Embedded Programming with C++

Stephen Williams
Picture Elements, Inc.

Embedded Programming with C++

Stephen Williams, *Picture Elements, Inc.* steve@picturel.com

May 5, 1997

Abstract

This paper presents uCR, a C++ runtime package for embedded program development. We make the case that in certain situations embedded programming is best done without the aid of a conventional operating system. A programming environment in the form of a C++ runtime is presented, and the environment, including the C++ language, is evaluated for appropriateness. Important factors are code size, performance, simplicity and applicability to a wide range of embedded targets.

1 The Problem

It is common, when building a newly designed board, to install only a few components at a time and test the partially built board to protect expensive components, to validate portions of a design, or just to contain the hardware debugging problems. The first time power is applied to a board, often only the CPU, memory and ROM socket are installed. Naturally, software is usually required and a development environment that works in this case is necessary, especially as the board design and construction progresses.

Even complex designs can have real estate constraints, leaving no room for the extra hardware to support a full operating system. A case example of this is shown in Figure 1. In order to fit this design on a PCI card, extra parts like UARTS had to be left out, and program memory had to be kept to one flash and 2 DRAM chips.

Conventional operating systems usually serve two interesting roles: they abstract the target hardware, and they provide a means of loading and executing programs, often in separate protection domains. An operating system provides an operating environment, including but not limited to a device driver interface and a common interaction with the user. It is separated from applications by a kernel structure, bounded by trap handlers or some form of call gate that allows the operating system to function to some degree independent of and protected from the applications that it carries.

Several commercial embedded operating systems are available that run on the relatively conventional CPU in Figure 1, but most commercial operating systems, available in binary form, require board support packages written to provide the necessary support for the O/S, including a console, time ticks, and memory setup.

The ISE board (Figure 1) in particular has no serial port, so program loading must be done either by programming the socketed FLASH memory with a prom programmer, or writing into the board support package a console driver that uses the PCI bus to communicate as a console. The MON960 monitor [8] supports the latter, and the Cyclone-911 board [4] in particular can be used this way, given the appropriate host software.[1]

Although it is sometimes nice to have an operating system that is portable, and essential that certain libraries be portable, it is rare that an embedded program is, or should be, portable. The whole point of a program is to manipulate the specific toaster. There is no value being able to run the toaster program on the VCR. It therefore is rarely useful to have a device-driver interface in an embedded kernel—such can actually make things harder.

We questioned the prudence of forcing a kernelized operating system onto a board with only a few LEDS and an oscilliscope for debug output, and a ROM socket for input. We anticipated this happening often, as designing and building boards is our business. We also noted that the device driver interface of a kernel is pointless, and our targets typically run a single trusted program from reset to power off. We eventually concluded that we didn't really need an operating system at all.

This, then, became the chosen path. We wrote a minimal runtime to support C and C++ that works on the sorts of target boards expected, and we provided that support for a specific compiler, the GNU

[1] Picture Elements supplies with the ISE board a bootstrap loader that loads COFF files from the PCI bus. The loader is written using uCR and the techniques described in this paper.

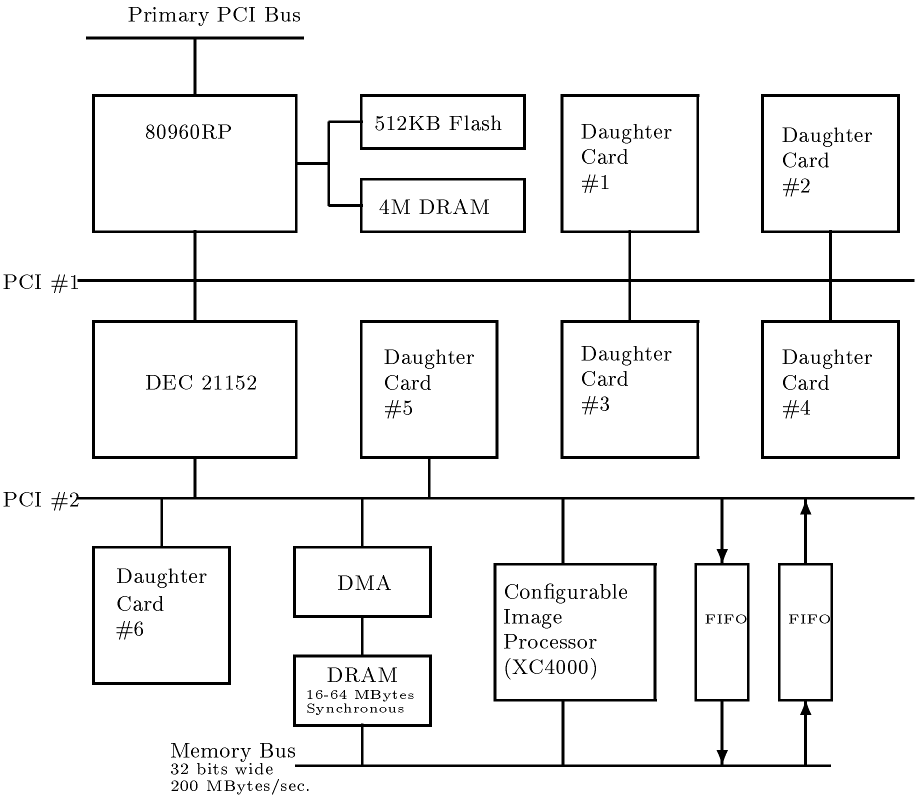

Figure 1: Imaging Subsystem Engine (ISE) Block Diagram [9]

GCC compiler. Writing the support for the compiler alone, we reasoned, would be easier then writing a board support package for compiler *and* operating system and would get everything needed without the added constraints of an operating system.

This runtime support for the compiler, called uCR[2], proved lightweight and powerful enough that we not only used it as the regular development environment, we used it to build bootstrap loaders and other programs in support of embedded development itself.

2 uCR, C and C++

The difficulty of porting embedded operating systems is more often dealing with the board, and not the CPU proper. It is the memory layout and I/O devices that make portable operating systems difficult. If one can reduce the development environment to something independent of the board, then there is only the CPU to worry about and not all the devices around it. uCR/i960 can run on any i960 without porting.[3]

Eliminate the devices from the environment, and eliminate the system call gate, and what remains is a runtime that connects the compiler to the CPU. In the embedded world, anything is possible and imposing irrelevent requirements like a console and a clock ticker can make things harder.

The problem, then, simply reduces to how one maps C++ to a CPU, with nothing else. Although board specific code still needs to be written, specifically the reset handler and application code, uCR makes no requirements other then those needed to support the compiler. This is a task to remind one about the nature of a programming language.

This is not quite the same as more conventional development environments requiring board support packages. Although uCR in practice needs board-specific startup code, it does not impose a style of interaction with the board and programmer. Typical systems require of the board support package console drivers, timer drivers (with the timer configured to tick at a specific rate) and package drivers to initialize the options you choose to include. uCR imposes no such requirements.

[2]uCR is an abbreviation of "Micro-C/C++ Runtime," pronounced U-C-R

[3]In practice, the uCR package, though not the core library, includes code specific to select boards in order to get the developer started writing more complex programs.

2.1 What uCR is

uCR is more properly called a C/C++ runtime than an operating system. A programmer writes an application program in C++ and compiles with the uCR headers. The compiler generates assembly code from the source files that the programmer writes, and what the compiler cannot do it delegates to the execution environment. uCR provides the execution environment for the generated code.

The programmer links the resulting object code with uCR libraries that fill in the parts left out by the compiler, and gets an executable image. This image is loaded into the target by prom programmer, ROM emulator, serial download—whatever works for you—and is executed.

uCR libraries add support for thread programming and interrupt handlers. These are features dependent on the CPU and not the board, so including them does not introduce board support problems. The uCR distribution also includes ancillary libraries that contain device classes, and other code that may not be specific to the CPU but is commonly used.

Interrupt handler support is also included with the uCR core library, because again it is a matter for the CPU and compiler how interrupt service routines are entered and left, and not specific to target boards. What the target boards do fix is the assignment of devices to specific interrupts, so uCR makes no attempt to guess such things.

2.2 What uCR is not

uCR is not a kernel, or a micro-kernel. There are no system calls and there are no task structures such as page translation tables or system call gates. Calls to uCR operations are ordinary function (or method) calls.

uCR also is not intended to abstract the board away. Operating systems that try to abstract the hardware away wind up instead requiring that the hardware be a certain way. That is frequently counter-productive. The uCR core does nothing to devices other than the CPU, and does nothing to get in between devices and the programmer.

2.3 Requirements of the Languages

Both C and C++ place some minimum requirements on the execution environment, but many of the constraints are imposed by the compiler, not the language. If some language construct can be handled easily by the CPU, then the compiler typically just generates the assembly code to deal with it. That is what compilers are for. However, when something is too hard, it gives up and generates a call to external code.

The C language is relatively easy to compile and the compiler only generates call instructions for calls to external functions. Floating point emulation is often placed in a library as well, if the target can reasonably be expected not to have a floating point unit. The C++ language is a bit more interesting. It has difficult constructs that compilers often give up on, like dynamic memory allocation.

The C language standard has a substandard, the *freestanding* C standard [1], to guide the implementer on what can be left out of the development environement and still be worthy of the name "C". In a nutshell, libraries are optional in a freestanding environment. It is rare for an environment to not include some of the more important optional parts, though.

The C++ Working Paper has a similar substandard.[4] [3] The standard libraries are not required of a freestanding C++ environemnt. Only a few support libraries, some specific C library routines, and support for the "`new`" and "`delete`" operators are expected. Things like streamio are certainly not required of a freestanding C++ execution environment, although a specific implementation may choose to provide it.

Static initializers must obviously be done correctly. To fail to initialize static objects is a clear and gross error, but the compiler certainly does not know how to arrange that on my toaster CPU. That, like the minimum library support, becomes a matter for the runtime, namely uCR.

The g++ compiler generates external calls for new and delete. This way, memory allocation can easily be provided with some help at link time. Most targets have memory available for allocation, and some have several different kinds of memory for allocation. Even if the default allocation operators do not apply, the placement "`new`" operator has some interesting advantages.

2.4 Requirements of Common Sense

Ultimately, it is not enough to just make the compiler happy. The programmer using the compiler is the real customer and the programmer wants to make devices do interesting things. It is therefore not useful to have a beautiful and fast string manipulation library if the programmer cannot fit the program in memory.

[4] Strictly speaking, C++ doesn't yet have any standard at all.

Therefore, any practical system should make as much of the standard libraries as reasonable available to the programmer, without imposing extra costs. One programmer may not wish to pay for stdio, but another may find it worth while.

Finally, we wanted a lot of power out of uCR, but simplicity and efficiency were most important. It is intended to be a language runtime, so certain standards, such as POSIX [5], were not considered desireable for embedded applications. We also tried to keep the size and complexity of the programming interface small and understandable. When testing a new board design, or debugging an older malfunctioning board, simple and obvious software behavior has its own special value.

3 Object Oriented Design

By using C++, Picture Elements gained a chance to use object oriented techniques in an embedded context, with threads of execution and interrupts. The obvious potential object classes are:

- Threads,
- Synchronization variables,
- Devices,
- The Debugger,
- Various containers.

The various containers include ring buffers, lists, strings, and the other sorts of things one expects of object oriented designs, are not unique to embedded programming so will not be discussed here.[3, 10, 12]

Incidentally, the requirement of keeping uCR simple and to the point precluded creation of a large and complex system. Instead of a rich texture of objects and classes, we finished with a simple and elegant design, with a few general but very useful classes. A welcome benefit of this is that programmers can learn and be productive with uCR relatively quickly.

3.1 Threads as Objects

The thread programming interface for uCR was revised several times before the current interface was settled on. At first, we designed threads as objects with interesting methods and put them in ThreadQueue containers. Eventually, however, we chose to attach most of the thread methods to the `ThreadQueue` class and left the `THREAD` a passive, opaque object. The run queue, we reasoned, would be a ThreadQueue that the programmer can manipulate like any other ThreadQueue. Threads in the run queue would be subject to execution, and could be suspended simply by pulling the thread from the run queue and placing it elsewhere.

```
class ThreadQueue {
    public:
        // Pass true to the flag to
        // put the thread in front.
      void enqueue(THREAD*, bool=false);
      THREAD*pull();
      THREAD*peek();
};
```

This proved successful, though occasionally cumbersome and less clear then the more conventional POSIX-style thread functions. uCR internally uses the ThreadQueue class for the run queue and suspension lists in synchronization primitives. These instances are generally hidden from the application programmers, who have ultimately chosen to use POSIX-style thread functions provided in the uCR library. Programmers may use the ThreadQueue class to implement new synchronization primitives, if desired.

The threads themselves are opaque objects of type THREAD, and are passed around to the ThreadQueue objects and thread manipulation functions like a thread identifier in a more conventional thread package. The THREAD object is a completely opaque token used by the programmer, and uCR, to represent the object that is a thread. This is more an abstract data type design then an object oriented design. It is a semantic quirk that this C abstract data type is much like a concrete object class in C++.

The idea of a thread as an abstract class with a virtual method for its behavior is known to us. Some call this paradigm an *active object*. [2] Active objects are different from passive objects in that they have their own thread of execution and activate passive objects by calling methods. Threads and interrupt handlers are two different kinds of active objects, synchronization primitives and devices examples of passive objects.

We experimented with active objects, but ultimately decided to implement threads using the THREAD token and a set of conventional thread manipulation functions. The traditional thread functions are well established, easy to use, small and generally don't come into play once the threads are created and started. However, the specific interface built-in to uCR allows for efficient implementation of active objects if desired.

3.2 Memory Heaps

Support for memory allocation in uCR is itself written in C++. However, special care must be taken that all the data structures for managing the default heap in particular be in place before any static initializers are called. To arrange for this, the default heap initializer is called separately and ahead of initializers by the startup function of uCR.

The HEAP_SPACE object is an object token like the THREAD type, and represents a segment of heap space from which memory may be allocated. The uCR startup creates an initial HEAP_SPACE object that is used by malloc and non-placement new operators.

Embedded systems often have different kinds of memory, for example SRAM for small private objects, or perhaps synchronous DRAM for manipulation of large images, and so uCR allows programmers to create other heaps in specified sections of address space.

The uCR support for memory allocation adds a placement new operator that takes as a parameter the HEAP_SPACE to use. Because of the way the heap data structures are designed, deallocation does not require a reference to the HEAP_SPACE object that created it. This feature was specifically included to support the delete operator. Any memory allocated with "new" or "new(HEAP_SPACE*)" can be deleted with "delete". This is necessary because "delete" cannot be overloaded to take a HEAP_SPACE parameter, and to require such would render "delete" unuseable for memory allocated from alternate heaps.

3.3 Synchronization Objects

These classes were obvious and successful. uCR includes several synchronization classes, most derived from the base class ISync. The ISync class is a concrete class that allows threads to wait for a general condition to become true, and allows other threads to notify of a possible change in state. This base class is the most primitive and general synchronization that allows threads to interact with other threads and interrupt service routines.[5]

```
typedef bool (*sfun)(volatile void*);
class ISync {
    public:
      void sleep(sfun fun,
                 volatile void*);
      void wakeup();
};
```

[5]ISRs may not block so may not wait for a condition, but can and usually do report a potential change in state.

The ISema class is a counting semaphore implemented with the ISync class. The only operations supported are increment and decrement (and initialize with a specific value). The methods are easily implemented with the sleep and wakeup operations of the ISync class.

The ILock class is a binary semaphore. In principle, it could be implemented with the ISema class, but it works out more efficiently derived directly from ISync. Operations for ILock are get and put, and do the obvious things.

ISema, ILock and other classes are implemented by deriving from the ISync class. They all have similar fundamental behavior that can be easily factored into the base ISync class. The Mutex class implements *monitor* style synchronization and does not fit well in the ISync class hierarchy, so it is implemented separately.

The Mutex class has "enter()" and "leave()" methods to enter critical sections of code. Only one thread may be active in a critical section, hence the synchronization. The Condition class is a way for a thread to sleep within a critical section. A thread sleeping on a condition is not considered active in the critical section so other threads can enter. However, the implementations of Mutex and Condition assure that at all times there is no more then one thread executing in the critical section.

3.4 Devices as Objects

uCR proper does not operate devices, or expect any to be present, but ancillary libraries include classes that drive various devices. The application may add more device classes simply by writing the code needed to manipulate device registers, and putting that code into classes.

Devices make good objects. Programmers seem to understand and respond well to this technique. As an example, a uCR library includes classes for communicating with a host through a PCI bus. The class "BUS" has a subtype BUS::Device that is the abstract type of a bus interface device. The PLX9060 class is derived from BUS::Device and drives the PLX Technology PCI9060 [11] interface chip. The I960RP class is also derived from BUS::Device and drives the ATU function unit of the Intel i960rp [6] microprocessor.

The BUS class in Figure 2 uses BUS::Device objects to implement a channel protocol with the host processor. The abstract BUS::Device class has a minimum set of methods used by the BUS class for sending packets to the host and getting packets back.

The classes derived from BUS::Device implement the minimum methods (which are pure virtual) and others that the device can support in addition to the minimum requirements. This is a fairly classic object oriented design. A similar hierarchy exists for timers (in case you choose to use them) where the abstract Ticker class provides a common interface for a generic clock and the derived classes implement the virtual methods as necessary for the specific hardware available.

The Ticker class hierarchy in Figure 3 is another example of an object oriented device driver design, also taken from the uCR libraries. The Ticker base class provides the methods of a generic interval timer, that portable code may use. The concrete class derived from the Ticker drives the real hardware timer to implement the behavior of the abstract class.

Even when inheritance does not make sense, device driver code fits well into classes. For example, the XC4000[6] class has the method device_configure() for programming the device, but does not specifically support or require any derivation.

Device classes can be templates, too. The uCR libraries include a template class LED that is a driver for light-emitting diodes. This template is also useful for controlling general purpose output bits, and other miscellaneous jobs.

```
template <class RT> class LED {
    public:
      explicit LED(volatile RT*base);
      void set(unsigned idx);
      void clr(unsigned idx);
      [...]
};
```

Often, LEDS are connected to a register somewhere in the address space of the processor. The register may be a word, or a byte, or whatever the hardware costs and design allow. The programmer uses as the template parameter the integer type needed to access the word (for example "char" or "unsigned long") and passes to the constructor the address of the register. Individual bits of the register can then be set or cleared with the "set()" and "clr()" methods.

The nice feature of this template is that the programmer can use custom types for RT that for example store its value in memory outside the address space, such as I/O space.

[6]A Xilinx Field Programmable Gate Array

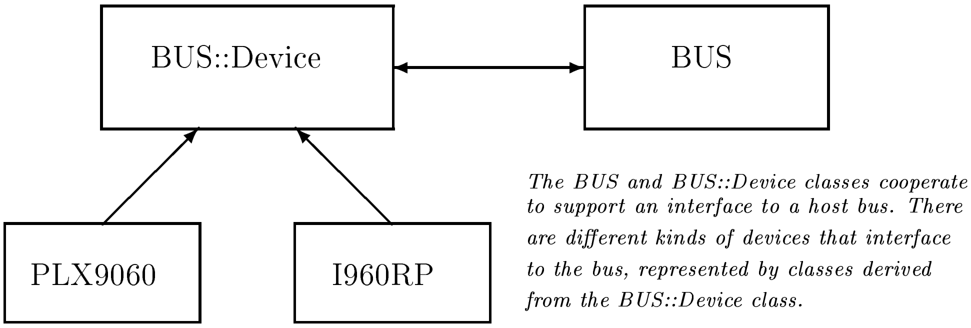

Figure 2: BUS Class hierarchy

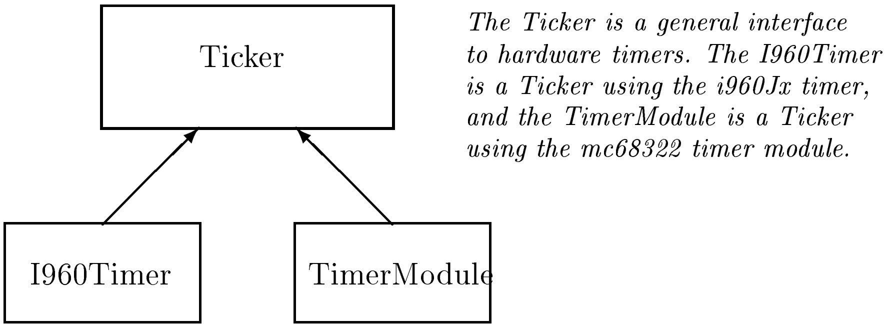

Figure 3: Ticker Class hierarchy

3.5 The Debugger

Some targets have sufficient I/O capabilities to support an interface to a debugger, so uCR provides the `GDB` class. A `GDB` object is an interface to a suitably modified version of the GNU Debugger GDB that runs on a host computer.

The `GDB` class is a concrete class that takes in its constructor a pointer to a suitable device class for use as the communication channel with the host. Once created, the debugger does not become available to the host until its "`go()`" method is called, generally by a special thread. This method never returns and forever communicates with the host, receiving and processing requests for action, memory, etc.

The uCR core leaves things like breakpoint traps and faults available to the programmer, so the `GDB` class uses standard interfaces to gain access to threads and the faults they make. The debugger runs in a thread of its own, so can be said to be an active object. As an active object, it is free to manipulate other threads, stop them, run them, examine memory, etc.

It turns out that not much of what the proxy needs to do is CPU specific, and other then actual I/O that communicates with the hosts, none of it involves board details. Many of the details of the target CPU are managed by the host GBD program, leaving only some CPU specific cache management and fault handling for the `GDB` class to cope with.

3.6 Summary

Table 1 summarizes the core set of types, plus a few others. Notice that this list is very short indeed. The uCR core really exists to provide a runtime context for the C++ program, and not to provide a lot of features. However, the thread and heap support tends to be compiler and CPU specific so must be provided in the core.

Device drivers are not required by the compiler, or by uCR, but some types of devices are common enough to offer in a packaged library some classes to help the programmer. The `BUS`, `Ticker` and `LED` classes are examples of class library support for devices. The list of device types here is not exhaustive. By putting device support in a library instead of a kernel, the drivers can be brought in by the linker automatically if the device is used by a program.

Type	Description
THREAD	Opaque object representing a single thread
ThreadQueue	Container for THREAD objects
ISync	Interrupt safe synchronization variable
ILock	binary semaphore
ISema	counting semaphore
Mutex	MONITOR style thread synchronization
Condition	Condition variables used with Mutex objects
HEAP_SPACE	Opaque object representing a memory heap
BUS	BUS interface abstract class
Ticker	Timer device abstract class
LED	L-E-D and output bit driver template
GDB	Proxy for the GNU Debugger

Table 1: Common uCR Object Types

4 Performance

All the cleverness in the world is useless if the resulting design performs poorly. This in fact is where we met the most resistence from C programmers. The premise of most criticism is that C++ code leads to inferior executables, either bloated by support for C++ capabilities, or somehow merely unoptimizable.

Much effort went into proving by example that tight and efficient C++ code is certainly possible. Since we chose to stick with a specific compiler, we had the luxury of studying the output assembly code and working it until the assembly couldn't be further improved. That experience has led to some generalizations.

4.1 Branching and Method Calls

Compilers are good enough that sequential sections of code are optimized very well and branches are the bulk of the execution time and code space. These are the logic of the program, and cannot usually be eliminated. The optimizer can be helped by avoiding conditional code, short loops should be unrolled, and it may be best to not branch around useless code in certain cases.

Object oriented designs, and C++ programs in particular, tend to introduce many smaller functions that perform near-trivial operations. For example:

```
class Foo {
        [...]
      int value()
          { return value_; }
      unsigned size() const
          { return 16; }
        [...]
};
```

The call to `value()` can be reduced to a single "`mov`" or "`ld`" instruction on most types of CPUs, and the `size()` method can be optimized completely away. However, if those methods were not defined inline then the compiler would be forced to generate a call and a return, would need to invalidate registers and/or shift register windows, and otherwise multiply the complixity.

By inlining, the call instruction is eliminated and the basic block is expanded to surround the method invocation. Subexpression elimination and register allocation can be applied more globally and code around the method call shrinks along with the method call.

C programs can also benefit from this technique. Linux source code, for example, is filled with tiny inlined functions of this sort. They are easier to read then similar macros, and more clearly express intent to the compiler.

4.2 Virtual Methods

Implementing virtual methods usually means an extra memory access before the call to the method. This sounds like a performance problem, but in fact it turns out to be a useful optimization, when used wisely.

A call to a C++ virtual method is often used to invoke a context-specific behavior. For example, one might use virtual methods to perform I/O on an abstract class Device. The typical C equivilent is to keep function pointers in a function, and use those function pointers to perform the operation.

The typical C code has the function pointers in the structure with the other variables, making the structure larger. Every instance has pointers to all the functions, so there are many pointers to every function. A more efficient way to do this is to keep in the structure only a pointer to a set of functions. Thus, the structure has only one pointer to represent the behavior functions.

C++ compilers do this automatically. The virtual table is generated by the compiler for each type, and placed in constant memory. Here is an example, for the i960, of a call to a virtual method following a pointer in sfoo:

```
        ld      _sfoo,g5
        ld      8(g5),g4
        ldis    8(g4),g0
        ld      12(g4),g4
        addo    g5,g0,g0
        callx   (g4)
```

This generated code loads into g4 the pointer to the function to be execute, and into g0 the pointer to the object. The external pointer variable "`sfoo`" contains the pointer to the object to be manipulated, and the virtual method takes no parameters. With carefully crafted C code, the programmer can save maybe one instruction here.

Obviously, however, this code is far too much to use for acessor methods, such as "`Foo::value()`" above. Although the virtual method performs a useful function efficiently, it is clearly too expensive for trivial functions. The common technique of creating a virtual method that returns a constant value to reveal the true object type is inefficient. If you must do this, run time type information is more practical.

One unexpected benefit of virtual methods is that in certain cases the compiler can tell a priori which implementation of a virtual method applies, and can optimize all the above away, as in:

```
        ld        _sfoo,g0
        ld        _sfoo+4,g4
        cmpible   g4,g0,L4
        mov       g4,g0
L4:
```

In this example, "sfoo" is an object and not a pointer. C++ knows exactly which implementation applies, and took the liberty of implementing the method inline. The only difference between this and the previous example is that "`sfoo`" is known exactly to be of type Foo.

4.3 Assembly Code

When dealing with "bare iron," some assembly code is inevitable. There is, for example, no reasonable way to implement thread switching entirely in C or C++, and in most processors interrupt and trap handlers must have assembly code to save the context of an interrupted thread and setup a fresh C/C++ calling sequence.

However, assembly code should typically be kept to a minimum. A human can do a good job of optimizing a small stretch of assembly code, but as the code grows, the human capacity to manage resource allocation becomes overwhelmed and the compiler performs better.

The paradox is that short assembly functions are more likely to be inefficient if placed in seperate source files, as the call and return overhead starts to overwhelm. What we really want is a way to write inline assembly code. Look at the following example:

```
inline int isr_hot_flag()
{
      register unsigned tmp;
      asm("modpc 0, 0, %0" : "=r" (tmp));
      return tmp & 0x2000;
}
```

This code returns true if the caller is running in an interrupt handler on an i960. The "`modpc`" instruction takes around 14 clock cycles to execute (it is slow) but that is not too bad. The `call` and `return` instructions on the i960Jx each consume 6 cycles and also push a register set, leading to at least 12 cycles for the call/return pair. Putting this function in a seperate source file would *double* the execution time of the function and may cause a register cache spill as well. Inlining this function is therefore rather important and powerful.

The benefits do not stop there. The GNU compiler syntax for inline assembly allows the programmer to match up registers and supply constraints that allow the C/C++ compiler to include the code in the optimization phase. The following degenerate case:

```
int foo() { return 0 && isr_hot_flag(); }
```

obviously leads to the following optimized i960 assembly code:

```
_foo__Fv:
        mov 0,g0
        ret
```

More complex situations are possible, but the point here is that the inline assembly can and should be included in the optimization passes of the compiler for the best results. A more complex example for the i386[7] works as follows:

```
inline int call_host1(int sys, int parm)
{
        asm ("int \$0x80\n"
                : "=a" (sys)
                : "0" (sys), "b" (parm) );
        return sys;
}
```

The previous code makes a system call by putting the parameter in the ebx register and the system call number in the eax register, and calling "`int $0x80`" to trap to the system. The compiler now knows how to allocate the eax and ebx registers around this assembly statement and can use that knowledge when optimizing register allocation around it. It can also eliminate the whole mess if the result is not at all used.[7]

This point doesn't have much to do with object oriented design (except that you can write methods in assembly code) but has everything to do with writing the low-level code in C++. If assembly code could not be inlined, an efficient implementation would necessarily pull more code into the assembly files to cut down on the cost of function call overhead. At that point, the C++ compiler would become more a hindrance then a help.

Including assembly code inline, and using constraints to control the optimizer, eliminates much of the interface overhead between C++ and assembly and gives the programmer the best of the C++ and assembly worlds.

4.4 Templates

This brings up an interesting theoretical optimization feature of templates. If in the previous example the class `Foo` were a template, the `size()` method could just return a template parameter. Thanks to template semantics, calls to the `size()` method are subject to constant elimination, which may in turn lead to dead code elimination. That is an example of the *compiler* deciding on the course of some branches, and eliminating the actual branch instruction from the execution stream.

Similar code in C uses preprocessor macros to get the same benefits, and is not type-safe. Templates used this way save much space and execution time, and are more readable. Use templates as a means of manipulating the type structure of the program, and not a way to create code, and templates may actually reduce to take no code space at all.

If template methods are not inlined, the compiler must generate an implementation of the template, often in a different compilation unit. To make matters worse, much near-duplicate code may be generated. For example, instantiations with the "`int`" type and the "`unsigned`" type may generate identical code for a small inlined method but generate unique implementations for complex out-lined methods. Comparisons, for example, require different assembly code for "`int`" and "`unsigned`" values.

Templates are therefore a terrific way to generate lots of excess code when used in this fashion. When inlined, the compiler may use template semantics to implement efficient, locally optimized code. Otherwise, they generate lots of redundant code.

Compiler writers are still arguing over how to deal with template repositories and the like, but in practice, for our purposes, the issue is moot. Large template classes or functions will expand to consume all available ROM. Practical templates should be small enough to be inline, and if inlined everywhere, there is no need for a template repository.

4.5 Smaller is Better

Software tends to expand to fill available memory, and programmers tend to judge their creations by the number of lines of code. There are two good reasons for worrying about program size, even when virtual memory is available and inexhaustible.

The most obvious reason to embedded programmers is that memory costs money and board space. Actually, it is the hardware designers who notice this first, and design in small amounts of memory. The programmer then asks for an additional 4Meg of flash memory and learns that 512Kbyte per chip means that 4Meg is 8 chips and the board just doesn't have that much space.

Large programs also tend to run slower. If it isn't simply because all those instructions take a long time to fetch from memory, it's because large programs overflow the instruction cache more often. Even dead code tends to spread the useful code into a larger address space and can lead to cache misses. Thanks to the widespread use of caches, memory

[7] If the assembly statement has side effects and should not be eliminated, it can be declared "`volatile`" and the compiler will preserve it.

and instruction, large programs are slow programs.

4.6 Link-time Efficiency

After all the compilation units are compiled, the program must be linked together with the run time library to create a single executable. On conventional operating systems, there are shared objects to be linked to, and the kernel to be made available. In a kernel-based operating system the common functions are placed in a protected kernel that all the applications share.

In an embedded environment, where there is only one application, the kernel is not being shared so its size should be included as part of the cost of the application. uCR is not kernelized, it is presented as a library that the linker uses to resolve symbols. Only the parts of uCR that are explicitly used are allocated space in the final image.

This is an example program, linked with uCR, that has only an empty main and the code necessary to enable all the devices on a Picture Elements ISE board. The text section includes executable code and constants and can be placed in ROM. The bss section is large because it contains generous stack space (10K) for the main thread.

```
text    data    bss     dec   filename
3520    1232    10664   15416 file.exe
```

The following is the same program compiled for Linux/SPARC. The stack space is not included in the totals, nor is the linux kernel itself or any of the 4 shared objects that this image links to.

```
text    data    bss     dec  filename
2688    2575    8       5271 a.out
```

Ignore the bss sections (mostly stack space in file.exe) and the linux image is about the same size. The linux image is for sparc whereas the uCR image is for i960 so the differences may easily be due to instruction set constraints.

What is significant about this example is that the entire uCR image takes up no more space then an image linked to run in an environment that has several megabytes of uncounted resources in the host kernel and shared objects. These resources provide a very important value in the case of Linux, but none of them are of interest in an embedded target.

As the program becomes more interesting, the uCR image sizes naturally increase. For example, the following numbers apply to a program that has included a debugger interface and code to drive a PCI bus interface, along with an implementation of a channel protocol that communicates with a driver on a host computer. In this image, 16K of buffer space is included in the bss number, along with the main stack.

```
text    data    bss     dec   filename
12544   2824    27284   42652 file.exe
```

The advantage of placing the uCR infrastructure in a library is that only the parts actually used are brought into the image. This can include individual methods of a class (those that are not inlined). A kernel image, on the other hand, must be included all at once or not at all.

5 Java, Anyone?

We have considered, and are still considering, the use of Java in embedded programming. There is an important problem with it, however, that reduces its usefulness, and that is the lack of an "asm" statement, and other inlining.

It turns out, when dealing with physical hardware, that there is always some little bit of assembly code that needs to be written. The obvious first thought is to put the assembly code is a library somewhere and call it when needed. But that is not necessarily the right answer. Consider the following familiar example for an Intel i960 microprocessor:

```
inline int  isr_hot_flag()
{
      register unsigned tmp;
      asm("modpc 0, 0, %0" : "=r" (tmp));
      return tmp & 0x2000;
}
```

This function basically reduces to the single instruction, the "modpc" instruction. Put that in a library and you get around it two branches and some register file shuffling, maybe even a few memory accesses. Wrap it up in a java class somewhere, and you also get the overhead of leaving and entering java.

A just-in-time compiler for java byte code would address many performance issues by generating unrolled directly executable machine code as the program runs. A syntax would be needed to specify specific assembly instructions for inclusion in the stream, or the compiler will not be able to match the optimization performance of C++.

6 Conclusions

uCR as a whole has become a richer environment, now that it works with more substantial processor boards. However, its original design goal to stay small and efficient seems to be working. The core library of uCR is still quite simple. We have also come to some conclusions on the matter of C++ and embedded programming.

We to this day face people telling us that C++ generates inefficient code that cannot possibly be practical for embedded systems where speed matters. The criticism that C++ leads to bad executable code is ridiculous, but at the same time accurate. Poor style or habits can in fact lead to awful results. On the other hand, a skilled C++ programmer can write programs that match or exceed the quality of equivilent C programs written by equally skilled C programmers.

The development cycle of embedded software does not easily lend itself to the trial-and-error style of programming and debugging, so a stubborn C++ compiler that catches as many errors as possible at compile time significantly reduces the dependence on run-time debugging, executable run-time support and compile/download/test cycles. This saves untold hours at the test bench, not to mention strain on PROM sockets.

7 Epilogue

There are times when the proper development environment for a project is not an operating system at all. For these times, run time suppport is all that is required for convenient development. If an operating system does not contribute to the solution, it is part of the problem and should not be there.

Without an operating system, however, the code generated must stand alone on the target hardware. The runtime support necessary to allow this, and even to add some extra features normally associated with operating systems (like threads and memory management) is fortunately not difficult or expensive.

uCR does a good job of providing the necessary runtime support to the compiler, with little overhead. Its small size comes from a careful attention to the details of performance, and stubborn controll of feature creep. The core design allows interesting functionality to be placed in libraries outside of uCR and brought in by the linker only if a programmer uses it.

With uCR, it is possible to get software for a new board up and running, if the CPU is already supported, in a few hours. It is an important milestone to get a trivial program running an infinite loop on a new processor board and it is best to not invest days getting there. Once trivial programs work, more extensive hardware debugging can commence.

The ongoing work on uCR is available for anonymous ftp from the Picture Elements ftp and web site, including the documentation, at:

`http://www.picturel.com/ucr/uCR.html`, and
`ftp://ftp.picturel.com/pub/source`.

It is indeed interesting that efforts to reduce code size and increase speed have led to the conclusion that significant portions of the source code must be visible to the programmer in the form of inline functions.

References

[1] American National Standaards Institute, 11 West 42nd Street, New York, New York 10036. *American National Standard for Programming Languages – C*, ansi/iso 9899-1990 edition, 1990.

[2] Grady Booch. *Object Oriented Design with Applications*, chapter 2, page 66. The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] Working paper for draft proposed international standard for information systems – programming language c++. http://www.cygnus.com/misc/wp/draft/index.html, April 1995.

[4] Cyclone Microsystems. *PCI Intelligent I/O Controllers User's Manual*.

[5] IEEE. *Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface*.

[6] Intel Corporation. *i960 RP Microprocessor User's Manual*.

[7] Intel Corporation. *Pentium Family User's Manual*.

[8] Intel Corporation. *MON960 Debug Monitor User's Guide*, 1995.

[9] Picture Elements, Inc. *Imaging Subsystem Engine - Theory of Operation*. Preliminary.

[10] P. J. Plauger. *The Draft Standard C++ Library*. Prentice Hall, 1995.

[11] PLX Technology. *PCI Bus Interface and Clock Distribution Chips*.

[12] Stephen Williams. Using ucr. http://www.picturel.com/ucr/uCR.html.