handling. The default method uses the `setjmp` and `longjmp` functions while the alternative method explicitly scans the execution stack looking for exception handling frames. Since the default method produces more compact code, especially when we told the compiler to use our own versions of `setjmp` and `longjmp` instead of the compiler internal versions, we decided to use the default method.

In Java, there are two basic constructs that cause the compiler to construct exception handling frames. The first one is obvious, namely the Java `try ... catch` construct. The other case are the `synchronized` methods and statements, which use exception handling code to release monitor locks.

Now, whenever a GCC compiled function establishes an exception handling frame, it first calls an internal function that returns an *exception context*. In our implementation, the context is a part of the currently running thread. Next, when entering the protected block of code, the compiler allocates a few bytes on the stack, calls `setjmp` to store the current execution context there, and links this stack frame to the front of a list of exception handling frames, available in the exception context.

If the execution of the block terminates normally, the stack frame is popped from the list and the stack is restored. On the other hand, whenever an exception is thrown, the exception context is consulted to get the list of exception handling frames, and each frame is called in order until the exception is caught or the list terminates. In the latter case, the execution of the thread is terminated.

5.2 Threads and synchronization

In the RCX, a thread of control is very simple, essentially consisting of the current processor state. A context switch merely changes the contents of the processor registers. The state is stored in a `Thread` instance, and is visible to the Java environment as a series of `volatile short` instance variables. This allows direct manipulation of non-running threads from Java code.

To save memory and simplify implementation, the number of threads is limited to 126. This allows a thread identifier to be stored in a single signed byte (thread number zero is used as a sentinel). Each thread also contains a link byte, possibly containing a thread ID of another thread. These link bytes are used to create lists of threads waiting for a specific event.

Monitors. Object level synchronization, implemented in `Thread.monitorEnter` and `Thread.monitorExit`, is implemented with a simple per object semaphore together with a counter and a queue. The `monitorEnter` and `monitorExit` actions disable interrupts by manipulating the processor condition code register through the `Thread.disableInterrupts` and `Thread.enableInterrupts` assembly routines.

The monitor semaphore is represented as a single `byte`, present in all objects (see Table 3). Whenever there are no threads within the object's monitor, the semaphore byte is zero. When the first thread enters the monitor, it places its thread ID to the semaphore byte. Any other threads attempting to enter the monitor will find the monitor occupied, and insert themselves to the head of the monitor queue.

When a thread leaves the monitor, the thread checks if there are any other threads waiting for the monitor. If such a thread is found, the semaphore is kept busy and the first waiter is removed from the queue and its ID is placed into the semaphore byte, after which it is woken up by linking it to the run queue. On the other hand, if there are no waiters, the semaphore is simply released.

Since a Java thread can recursively enter a monitor several times, yet another variable is needed to keep count of these recursive entries. Due to the limited stack in the RCX, we decided to use a single byte as this counter as well.

Condition variables. Another type of synchronization is provided by the Java `wait` and `notify` primitives. In Java, any thread holding an object monitor may invoke the `Object.wait` method. This suspends the thread issuing the call until either another thread invokes the `Object.notify` method for the same object, or a time out occurs.

Again, our implementation is fairly simple. In addition to the monitor thread queue, each object also includes a waiter queue. This queue is also implemented as a simple byte, which is initialized to zero. Whenever a thread enters a `wait`, it stores the current value in the wait queue to its thread ID link, and places its own thread ID in the wait queue byte; this, effectively, places the thread at the head of the waiter list.

When another thread invokes `notifyAll`, thereby waking all threads waiting on a thread's wait queue, the thread list is simply moved to the monitor queue. The `notify` method, instead, just moves the head of the wait queue to the monitor queue. (In this case the wait queue acts as a stack, but that is perfectly fine according to the Java language specification.) The notified thread or threads are woken when the notifying thread leaves the monitor, as explained before.

Wait time-outs are currently not supported; their addition may require slight revisions to the implementation.

5.3 ROM services

The ROM services are encapsulated into a number of platform specific classes, enumerated in Table 8. IR communication is not supported, yet. The platform specific classes form a package of their own, called `com.rcx`.

Class	Description
`Button`	Initialize, read and shutdown RCX buttons.
`Display`	Modify the LCD screen contents.
`Motor`	Power motors in a controllable way.
`Power`	Allows access to power savings.
`Port`	Direct access to the hardware I/O registers.
`Sensor`	Power, read, and shutdown sensors.
`Sound`	Allows sounds to be played.
`Timer`	Access to the hardware timers.
`Vector`	Direct access to the interrupt vectors.

Table 8: Platform specific classes in the `com.rcx` package.

Most of the platform specific classes contain a number of native synchronized methods that are used to directly call the ROM routines. In most cases, these methods are `public`, allowing direct access from anywhere in the program. Only those ROM calls that require specific arguments or otherwise cannot be called without possible problems are protected by restricted visibility.

5.4 Hardware access

In some cases it is clearly beneficial to bypass the ROM and to access the hardware directly. To support this, we modified the GCJ compiler so that any variable defined

```
class Port {
  ...
  static transient volatile PORT4;
```

```
SECTIONS {
 ...
 .eight (0xFF00) : {
  _Q14Port$PORT4 = 0xB7;
  ...
 } > eight
 ...
```

Figure 5: Definition of the variable `Port.PORT4`, which allows direct access to the hardware I/O register number 4.

as `static transient volatile` is considered as if it were a declaration of an external variable instead of being a definition. Thus, the compiler does not allocate any memory for those kinds of variables. Therefore, their location in the memory can be freely decided by the linker. Again, utilizing linker directives made this easy. Figure 5 illustrates the definition of I/O Port 4, whose memory address 0xFFB7. Among other things, port 4 can be used to directly read the status of two of the user buttons.

5.5 Event model

In Java, it is customary to represent changes in external environment as events. Thus, our intention is to abstract changes in button and sensor values as events. The basic idea is to have a thread polling on a specific I/O port, waking up on interrupts generated either by a change in the port or by a timer. If the polling thread notices that the value represented at the port has changed, it takes an event object from a queue of free event objects, fills in the appropriate values, and places the object in an appropriate event queue. A non-interrupt level thread would be waiting on the queue, and handle the event after the interrupt routine has been completed. Finally, the event would be passed back to the free event queue when it is not needed any more.

5.6 Other services

We expect to enhance our operating system with a number of additional services. The planned services include communications (both basic IR and IP over IR) and scheduled power management (to save power in long running sensor-type applications). However, these features are currently left for further study.

6 Evaluation and lessons learned

It was no surprise to us that it was both challenging and fun to write a new operating system (or an operating systlet) in a new language for a hardware we were not familiar with when we started. However, the main obstacles came from a direction we could not anticipate. That is, the intrinsic interrelationships between a compiler and the corresponding runtime system are much more complex and fragile than we originally expected. Writing a runtime system independent compiler for C is clearly feasible, as shown by GCC. The same applies, more or less, to the GCC C++ compiler. However, the current GCJ compiler is far from being runtime neutral, and currently one is required to have good knowledge of the compiler internals in order to be able to write a new type of runtime system. We learned this the hard way, and hope that this paper and our modifications to the GCJ allow others to do the same more easily.

Looking at the situation from another direction, the fact that we were able to complete the project in the first place is an indication of the feasibility of our approach. First, we have shown that the idea of using Java as a low level language to implement both a Java runtime environment and a minimal operating system in Java is viable. The basic methods, or tricks, that we used in our implementation include the following:

- Using a compiler that produces native code instead of byte code.
- Wiring the compiler-generated Java runtime primitives (see Table 6) back to Java, thereby allowing the primitives to be implemented in Java instead of some other language.
- Enabling Java source level access to the meta-information generated by the compiler. This allows, for example, an easy pure Java implementation of `Class.isAssignableFrom`.
- Converting `static transient volatile` and `synchronized native` modifiers into external declarations and back end specific attributes, respectively, which make it possible to tailor the compiler and linker to provide direct access to the underlying ROM routines and hardware addresses.

Second, our experience indicates that writing an operating system in a beautiful object oriented language, such as Java, gives a number of benefits. In the present case, the target environment is such a simple device that a strict boundary between the operating system and an application would probably only complicate things and make the application both bigger and less efficient. Object-orientation allows some of the underlying problems to be solved in a neat way. That is, visibility rules, data

hiding, and inheritance make it possible to provide an application programmer an environment where the application may be tightly integrated with the operating system without compromising architectural layering or introducing unnecessary bugs. We allege that the same principles could also be applied in a more complex case if appropriate memory management hardware was added.

Considering the language, the main benefits of Java lie in its relative strictness when compared with C++. That is, given any C++ based operating system level framework, the programmer is required to understand considerable amount of the implementation details in order not to mistakenly break the underlying semantic assumptions of the framework. In the case of Java, the semantics-related problems are easier due to the more strictly defined language specification and fewer possibilities for a programmer to circumvent language-level object abstractions.

The use of Java brings up another benefit. That is, since the APIs are to a large extent compatible with the standard Java APIs, it should be possible to port a large number of Java packages to the RCX with no or minimal changes. Now, for example, porting a minimal JACL [18] interpreter to the RCX should not be too hard.

Hence, our work has shown that Java can be used as a viable language for low level programming, with benefits unavailable from other approaches.

7 Future work

At the present time (April 2000), garbage collection, threads and the event model require more work. Some of the modifications made to the GCJ compiler could be made more generic and supplied back to the standard version of GCJ. An extension to study is the ability to handle dynamically loaded code based on the work recently introduced in LegOS. However, due to the Java visibility constraints this may not be easily adoptable.

Once the basic operating system platform has stabilized, we plan to focus on communication issues. The aim is to port our Java Conduits Beans (JaCoB) protocol framework [19] to the RCX, and to build a minimal IPv6/UDP implementation on the top of that. Our hope is to see if it would be possible to make the RCX robots first class citizens in Jini communities.

Availability

The source code for the system is available at `http://www.tcm.hut.fi/~pnr/rcx/`. The actual source tree is supplied as a gzipped tar file, whose size is about 250 kilobytes. Building the system requires patched versions of both GNU binutils and GCC; the necessary patches are provided. The binutils patch is minimal (less than 2 kilobytes) and should not cause any problems. However, since the various versions of the GCC patch are fairly large (about 150 kilobytes) and since they were made against GCC-current instead of any specific released version, building a working compiler may require some manual work, or, alternatively, using CVS to check out GCC-current of the date when the particular version of the GCC patch was created.

Acknowledgments

This work would have not been possible without the large number of people working on the RCX reverse-engineering and the various programming environments, including, in no particular order, Kekoa Proudfoot, Markus L. Noga, David Baum, Peter Liu, Stephen Spackman, Michael Daumling, Ross Paterson, Frank Cremer, Sergey Ivanyuk, Mark Falco, Mario Ferrari, Frank Mueller, Tom Emerso, Lou Sortman, Luis Villa, David Van Wagner, Michael Nielsen, Chris Dearman, Eric Habnerfeller, and Ben Laurie.

Since the availability of the compiler source code was essential for this work, we want also to thank Richard Stallman, the Free Software Foundation, Cygnus Solutions (now part of Red Hat), the GCJ implementation team including Alexandre Petit-Bianco, Per Bothner, Andrew Haley, Tom Tromey, Anthony Green, Warren Levy, Bryce McKinlay, and others, and the large number of volunteers for their work in providing free software in general, and the GNU Compiler Collection in particular.

We are also grateful to Tuomas Aura, Hannu Napari and Lauri Savioja of HUT and Chris Demetriou and the anonymous reviewers of USENIX for their comments and suggestions how to improve the paper, to our students Markus Aholainen and Veera Lehtonen for their feedback about some early versions of the system, and especially to Petri Aukia of Bell Labs for his numerous constructive suggestions concerning this work in its early stages.

References

[1] *Lego Mindstorms*, http://www.legomindstorms.com/

[2] *The MIT Programmable Brick*, http://el.www.media.mit.edu/projects/programmable-brick/

[3] Kekoa Proudfoot, *RCX Opcode Reference*, http://graphics.stanford.edu/~kekoa/rcx/opcodes.html

[4] Lego RCX Code, in *Robotics Invention System User Guide*, The Lego Group, 1998.

[5] *Lego Dacta RoboLab*, http://www.lego.com/dacta/robolab/default.htm

[6] David Baum, *Not Quite C (NQC)*, http://www.enteract.com/~dbaum/nqc/index.html

[7] Markus L. Noga, *LegOS Home Page*, http://www.noga.de/legOS/

[8] Kekoa Proudfoot, *Librcx*, http://graphics.stanford.edu/~kekoa/rcx/tools.html#Librcx

[9] *GNU Compiler Collection (GCC) home page*, http://gcc.gnu.org/

[10] Michael Durrant and D. Jeff Dionne*, uCsimm Home Page*, http://www.uclinux.com/uC68EZ328/index.html

[11] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath, *The Jini™ Specification,* Addison-Wesley, Reading, MA, July 1999.

[12] Kekoa Proudfoot, *RCX Internals*, http://graphics.stanford.edu/~kekoa/rcx/index.html

[13] *Hitachi Single-Chip Microcomputer H8/3297 Series*, http://semiconductor.hitachi.com/products/pdf/h33th014d2.pdf

[14] Gintaras R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates Nutshell Series, Sebastopol, CA, 1988.

[15] *Java™ 2 Platform, Micro Edition (J2ME)*, http://java.sun.com/j2me/

[16] *Java Card™ Technology*, http://java.sun.com/products/javacard/

[17] *EmbeddedJava™ Technology, Source Edition*, http://www.sun.com/software/embeddedjava/

[18] Ray Johnson, *Tcl and Java Integration,* Sun Microsystems Laboratories, Palo Alto, CA, Feb 1998.

[19] Pekka Nikander and Juha Pärssinen, "A Java Beans Framework for Cryptographic Protocols," in Mohammed Fayad, Douglas Schmidt and Ralph Johnson (Editors), *Object Oriented Frameworks, Volume II*, Wiley, 1999.