Modern C++ embedded systems â€“ Part 2: Evaluating C++

February 17, 2015

Having discussed the implementation of the main C++ language features in **Part 1** of this series, we can now evaluate C++ in terms of the machine code it generates. Embedded system programmers are particularly concerned about code and data size; we need to discuss C++ in these terms.

How big is a class? In C++, most code is in class member functions and most data is in objects belonging to these classes. C++ classes tend to have many more member functions than a C programmer would expect to use. This is because well-designed classes are complete and contain member functions to do anything with objects belonging to the class that might legitimately be needed. For a well-conceptualized class, this number will be reasonably small, but nevertheless larger than what the C programmer is accustomed to.

When calculating code size, bear in mind that modern linkers can extract from object files only those functions that are actually called, not the entire object files. In essence, they treat each object file like a library. This means that unused non-virtual class member functions have no code size penalty. So a class that seems to have a lot of baggage in terms of unused member functions may be quite economical in practice.

Although class completeness need not cause code bloat for non-virtual functions, it is reasonable to assume that all virtual functions of all classes used in a system will be linked into the binary.

How big is an object? The size of an object can be calculated by examining its class (and all its base classes). Ignore member functions and treat the data the same as for a struct. Then add the size of a pointer if there are any virtual functions in the class or base classes. You can confirm your result by using the sizeof operator. It will become apparent that the combined size of objects in a system need be no greater than the size of data in a C-based procedural model. This is because the same amount of state is needed to model a system regardless of whether it is organized into objects.

C++ and the heap

Heap usage is much more common in C++ than in C. This is because of encapsulation. In C, where a function requires an unknown amount of memory, it is common to externalize the memory as an input parameter and leave the caller with the problem. This is at least safer than mallocing an area and relying on the user to free it. But C++, with its encapsulation and destructors, gives class designers the possibility (and responsibility) of managing the memory used by objects of that class.

This difference in philosophy is evident in the difference between C strings and a C++ string class. In C, you get a char array. You have to decide in advance how long your string can be and you have to continuously make sure it doesnâ€™t get any bigger. A C++ string class, however, uses â€˜newâ€™ and â€˜deleteâ€™ to allow a string to be any size and to grow if necessary. It also makes sure that the heap is restored when the string is destroyed.

The consequence of all this is that you can scrape by in an embedded system written in C without using â€˜mallocâ€™ and â€˜freeâ€™, but avoiding â€˜newâ€™ and â€˜deleteâ€™ in C++ is a much bigger sacrifice.

The main reason for banning heap usage in an embedded application is the threat of heap

fragmentation. As the software runs, memory allocations of different sizes are acquired and released. The situation can arise where many small allocations are scattered through the heap and, although a large fraction of the heap may be available for use, it is all in small fragments and it is not possible to provide an allocation bigger that the largest fragment.

In an embedded system that runs continuously for years, heap fragmentation may occur only under certain conditions a long time after deployment, and may have been missed in test coverage.

Heap fragmentation can be avoided by using a non-fragmenting allocator. One solution is to re-implement operator â€˜newâ€™ and operator â€˜deleteâ€™ using a collection of fixed-size buffer pools. Operator â€˜newâ€™ returns the smallest available buffer that will satisfy the request. Since buffers are never split, fragmentation (or external fragmentation to be precise) does not occur. Disadvantages of this technique are that it uses more memory and that it must be configured to provide the right number of the right sized buffers.

Another alternative to banning heap usage outright is to allow it during initialization, but ban it once the system is running. This way, STL containers and other objects that use the heap can be modified and configured during initialization, but must not be modified once initialization is complete. With this policy, we know that if the system starts up, it wonâ€™t run out of memory no matter how long it runs. For some applications, this level of flexibility is enough.

ROMable objects

Linkers for embedded systems allow const static data to be kept in ROM. For a system written in C, this means that all the non-varying data known at compile time can be specified by static initializers, compiled to be stored in ROM and left there.

In C++, we can do the same, but we tend not to. In well-designed C++ code, most data is encapsulated in objects. Objects belong to classes and most classes have constructors. The natural object-oriented equivalent to const initialized data is a `const` object. A `const static` object that has a constructor must be stored in RAM for its constructor to initialize it. So where in C a const static object occupies cheap and plentiful ROM, its natural heir in C++ occupies expensive and scarce RAM. Initialization is performed by start-up code that calls static constructors with parameters specified in declarations. This start-up code occupies more ROM than the static initializer would have.

So if a system includes a lot of data that can be kept in ROM, special attention to class design is needed to ensure that the relevant objects are ROMable. For an object to be ROMable, it must be capable of initialization by a static initializer like a C `struct`. Although the easy way to do this is to make it a simple C `struct` (without member functions), it is possible to make such a class a bit more object-oriented.

The criteria for a static initializer to be allowed for a class are:

- The class must have no base classes.
- It must have no constructor.
- It must have no virtual functions.
- It must have no private or protected members.
- Any classes it contains must obey the same rules.

In addition, we should also require that all member functions of a ROMable class be const. A C struct meets these criteria, but so does a class that has member functions.

Although this solves the ROMability problem and enhances the C struct with member functions, it falls far short of the object-oriented ideal of a class that is easy to use correctly and difficult to use

incorrectly. The unwary class user can, for example, declare and use a `non-const, uninitialized` instance of the class that is beyond the control of the class designer.

To let us sleep securely in our object-oriented beds, something more is needed. That something is â€˜class nestingâ€™. In C++, we can declare classes within classes. We can take our dubious class that is open to misuse and put it in the private segment of another class. We can also make the const static instances of the dubious class private static members of the encapsulating class. This outer class is subject to none of the restrictions that the ROMable class is, so we can put in it a proper restricted interface to our const static data.

To illustrate this discussion, let us consider a simplified example of a handheld electronic multi-language dictionary. To keep it simple, the translator translates from English to German or French and it has a vocabulary of two words, â€˜yesâ€™ and â€˜noâ€™. Obviously, these dictionaries must be held on ROM. A C solution would be something like **Listing 20**.

```
/* A C ROMable dictionary */

#include <stdio.h>

typedef struct {
    const char* englishWord;
    const char* foreignWord;
} DictEntry;

const static DictEntry germanDict[] = {
    {"yes", "ja"},
    {"no", "nein"},
    {NULL, NULL}
};

const static DictEntry frenchDict[] = {
    {"yes", "oui"},
    {"no", "non"},
    {NULL, NULL}
};

const char* FromEnglish(const DictEntry* dict, const char*
english);

const char* ToEnglish(const DictEntry* dict, const char*
foreign);

/* ... */

int main() {
    puts(FromEnglish(frenchDict, "yes"));
    return 0;
}
```

Listing 20: A C ROMable dictionary

A Dict is an array of `DictEntry.` A `DictEntry` is a pair of `const char*` pointers, the first to

the English word, the second to the foreign word. The end of a Dict is marked by a DictEntry containing a pair of NULL pointers. To complete the design, we add a pair of functions that perform translation from and to English using a dictionary. This is a simple design. The two dictionaries and the strings to which they point reside in ROM.

Let us now consider what happens if we produce a naÃ¯ve object-oriented design in C++. Looking at **Listing 20** through object-oriented glasses, we identify a class Dict with two member functions: `const char* Dict::fromEnglish(const char*`, and `const char* Dict::toEnglish(const char*).` We have a clean and simple interface. Unfortunately, **Listing 21** wonâ€™t compile. The static initializers for `frenchDict` and `germanDict` try to access private members of the objects.

```
// NOT a ROMable dictionary in C++

#include <iostream>
using namespace std;

class Dict {
public:
    Dict();
    const char* fromEnglish(const char* english) const;
    const char* toEnglish(const char* foreign) const;
private:
    enum { DictSize = 3 };

    struct {
        const char* english;
        const char* foreign;
    } table[DictSize];
};

// *** Following wonâ€™t compile ***
const static Dict germanDict = {
    {
        {"yes", "ja"  },
        {"no",  "nein"},
        {NULL, NULL}
    }
};

// *** Following wonâ€™t compile ***
const static Dict frenchDict = {
    {
        {"yes", "oui" },
        {"no",  "non" },
        {NULL, NULL}
    }
};

// ...

int main() {
```

```
        cout << germanDict.fromEnglish("yes");
        return 0;
    }
```

Listing 21: A C++ ROMable dictionary NOT!

If we make these members public and eliminate the constructor as in **Listing 22**, the class will meet the criteria for static initializers and the code will compile, but weâ€™ve broken encapsulation. Users can see the internal implementation of the class and bypass the intended access functions. Even worse, they can create their own (`con-const`) instances of `Dict` whose internal state is outside our control.

```
    // A ROMable dictionary in C++, but with poor encapsulation

    #include <iostream>
    using namespace std;

    class Dict {
    public:
        const char* fromEnglish(const char* english) const;
        const char* toEnglish(const char* foreign) const;

    // PLEASE donâ€™t access anything in the class below this
comment.
    // PLEASE donâ€™t create your own instances of this class.

        enum { DictSize = 3 };

        struct {
            const char* english;
            const char* foreign;
        } table[DictSize];
    };

    onst static Dict germanDict = {
        {
            {"yes", "ja"},
            {"no", "nein"},
            {NULL, NULL}
        }
    };

    const static Dict frenchDict = {
        {
            {"yes", "oui"},
            {"no", "non"},
            {NULL, NULL}
        }
    };

    // ...
```

```
int main() {
    cout << germanDict.fromEnglish("yes");
    return 0;
}
```

Listing 22 : A C++ ROMable corruptable dictionary

Now, letâ€™s do it right. In **Listing 23**, the class Dict in Listing 22 becomes Table, which is nested privately within the new class `Dict. Class Dict` also contains, as a static member, an array of Tables, which we can initialize statically. The function `main()` shows use of this class Dict, which has a clean interface.

```
#include <iostream>
  using namespace std;
  class Dict {
  public:
      typedef enum {
          german,
          french
      } Language;

      Dict(Language lang);

      const char* fromEnglish(const char* english) const;

      const char* toEnglish(const char* foreign) const;

  private:
      class Table {
      public:
          const char* fromEnglish(const char* english) const;
          const char* toEnglish(const char* foreign) const;

          enum { DictSize = 3 };

          struct {
              const char* english;
              const char* foreign;
          } table[DictSize];
      };

      const static Table tables[];

      Language myLanguage;
  };

  const Dict::Table Dict::tables[]= {
      {
          {
              {"yes", "ja"},
```

```
                {"no", "nein"},
                {NULL, NULL}
            }
        },
         {
            {
                {"yes", "oui"},
                {"no", "non"},
                {NULL, NULL}
            }
        }
    };

    // ...

    int main() {
        Dict germanDict (Dict::german);
        cout << germanDict.fromEnglish("yes");
        return 0;
    }
```

Listing 23: A clean C++ ROMable dictionary

So to make the best use of object-oriented design for data on ROM, special class design is needed.

Title-1

Reality check

Having minutely examined the costs of C++ features, it is only fair to see how C stands up to the same degree of scrutiny. Consider the C code in **Listing 24**. One line contains a floating point value, which will pull in parts of the floating point library and have a disproportionate effect on code size if floating point is not needed. The next line will have a similar effect, `if printf` has been avoided elsewhere. The next line calls `strlen(s) strlen(s)` times, rather than once, which has a serious impact on execution time.

```
/* Reality check - some things to avoid in C */

    #include <stdio.h>
    #include <string.h>

    int main() {
        char s[] = "Hello world";
        unsigned i;

        int var =1.0;
        printf(s);
        for (i=0; i < strlen(s); i++) {
            /* ... */
        }
        return 0;
    }
```

Listing 24: C reality check

But who would argue that these mistakes are reasons to not use C in embedded systems? Similarly, it is wrong to brand C++ as unsuitable for embedded systems because it can be misused.

Class libraries and embedded systems

A major benefit of using C++ is the availability of class libraries. Object-oriented design makes class libraries easier to use (and harder to misuse) then their procedural predecessors. But, as with procedural libraries, some class libraries cause excessive code bloat for the functionality they provide. If memory footprint is a concern, it is wise to evaluate a library before committing to it and measure what happens to memory footprint when the library is used.

The Standard C++ Library includes several components that are of use in a variety of embedded systems. STL containers and algorithms support handling of collections of data. The std::string class supports strings of variable length.

STL containers and std::string use the heap, with consequences that have already been discussed. Also, because STL containers and algorithms are templates, memory consumption should be tracked to make sure that they don't lead to code bloat.

The std::ostream class supports outputting a string representation of an object independent of type, which can be useful for diagnostic purposes and in templates. Some implementations, however - even some provided in toolchains for embedded systems - have a very large code size for what they do. In a recent project, I had to write a lightweight alternative to std::ostream to avoid its excessive memory footprint.

Another valuable library for embedded systems is the **boost library**. The library is free. It is widely used and is compatible with a range of compilers. It consists of well over 100 diverse classes and templates including smart pointers, regular expressions, circular buffers, command line parsing, logging, etc. Most components are independent of the others, which keeps the memory footprint small.

But building boost for an embedded system is not a trivial undertaking, unless it is supported by the toolchain provider. Instructions are provided on the boost web site, including compilation with different compilers.

C++11 in embedded systems

Everything said so far applies to C++11 (and C++14). Now let us examine the major new language features in C++11 from the point of view of runtime cost, both memory consumption and execution time. We won't discuss additions to the Standard C++ Library, which is outside the scope of this article. We will list features that have no runtime cost, features that offer improvements in either speed or size and finally, features that have a runtime cost in either size or speed associated with their use. The following new features in C++11 have no runtime cost:

`auto` The auto keyword has a new meaning in C++11 that is different from its meaning in C. It means 'automatic' variable type determination. So 'auto x = y;' means that the type of x is the same as the type of y.

`decltype` Decltype evaluates the type of an expression.

`override` and **`final, =delete`** and **`=default`** The '`override`' specifier on a member function means that it overrides a member function in a base class and it is an error if no such base class function exists. Similarly, '`final`' means that this function must not be overridden in a subclass. The `=default` and `=delete` modifiers are used to control the use of compiler generated default functions like copy constructors.

`Range-based for` The range-based for statement '`for (auto x: container`' is transformed by the compiler into a regular `for` loop. The `for` loops in the following fragment in **Listing 25** are equivalent.

```
#include <vector>

int main() {
    std::vector<int> v;
    for (auto it = v.begin(); it != v.end(); ++it) {
        auto& x = *it;
        /* ... */
    }
    for (auto& x: v) {
        /* ... */
    }
    return 0;
}
```

Listing 25: Range-based 'for'

`Suffix return type` C++11 allows the return type of a function to be at the end of a declaration instead of the beginning. This is useful in some template code. The following statements in **Listing 26** are equivalent.

```
int f();
auto f() -> int;
```

Listing 26: Suffix return type

Lambda Lambda expressions are equivalent to a locally-defined function object class and have the same costs. The benefit of a lambda expression is a much terser syntax.

initializer_list STL containers support initialization from an 'initializer list', specifically `template<typename T> std::initializer_list<T>.` An initializer list can be stored in ROM, so, although the STL container occupies RAM, the data to populate it can have a compact representation in ROM.

Features that can increase speed or reduce size

Move constructors This feature transfers the state of one object to another object while removing it from the original object. Move constructors are provided by the class programmer and called by the compiler instead of a copy constructor where it is known that the original object state will no longer be needed. A common example is when a function returns an object by value. Without a move constructor, this would result in the object being copied and then the original object being destroyed, both potentially expensive operations.

The Standard C++ Library exploits move constructors, so using C++11 may yield significant performance improvements over C++03 in STL containers and std::strings without any changes to application code.

constexpr The new keyword `constexpr` indicates an expression or a function that can be evaluated at compile time. It is fairly straightforward, for example to write a constexpr function that evaluates whether a number is prime.

But note the emphasis on the word ‘can’. A compiler that ignores the keyword is still a legal C++11 compiler and the things that can be done in a constexpr function are limited. Loops and variables are both prohibited. So use of constexpr can result in a function that compiles, but is evaluated at runtime instead of compile time. This can be extremely inefficient because of the limitations on constexpr functions.

We can anticipate compilers that make sophisticated computations at compile time, saving both memory and execution time, but we’re not there yet. Visit **gcc.godbolt.org** to examine machine code generated by a number of modern compilers and how much they do with constexpr functions.

noexcept The noexcept specifier guarantees that a function won’t throw an exception. This allows the compiler to do some additional optimizations, which can speed execution and marginally reduce code size.

(Warning: It is left entirely to the programmer to make sure that an exception doesn’t propagate to the point where it would leave a noexcept function. If it does, `std::terminate()` is called. Typical compilers don’t warn about this even in the most obvious cases.)

Speed or size penalties As explained above, using constexpr functions without checking the code generation can result in inefficient runtime evaluation instead of compile time evaluation.

Title-1
More about lambdas
For most C++11 features, once you understand what they do, it isn’t a big deal to understand how they do it. But lambdas are not so easy.

To explore what lambdas do and how they do it, consider the following coding problem. We need to write a function that counts the number of values in the array that are less than a parameter. In C, we might write something like this.

```
    int count_less_than(int const* first, int const* last, int
value) {
        int result = 0;
        while (first != last)
            if (*first++ < value)
                ++result;
        return result;
    }
```

Listing 27: C function to count integers less than value

When this was compiled using gcc 4.9.0 with option –Os on **godbolt.org**, the following sequence of 10 machine code instructions was generated.

```
        xorl    %eax, %eax
.L2:    cmpq    %rsi, %rdi
        je      .L6
        addq    $4, %rdi
        xorl    %ecx, %ecx
        cmpl    %edx, -4(%rdi)
        setl    %cl
        addl    %ecx, %eax
```

```
            jmp     .L2
.L6:        ret
```

Listing 28 : Machine code for Listing 28

In C++, we can use the `count_if` algorithm (see **http://en.cppreference.com/w/cpp/algorithm/count**) to eliminate the pointer arithmetic. But `count_if` needs a function or functor to call that takes one argument.

So what's a functor? A functor is an object that looks and acts like a function. It has `operator()` defined that takes parameters and returns a result. Unlike a function, a functor may have internal state in the form of member variables. Functors are often used in conjunction with algorithms like `count_if` to serve as callbacks.

A solution using count_if could look something like this:

```
#include <algorithm>

    class less_than_threshold {
    public:
        less_than_threshold(int threshold): value(threshold) {
        }

        bool operator()(int x) const {
            return x < value;
        }
    private:
        int value;
    };

    int count_less_than(int const* first, int const* last, int
value) {
        less_than_threshold ltt(value);
        return std::count_if(first, last, ltt);
    }
```

Listing 29: Count integers less than a value using count_if and a functor

The Class `less_than_threshold` is a functor. It stores a threshold value in the member variable value. When it's called like a function it returns true if the parameter to the call is less than value. Now we've eliminated the pointer arithmetic from the C example, but we had to add a whole class (`less_than_threshold`) to do it.

Lambda expressions allow us to do all that much more succinctly. With a lambda expression, we can add the class `less_than_threshold` at the point in the code where we need it as shown in **Listing 30.**

```
    #include <algorithm>

    int count_less_than(int const* first, int const* last, int
value) {
        auto ltt = [value](int x) { return x < value; };
```

```
        return std::count_if(first, last, ltt);
    }
```

Listing 30: Count integers less than a value using a lambda

In this version, the `ltt` object is defined using a lambda expression. It captures the local variable value, takes a parameter x, and evaluates whether x is less than value.

The last example puts the lambda definition on a separate line for clarity. This isn't necessary. The function `count_less_than` can be reduced to one line as shown in **Listing 31**.

```
    #include <algorithm>

    int count_less_than(int const* first, int const* last, int
value) {
        return std::count_if(first, last, [value](int x) { return x
< value; });
    }
```

Listing 31: Count integers less than a value in one line

As shown in **Listing 32**, when the code from the previous three listings was compiled using g++ 4.9.0 using –Os, machine code generated was identical in all three cases and the same length as Listing 28.

```
        xorl    %eax, %eax
.L8:    cmpq    %rsi, %rdi
        je    .L11
        xorl    %ecx, %ecx
        cmpl    %edx, (%rdi)
        setl    %cl
        addq    $4, %rdi
        addq    %rcx, %rax
        jmp    .L8
.L11:   ret
```

Listing 32: Machine code for Listing 29, Listing 30 and Listing 31

C++14 in embedded systems
C++14 is a 'maintenance upgrade' of C++11 with minor extensions and bug fixes. For an explanation of the major features and links to information on compiler support, see **The C++14 standard and what you need to know.**

As we did for C++11, we will examine the major features in C++14 from the point of view of runtime cost, both memory consumption and execution time. Compiler support for these features is still patchy, but hopefully will improve quickly.

Digit separators and binary literals C++14 adopts binary literals, indicated by a prefix of 0b or 0B, that are already supported by some C and C++ compilers. It also allows a single quote (') to be used as a separator in both integer and floating point literals. The separator is ignored by the compiler, but makes the numbers easier for humans to read. Here are some examples:

C++ 11	100000 0	1000000 00	0xaaaa	0xffffffffffffffff ffULL	123456 .0
C++ 14	1'000' 000	100'000 '000	0b1010'1010'10 10'1010	0xffff'ffff'ffff 'ffff'ULL	123'45 6.0

Function return type deduction In C++14, we can declare a function return type as ‘auto’ and have the compiler figure it out from return statements in the function. This seems to be a step backwards in terms of separating interface and implementation, but it is convenient for small inline functions and it has no runtime costs.

Variable templates Previously, templates were used only for generating classes and functions. Now they can be used to generate variables as well. For example:

```
template<typename T> constexpr T pi = T(3.1415926535897932385);
// pi<float> is pi as a float.
// pi<double> is pi as a double
// pi<std::complex<double>> is pi as a complex number.
```

Listing 33: Variable template

Features that can increase speed or reduce size
Generic lambdas As shown in **Listing 34**, generic lambdas are like templates, but with a clearer, more elegant, syntax. Their size and speed trade-offs are similar to templates.

```
// Generic lambda with "auto" parameters
auto add = [](auto x, auto y) { return x + y; }

// How it might have been done, but wasn’t. Won’t compile.
template<typename T1, typename T2> auto add = [](T1 x, T2 y) {
return x + y; }
```

Listing 34: Generic lambda

Relaxed constexpr C++14 is less restrictive about what it permits in a constexpr function. Variables, if statements and loops are all allowed. This means that a constexpr function can be written more like a regular function and is less likely to produce poor performance if evaluated at runtime. But the caveat for C++11constexpr functions - that they may be evaluated at runtime rather than compile time - still applies.

Generic lambdas and relaxed constexpr can improve size and/or speed, but if used inappropriately can have the opposite effect.

Compilers and tools for embedded systems do this in 18 point and space after

Most C cross-compiler vendors for 32-bit and 64-bit targets offer C++ compilers. But because C, rather than C++, is still the default language for embedded development for a resource-constrained environment, we can’t assume that a successful toolchain has good C++ support.

Here are some things to check when choosing a toolchain for embedded C++ development:

What size is ‘Hello World’? The memory footprint of ‘Hello world’ – the C++ version, not the C version – may be surprisingly large.

How do strings and STL containers affect memory footprint? Add string and STL container operations to 'Hello world' and see if the increase in memory footprint is reasonable. Although STL containers are template instantiations, skilful library design can minimize the amount of code unique to each instance. So, for example, an std::map<std::string, int> can share a lot of code with an std::map<std::string, unsigned> because they share the same key type std::string and a lot of the code in std::map depends only on the key type, not the value type. Does the std::map supplied with the compiler exploit this?

Does the debugger work well for C++? Can you set a breakpoint in an inline function or in a template? Does it actually work if you do? Can you easily look at the state of an object?

How well does it support C++11/14? You can get by perfectly well using C++03, but support for C++11 and C++14 reflects the vendor's commitment to supporting C++ developers. At the time of writing, compiler support for C++14 features is sparse.

Is boost available for/with the toolchain? Boost is a valuable library, but cross-compiling it isn't for the faint-hearted. If the vendor has done it already, that's a measure of the vendor's commitment and saves a lot of work.

Conclusion

Most C++ features have no impact on code size or on speed. Others have a small impact that is generally worth paying for. To use C++ effectively in embedded systems, you need to be aware of what is going on at the machine code level, just as in C. Armed with that knowledge, the embedded systems programmer can produce code that smaller, faster, and safer than is possible without C++.

Part 1: Modern C++ in Embedded Systems - Myth and Reality

Further information

The C++ Programming Language, 4th Edition by Bjarne Stroustrup is a very readable reference book for C++11.

C and C++ Reference is good for immediate specific information about most versions of C and C++ and their standard libraries.

Effective C++ in an Embedded Environment by Scott Meyers goes into much more detail on some of the issues addressed here, including vtable implementation with multiple inheritance, virtual inheritance, etc.

Godbolt's **Interactive Compiler** allows you to play with different C++ compilers and instantly see the code generated.

Dominic Herity *is a Principal Software Engineer at Faz Technology Ltd. He has 30 years' experience writing software for platforms from 8-bit to 64-bit, with full life cycle experience in several commercially successful products. He served as Technology Leader with Silicon & Software Systems and Task Group Chair in the Network Processing Forum. He has contributed to research into Distributed Operating Systems and High Availability at Trinity College Dublin. He has publications on various aspects of Embedded Systems design and has presented at several conferences in Europe and North America. He can be contacted at* ***dherity@gmail.com****.*