C++ in an Embedded Environment

Andrew McRae

Megadata Pty Ltd.
2/37 Waterloo Rd
North Ryde
andrew@megadata.mega.oz.au

This paper describes ongoing efforts to use C++ in an embedded environment, and details the porting of the C++ support environment, as well as the pecularities (benefits and otherwise) of using and programming with C++ in a different environment to typical computing platforms.

1. Introduction.

Programming embedded systems tends to be a specialised field, and often is seen as a less disciplined art than programming 'real' computers. Why is this so? The embedded environment contains some unique challenges that are not trivial to deal with. This paper presents one attempt to marry the diverse worlds of embedded systems and modern object-orientated programming languages, and describes the porting of the language support, problems encountered when using C++ in this area, and the real benefits experienced.

2. Embedded Systems.

Traditionally, embedded systems have been usually programmed in Assembler language. Reasons cited for this fall broadly into two categories; resource (memory/cpu) limitations dictated low-level control over the code, and often the code interfaced closely with hardware; secondly, the available high level languages did not provide the facilities necessary for embedded coding, or provided them at too high a cost to warrant their use. The advent of C as a full featured, yet portable and efficient, programming language provided embedded developers with an opportunity to move away from Assembler to more structured coding methods. The rapid acceptance of C in the embedded area testifies to its suitability to the environment.

At the same time the rapid technology improvements and downward price curves has allowed embedded products to contain better and faster CPUs and much more memory than was previously even thought possible (though programmers will *always* find ways to fill up that last 1K of RAM). This gave greater freedom in choosing the coding tools and methods. Designers and coders were freed from the confines of cramming functionality into small spaces, and could employ newer and better structured design and coding techniques (though there are still some coders who say that this is not necessarily a *good* thing, and it does people good to have to get a program working in under 4K of ROM).

Megadata has been using C in embedded systems for over 8 years. A natural progression was to continue the trend towards higher level languages by using C++ in the embedded environment. Here beginneth that tale.

3. The Environment.

It is hard to write software for embedded systems. There are a number of reasons for this, namely:

- Lack of hardware support for memory protection. A common error is inadvertant corruption of memory associated with another process (or a driver, or the kernel), with the predictable unpredictable results.
- Software errors can lead to system crashes that can leave the processor or memory scrambled, which makes post-mortem debugging impossible i.e No *core* files that you can examine and backtrace with

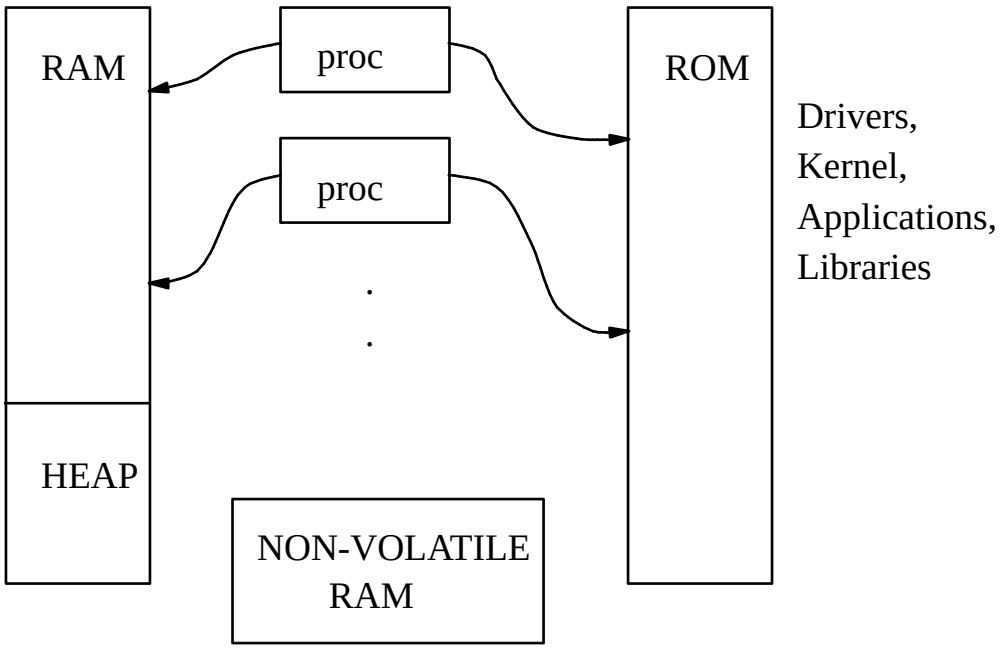
your symbolic debugger.

- Resource limitations such as memory, CPU performance etc. often impose a strict regime of resource allocation, especially since memory is usually real rather than virtual. This can lead to obscure coding practices, and often good design is forsaken in the (mistaken) belief that elegant designs cost more in resources.
- Arcane operating environments. Often hardware requirements dictate the software environment, such as device interfaces, system services etc. This can often lead to long (and high) learning curves.
- A requirement for robustness and fault tolerance. Most embedded systems must operate under unmanned and often adverse conditions, such as hardware failure, resource exhaustion, strict timing requirements etc, whereas under UNIX† an error might be handled by printing an error message and calling *exit()*, something which is not usually an option in an embedded system.

The approach that Megadata has taken in providing a usable and effective platform for developing embedded systems is to provide an environment and API closely conforming to a UNIX programming environment. This allows code to be prototyped on UNIX workstations running friendly development tools, reduces the learning curve for new developers, and provides a common foundation for sharing source code.

The best way to describe the execution environment is as single shared address space with multiple executing threads. Each thread is like a separate process under UNIX , which can perform device and network I/O, start other processes, have separate stacks etc. (and thus are heavier in weight than normal threads). All processes share the same address space (both program and data space). Kernel services are accessed as subroutines from the application code, and there is no kernel/user memory distinction (this is a mixed blessing; driver/application transfers are fast and efficient, but no memory protection can be enforced).

Diagramatically, this appears as:



As can be seen, RAM is split into two sections; the first is the usual (BSS) preallocated static locations, and the second is a common memory pool (the heap), which is managed using the standard free store subroutines *malloc* and *free*. The same memory pool is shared for virtually all dynamic memory allocation such as process table blocks (including stack), driver control blocks, application memory structures, and library requirements. The only other memory allocation scheme used is the network *mbuf* scheme, which is

† UNIX is a trademark of Bell Laboratories.

designed for virtual memory and requires that the memory accessed lies in a contiguous address space. The RAM heap is also used as the free store managed by the C++ *new* and *delete* operators.

Most embedded systems also have some separate non-volatile memory store i.e a memory bank where the contents are preserved when power is off or if the system is restarted. This is often EEPROM, but can also be static RAM in SmartSockets™. One of the goals of the porting exercise was to allow C++ persistent objects by using classes and methods that understood the non-volatile memory store.

4. Run Time Support.

C++ is somewhat more demanding than C in its requirements for runtime support. Examples include global and static construction of objects, inbuilt free store management, and assumptions (expectations?) of runtime exception conditions such as out-of-memory handling. The challenge was to preserve these natural C++ features without compromising the robustness requirements.

4.1. Free Store Management.

The C++ free store mechanism was easily implemented using the available *malloc* and *free* calls. As long as enough contiguous memory exists to satisfy the request all is well. A problem arises when the memory pool is too small or too fragmented to provide the memory. Under UNIX this usually results in a message to *stderr* and a call to *abort*, which terminates the program; rather ungraceful, but arguably a reasonable thing to do given the circumstances. Often the C++ support libraries allow user hooks to catch these exceptional cases, and provide customised graceful exiting (as long as the exiting does not itself require more memory!). Rarely does the application code have to deal with failure of memory allocation (usually reflected as a failure to create an object instance via *new*), and so most programs simply assume that objects are **always** created correctly.

In an embedded application it is important that such conditions be dealt with rationally and securely. The following strategies were formulated in an attempt to provide for the possibility of free store exhaustion.

- The successful creation of an object is never assumed. All object instances created with *new* are checked to ensure that the creation worked.
- In the event of free store exhaustion, no application or system structure should be left in an inconsistent state. No partially initialised objects can be tolerated.
- Non-critical objects (such as caching, lists etc.) are linked to a low memory handler so that they can be reclaimed upon demand.

These strategies are aimed at coping with temporary memory high water marks. Ultimately the only real solution for scarcity of free store is more memory.

4.2. Constructors and Destructors.

C++ allows global objects that are constructed ‘by magic’ before any user code is allowed to run, or at least before *main* is called. Within the GNU C++ framework this is achieved by the compiler detecting the compilation of the function named *main*, and emitting as part of the function prologue a call to a support routine named *__main*. The *__main* routine processes the list of global constructors, and finally returns to the real *main* where the user code now sees that all the global objects are constructed. The use of *__main* relies on the convention that *main* is the start of the user executable code.

Within the embedded environment this may not be the case, so separate execution of global constructors was explicitly added as part of the kernel initialisation.

Global destructors within a UNIX environment are executed as part of the termination of program, usually as part of the *exit* processing. These destructors may have important tasks to perform such as updating files, logging messages etc. In embedded systems there is no such equivalent, as most of these systems execute until power is turned off. Under UNIX of course, when a process terminates, the memory image containing the object instances is, to all intents and purposes, lost. In an embedded system processes can create objects, and when the process exits it should only delete the objects that it created. Having a common address space and compile-time name space allows processes to create objects that other processes can

access, use or destroy as necessary. An example is an alarm list object, where an initialising process creates the list and allocates a certain number of entries. Any other process can access that list, and extend the number of entries, or remove old entries etc. This is a powerful mechanism, but is somewhat foreign to the normal C++ environment. It implies that every process must ensure that every object created is accounted for, and is either deleted, or responsibilty for that object given to another process. To provide for a better memory garbage collection model, and for the possibility that processes may unexpectedly exit, objects may be attached to a process so that when that process exits, the objects are automatically deleted.

4.3. Persistent Objects.

Typical Megadata embedded systems have some form of non-volatile store, that contains configuration information, downloaded control sequences, and telemetry history data. Whilst a formalised C interface exists to access this memory, it was natural to use C++ objects to hide the details of the memory store; and so a base class was developed that allowed persistent objects to be created and manipulated. Much more flexible use can now be made of the non-volatile store, and C++ objects can inherit the persistence class to provide non-volatile storage of critical data.

5. Benefits of C++.

Much has been written about OOPLs, and much specifically about C++. There are the inevitable flame wars, the nit-picking, the well reasoned critiques, the purist vs. realist arguments and so forth, but when all is said and done it looks like C++ will overtake C as the primary systems programming language of the ’90s. A lot is said about the benefits of C++ over C in the general computing world, but what advantages does it have in the embedded world? Most advantages enjoyed by C++ in a ‘normal’ environment are magnified in the embedded world. Common areas of difficulty in the embedded area are helped greatly by the employing the object orientated features of C++. Some direct benefits that have been experienced are worthy of more specific mention.

5.1. Stricter Type Checking.

Whilst the stricter type checking of C++ (and the type-safe linkage that goes along with it) is of benefit to UNIX level coding, it provides a much greater degree of confidence in embedded systems development. Common problem areas such as passing values themselves rather than pointers to values are discovered at compile time. Strict checking of library routine interfaces allow compile time confirmation of correct parameter types and number. When developing code for embedded systems, experience has shown that it is an order of magnitude easier to find problems by comprehensive checking at compile time, rather than debugging at run time.

5.2. Name Space Management.

When the embedded system is one large process address space (and is linked as one large name space), then there is increasing problems with name space management, both with avoiding re-use of function names, and also with inadvertant use of the same name of a global variable. Complicated naming conventions can spring up to avoid pollution of the name space, which quickly tends to create chaos in the naming of functions and variables. An example is the UNIX kernel itself. Any browse through a kernel namelist will quickly show where whole sections of the kernel are delineated through use of name prefixes, such as *so, sys, kern_, vfs_* etc. Even worse is the allocation of device driver names.

C++ neatly avoids these name space problems by hiding the scope of the names within the object framework, as well as allowing type-safe name linkages so that the same function name may be used with different arguments and return types (but resolve to different functions).

5.3. Modularity and Information Hiding.

The use of C++ objects to modularise code has tremendous advantages in the embedded world. This allows a greater degree of code reuse across embedded and non-embedded systems by hiding the details of hardware and system interfaces. One example is the creation of standard building blocks such as timers, telemetry communication line protocol handlers, telemetry I/O objects. Under an embedded system these objects may map to lower layer device drivers or to hardware itself. On UNIX the same objects (or at least

the same object interface) may map to simulated hardware, or different STREAMS drivers etc. Thus code can be developed and operated on UNIX , and then ported to an embedded system with a high degree of confidence in the correctness of the code.

5.4. Performance Gains.

The use of *inline* functions, especially when used as an object method, allows structured and safe coding without compromising performance constraints.

5.5. Free Store Management.

The use of *new* and *delete*, along with the scope related creation and deletion of objects, provide safer and more consistent management of the available memory pool. It becomes much easier to correctly structure memory use by employing objects to hide private data, and so enforce the application code's view of data. It also resolves common problems with passing incorrect pointers to *free*, or obtaining memory using *malloc* and then forgetting to free it (causing a memory hole). This typically occurs when a subroutine allocates some memory via *malloc*, and during some processing a premature return taken that does not free the allocated memory. Using an automatic C++ object containing a constructor that allocated the memory means that whenever the object went out of scope the memory could be freed without the programmer having to keep track of the memory use.

6. Disadvantages of C++.

Are there specific disadvantages to C++ in the embedded world? It is hard to classify the following points as 'disadvantages' *per se*, but they are worth mentioning as issues that need to be kept in mind. They are probably more related to C++ programmers and designers than C++ the language.

6.1. Reliance on Run Time Support.

C++ relies on the free store operators and global constructors/destructors; that is to say, C++ does not *require* these facilities, but much useful functionality is lost by not allowing them. This implies some overhead or extra support that C does not need, and, under some circumstances, what can seem to be perfectly legitimate code can fail because the programmer is not aware of the underlying support requirements. An example is the declaration of an object that fails to be correctly constructed (possibly due to memory limitations); unless the object has some way of flagging its inconsistent state **and** the programmer actually checks the object's state, the code may crash in unpredictable ways. The problem is that the programmer may not be aware of the possibilities of the object being in an inconsistent state. This issue is explored further in the next section.

6.2. Invisibility of Code.

This is a subtle issue which is related to the programmer's visibility and awareness of operations in developing software in the embedded field. By using C++ objects, the programmer is often not aware of the cost of the object, both in memory terms and the CPU processing time of the object, and naive coding may cause resource exhaustion or unacceptable performance problems. What this boils down to is that the programmer may be thinking he is just accessing a variable, but he may be causing an unknown amount of processing to go on behind the scenes. This is also a object library design issue, where the designer and implementor generally should be aware of the intended use and users of the library (and that the users are aware of the object's nature). Badly designed libraries can also lead to software bloat. This is usually much less of a problem on UNIX , as people are getting so used to software bloat that a little more doesn't seem to make any difference. In embedded systems where memory and CPU resources may be severely constrained, this could be a major limitation.

The answer to this problem is design and code your libraries right, and choose the right people to use the libraries. The rest is easy.

7. Future Work.

C++ is an evolving language. One extension that has been adopted into the standard is *exceptions*. Exceptions are especially attractive in an embedded situation, where errors such as out-of-memory, maths errors etc can be specifically caught and dealt with as part of the language (rather than an external hook into a function). The use of exceptions will allow greater robustness, and allow exceptional conditions to be managed in a somewhat more graceful manner, especially in mission critical or unmanned situations.

Unfortunately at this stage few (if any) compilers support exceptions, and it is unknown just what level of runtime support is required for exceptions to operate.

Better garbage collection and memory management schemes could be designed; in a non-virtual system there will always be memory management issues to deal with.

8. Conclusions.

C++ runs well in an embedded environment, and can provide greater flexibility and easier programming than standard C. Some problematic areas still exist, such as the usual assumption that memory allocation always works. Exceptions will be a very welcome addition to the environment, and will provide robust handling of errors.

C++ has been used in a number of embedded systems, and experience has shown that the advantages are manifest. Progress is being made in building an ‘industrial strength’ set of library classes that deal with the specific design and implementation issues confronting the embedded developer, such as non-volatile storage, communication line protocol handlers, telemetry I/O devices, event list handling etc.

The day may come when we look back on the days we used C and think the same thoughts that we think about Assembler now.