Efficient C/C++ Coding Techniques

Embedded Systems Conference Boston 2001

Class 304

Darren G. Moss

Research & Development

Iomega Corp.

Introduction

Modern electronics have a tall order for their firmware. Each revision of an embedded system often requires additional features, increased complexity, and a shorter development schedule than its predecessor. Beyond that, firmware usually operates under tight timing constraints and with limited ROM/RAM space. By nature, firmware is highly coupled with the hardware it drives, particularly the microcontroller.

Assembly programming is the traditional approach for such systems. Each sequence of instructions is handpicked and often results in an efficient use of space and machine cycles. Unfortunately, the assembly language has a habit of entangling the complexities of the system with its instruction set restrictions and peculiarities; thus extending the amount of time required to design and develop the firmware.

The C/C++ language allows complex features to be more clearly and quickly implemented. Its nature actively promotes structured and modular firmware development. In addition, C/C++ abstracts the firmware from its microprocessor, promising portable code (though it is still highly coupled to the devices it controls). Yet, for meeting timing constraints and for economical use of ROM and RAM, C/C++ comes up short. It is no secret that converting from assembly to C/C++ will require more ROM/RAM space and that attainable throughput (overall speed) will decrease.

Objective

The number and types of trade-offs between assembly and C/C++ are many and vary from one environment to another. The two trade-offs that are of concern in this study are the firmware ROM/RAM requirements and attainable throughput. Ideally, the firmware size and speed gap can be effectively bridged through research of C/C++ language constructs. The assembly output of the C/C++ compiler can be used to select language constructs that translate into efficient machine code. Ultimately, the size and speed disadvantage can be minimized.

Two of the biggest factors in determining the efficiency of language constructs are the compiler itself and the instruction set of the microprocessor. The possible combinations of compilers and microprocessors are virtually limitless; so the goal is to discover the types of constructs that are most likely to produce efficient firmware for any combination. In other words: portable optimizations.

Methodology

For this treatment, the Motorola M68HC11, Hitachi H8, Mitsubishi M16C, ARM, and MIPS microprocessors were selected for evaluation. Where possible, the compilers supplied by the processor manufacturers were used. As a general rule, all compiler optimizations were enabled, with size as the higher priority. In most all cases, the test C/C++ inputs are “real-life” code samples.

Analysis of the compiler output consists of the following steps:

1. Each instruction is measured for both ROM/RAM space and machine cycle usage.
2. When the same instructions are executed multiple times (i.e. in a loop), their machine cycle usage is accounted on a "per-iteration" basis.
3. The size and speed measurements for each code sample are converted into a usage percentage with 100% being the most and 1% being the least.
4. With the measurements converted into usage percentages, they can be compared across compiler/processor combinations to discover efficiency trends.

It is important to convert the ROM/RAM byte counts and execution cycle counts into a usage percentage, so that a single C/C++ construct can be compared across the different compiler/processor platforms clearly and fairly. In other words, efficiency comparisons are in terms of *twice as fast* and *half the size* instead of *53 cycles on one platform* and *34 bytes on another*. Again, the goal is to discover techniques that are efficient on most platforms.

Command Processing

Firmware systems are often a component of a larger system. Implementing the interface protocol between components can easily fall onto the firmware's shoulders. The interface or communications protocol can range from simple to complex; to name a few: SCSI, ATA/ATAPI, PCI, FireWire, and USB.

Commands and messages are usually represented as enumerated values. Those values can be dense: a hundred commands spread across a hundred values (a 1:1 ratio), or sparse: say a hundred commands spread across four hundred values (a 1:4 ratio). For this study, both dense (a ratio better than 1:4) and sparse (a ratio of 1:4 or worse) placements of values are treated.

Fortunately, the C/C++ language directly supports two constructs useful to command processing: the *if-elseif-else* and *switch-case* keywords. In addition, a type of jump-table can be built using an array of function addresses indexed to the values of the commands they process. As usual, there are trade-offs between these types of command processing.

If-elseif-else, Switch-case, and Jump-table Constructs

The basic *if-elseif-else* sequence compares the command value with all supported values. This seems simple and straightforward enough, but as the firmware rolls through revisions and additional features, simple command value compares (i.e. *if (command == 0x53)*, etc.) can too easily have obscure conditions introduced (i.e. *if (command == 0x53 && (global & 0x10))*, etc.). That should raise a few red flags – maintenance madness. An *if-elseif-else* sequence, freed from its obscure conditions, was used for both the dense and sparse command processing analysis (Listing 1).

There are not many permutations of the *switch-case* construct, although a missing *break* statement can still cause trouble. Usually, the order of the cases does not affect the generated assembly, as it does with *if-elseif-else* sequences. It is very possible that the *switch-case* construct is the simplest to create and maintain (Listing 2).

```
if (index == 62)
{
  result = Case62(param);
}
.
.
.
else if (index == 95)
{
  result = Case95(param);
}
.
.
.
else if (index == 35)
{
  result = Case35(param);
}
.
.
.
else
{
  result = -1;
}
```

Listing 1 ***If-elseif-else*** **Code Sample**

```
switch (index)
{
  case 62:
  result = Case62(param);
  break;
  .
  .
  .
  case 95:
  result = Case95(param);
  break;
  .
  .
  .
  case 35:
  result = Case35(param);
  break;
  .
  .
  .
  default:
  result = -1;
  break;

}
```

Listing 2 ***Switch-case*** **Code Sample**

Finally, the jump-table method of command processing: it probably has its roots in efficiency techniques developed in the assembly realm. In C/C++, the common method is to build an array of function addresses indexed by their corresponding command value. An array index that does not have a corresponding command is filled with the address of a function that returns an error value (Listing 3). Though the initial creation of a jump-table is fairly straightforward, adding and removing commands at their exact index locations, can be nothing less than painful.

```
typedef int (* jumpfnct)(void * param);

static int CaseError(void * param)
{

  return -1;
}

static jumpfnct const jumptable[] =
{
  CaseError, CaseError, ...
  .
  .
  .
  Case44,    CaseError, ...
  .
  .
  .
  CaseError, Case255
};

  result = index <= 0xFF ? jumptable[index](param) : -1;
```

Listing 3 Jump-table Code Sample

Analysis

As no surprise, the *if-elseif-else* sequence tends to be both larger and slower than its counterparts. Converting the code sample to fit a *switch-case* construct improves the size, if the values are dense, and increases the speed of the resultant assembly output. Finally, the code space required by jump-tables can range from nearly the biggest to the very smallest, depending on the density of the index values. It is also interesting to note that jump-tables have the fastest execution time (Chart 1).

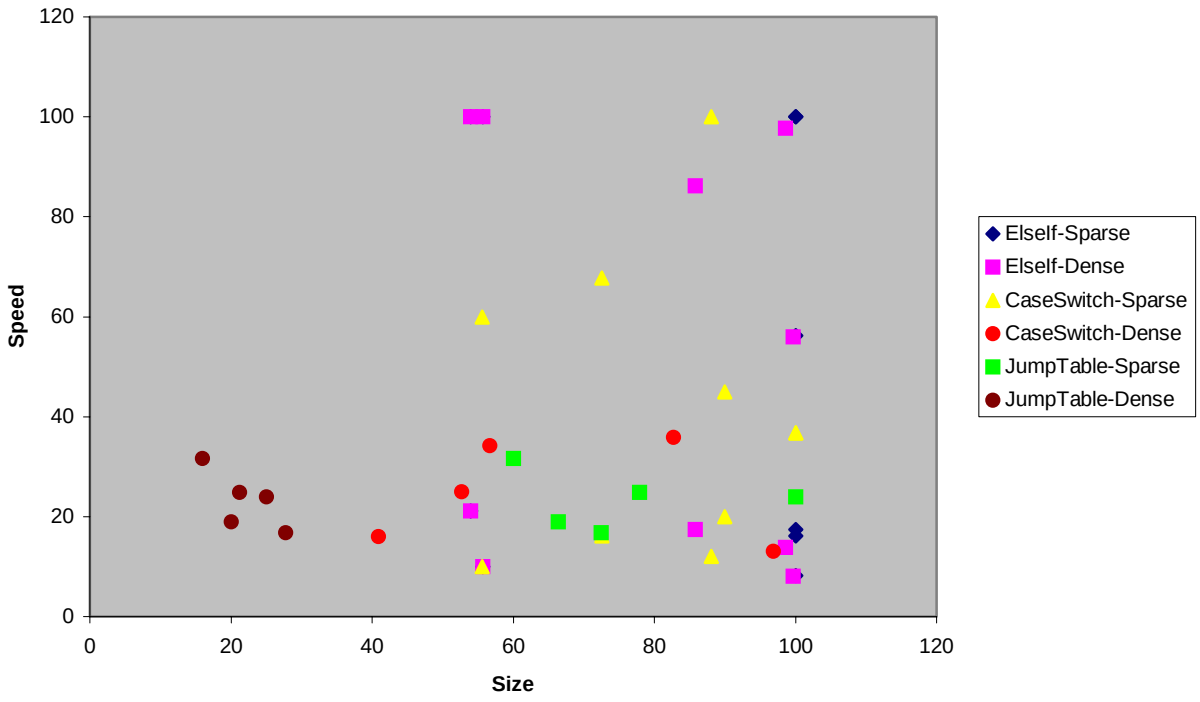

Chart 1 Command Processing Efficiency Results

All the compilers generated similar assembly code for both the sparse and dense *if-elseif*-else code samples. Usually a couple of instructions are needed to do the comparison (i.e. *else if(value == 95)*, etc.) and an additional two to five instructions for the conditional block. Because both the comparison block and the conditional block have to be duplicated for each possible command value, the *if-elseif-else* sequence is lengthy. Worse, values that have placement near the first of the sequence will be executed within a short time frame, while values have placement near the end will not be executed until all the preceding conditions are checked and found to be false. That is a crippling feature, especially when dealing with real-time and quick-response systems. The HC11 assembly output demonstrates those characteristics (Listing 4). It is interesting to note that the HC11 output is the only one that used a stack.

The *switch-case* sample is unique from the other samples due to the range of assembly output. To their credit, all compilers generated a jump-table for the dense-value sample. For the sparse-value sample, the H8 compiler produces a simple *if-elseif-else* sequence (more efficient than its C-to-assembly *if-elseif-else* translation). The HC11 compiler uses a look-up table and sequentially scanning code for its translation. The MIPS compiler converted both the sparse samples into a jump-table. Finally the ARM and M16C both generate a type of binary-search/else-if code sequence for sparse *switch-case* constructs. This is the most intriguing output because it is both smaller than a table-based method and faster than an *if-elseif-else* sequence. The ARM is the better example of the binary-search/else-if code (Listing 5). The basic idea is that the first comparison is with a median value. If result of the comparison is “equal”, then the flow branches to its conditional block. Should the result be “greater than”, the flow branches to the median value comparison of the upper half of the *switch-case* values. Otherwise (the result is

"less than"), the flow simply falls into the next comparison, which is the median value of the lower half of the *switch-case*. Eventually, a "leaf" comparison can be reached, where if the comparison fails, then the flow branches to the default case.

```
TSX           ; comparison
LDD  2,X      ; block.
CPD  #002C
BNE  NextCmp
LDD  6,X      ; conditional
BSR  Case44   ; block.
TSX
STD  0,X
BRA  Epilog
.
.
.
LDD  #FFFF    ; final "else"
TSX           ; block.
STD  0,X

Epilog:
```

Listing 4 ***If-elseif-else*** **Assembly (HC11)**

```
CMP  R0,#0x3E  ; median val.
BEQ  Case62
BGT  Median
.
.
.
CMP  R0,#0x1A  ; leaf val.
BEQ  Case26
B    Default
.
.
.
CMP  R0,#0x5F  ; last val.
BEQ  Case95

Default
MVN  R0,#0
```

Listing 5 ***Switch-case*** **Assembly (ARM)**

Although the corresponding C/C++ code does not suggest it, the assembly output for the jump-table sample (both sparse and dense values) is the simplest. All compilers have a one-to-one correlation of their output with the C/C++ input (what you design is what you get). All assembly listings first perform a range comparison, calculate the address, and then jump to it. That can be especially reassuring to all the assembly-turned-C developers. The M16C displays this correlation just as clearly as the other platforms (Listing 6). Though the size of the jump-table changes as entries are added and removed from it, the actual time to execute the calculate-and-jump does not. That is a definite boon for real-time systems everywhere. It is also important to note that, even though the jump-table is defined as an array (in C/C++), because it is declared as "const", it can be placed in system ROM.

```
MOV.W   R0,A0              ; range test.
MOV.W   5[FB],R0
CMP.B   #255,A0
JGT     Default
SHL.W   #2,A0              ; calculate
JSRI.A  jumptable:20,[A0]  ; and jump.
JMP.S   Epilog

Default:
MOV.W   #-1,R0
```

Listing 6 Jump-table Assembly (M16C)

Summary

In short, *if-elseif-else* constructs are not well suited for processing commands, especially not when there are many layers of processing in each command. In contrast, the *switch-case* is suitable for this type of work, particularly where the command values are densely packed together. Where speed is a concern (usually important to real-time systems), or ensuring that the execution time will not vary from one command to the next, nothing beats a jump-table, whether the commands are dense or sparse. If a jump-table is populated with dense values, it is the fastest and the smallest possible construct.

Iterative Processing

Quite often, firmware is found providing intermediate processing for the data that passes through it. The amount of time required to perform this data processing is most critical when there is a large quantity of data. Such large quantities of data are usually organized into arrays and iteratively processed, that is, in a loop construct.

When iteratively accessing large arrays (500 or more data items), a small change of instructions or their sequence can produce huge execution-time savings. This type of code tuning is commonly known as “loop optimization”. There are time-proven loop improvements such as: unswitching, jamming, unrolling, etc. that deserve mention, but are not treated in full here. In this study, lesser-known enhancements are given treatment, particularly: array access by index, array access by pointer, and loop counting down.

Unswitching

A loop that “switches” re-evaluates a fixed condition on iteration, for example, whether to load data from either a default or custom table (Listing 7). The term “unswitching” refers to the removal of the condition evaluation from inside the loop, placing it outside of the loop, and duplicating the loop within both of the condition cases (Listing 8). Though the loop is now duplicated, each copy is simpler than the first and therefore more efficient. The only drawback is that now both loops need to be maintained in parallel – more than just an annoyance, a maintenance headache.

```
for (i = 0; i < count; ++i)
{
  if (type == TYPEA)
  {
    sum += a[i];
  }
  else
  {
    sum += b[i];
  }
}
```

Listing 8 "Switched" Loop

```
if (type == TYPEA)
{
  for (i = 0; i < count; ++i)
  {
    sum += a[i];
  }
}

else
{
  for (i = 0; i < count; ++i)
  {
    sum += b[i];
  }
}
```

Listing 7 "Unswitched" Loops

Jamming

Jamming (a.k.a. fusion) is the combination of two or more similar loops into one. To be beneficial, though, there needs to be some degree of correlation between the data of the jammed loops (Listing 9). If there is enough correlation between the two data sets, the greatest benefit is to combine them into a data structure and then loop through an array of data structures (Listing 10). The benefit is that only one array or pointer needs to be held in a register, as opposed to two or more registers consumed by array addresses or pointers (which often leads to pushing and popping array addresses onto and off of the stack).

```
for (i = 0; i < count; ++i)
{
  type[i] = 0x10;
}

for (i = 0; i < count; ++i)
{
  size[i] = 0;
}
```

Listing 10 "Unjammed" Loops

```
struct
{
  int type;
  int size;
  .
  .
  .
} group;

for (i = 0; i < count; ++i)
{
  group[i].type = 0x10;
  group[i].size = 0;
}
```

Listing 9 "Jammed" Loop

Unrolling

Frequently, data that needs to be processed in a loop is aligned to a known byte boundary (i.e. one video (QCIF) scan line – 264 bytes, an eight-byte alignment, etc.). If the data would be correctly processed one byte at a time (Listing 11), then the loop can be "unrolled" to its byte-boundary and be processed more efficiently (Listing 12). The main advantage is that the number of iterations of the loop decreases, which, in effect, translates to more time to process data and less time lost to overhead.

```
for (i = 0; i < count; ++i)
{
  a[i] <<= 2;
}
```

Listing 12 "Rolled" Loop

```
for (i = 0; i < count; i += 2)
{
  a[i + 0] <<= 2;
  a[i + 1] <<= 2;
}
```

Listing 11 "Unrolled" Loop

Accesses by Index, Accesses by Pointer, and Counting Down

The most commonly seen use of arrays and loops has an index count up to a value as the control of the loop. That same index is then used to access data in one or more arrays (Listing 13). Occasionally, and usually for purposes other than looping through an array of data, a pointer is substituted for an array. For this treatment of pointer accesses, all the arrays are accessed via pointers (Listing 14). Finally, when pointers are used in place of arrays, it is a simple change to loop up through the data but count down for loop control (Listing 15). All these are subtle, inconspicuous differences that probably go unnoticed, but consistently improve performance.

```
for (index = 0; index < count; ++index)
{
  sum[index] = left[index] + right[index];
}
```

Listing 15 Index Access to an Array

```
for (index = 0; index < count; ++index)
{
  *sum = *left + *right;
  ++right;
  ++left;
  ++sum;
}
```

Listing 14 Pointer Access to an Array

```
for (; count > 0; --count)
{
  *sum = *left + *right;
  ++right;
  ++left;
  ++sum;
}
```

Listing 13 Loop Control by Counting Down

Analysis

Pointers prove themselves to be more efficient than arrays, except in the case of the ARM processor – which is addressed later. Also, the counting down loop control is shown to be more efficient (Chart 2). The following analysis dissects the compiler assembly output and explains these results.

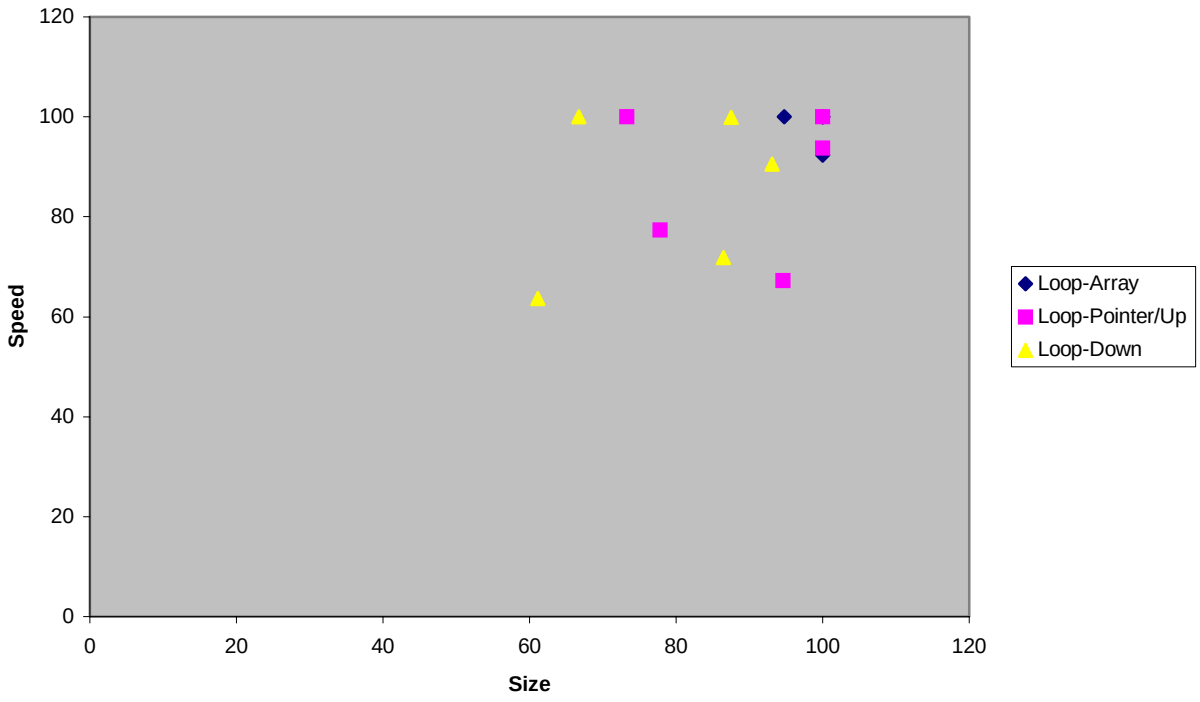

Chart 2 Iterative Processing Efficiency Results

All compiler/processor platforms, except for the ARM processor, produced similar assembly code for the array/index access type loop (Listing 16). The loop control index is converted into an array offset and then added to the array address, for each of the arrays accessed in the loop. Fortunately in this case the size of the array elements is a power of two and the index-to-offset conversion is simple. Excessive overhead will result when an element size, such as 33 bytes, is used. Such conversions (i.e. an index of 5 converted into an offset of 165) are execution-time costly because they often require a multiplication. Finally, the index is incremented and compared against the preset loop control count. It is interesting to note that the stack is used to temporarily store the address of the *sum* array.

Pointer access of arrays eliminates the index-to-offset conversion penalty by simply incrementing the pointers after accessing its data (Listing 17). In the case of an element size like 33 bytes, the pointer is simply incremented by 33 bytes, the same size and speed cost as an element size of a power of two. Even though the math involved in the accesses has been simplified by using pointers, still the stack is used – this time to temporarily store the value of the *count* variable.

```
SUB.W   R6,R6

LoopTop:
MOV.W   R6,R5    ; convert the
ADD.W   R5,R5    ; index.
MOV.W   @(SP),R0 ; calculate
ADD.W   R5,R0    ; the address.
.
.                ; repeat.
.

MOV.W   @R1,R1   ; load the
MOV.W   @R2,R2   ; array values.
ADD.W   R2,R1
MOV.W   R1,@R0   ; store the sum.

ADDS.W  #1,R6    ; increment the
CMP.W   R4,R6    ; index and loop
BCS     LoopTop
```

Listing 17 Index Access to an Array (H8)

```
SUB.W   R6,R6

LoopTop:
MOV.W   @R3,R0   ; load the
MOV.W   @R5,R1   ; array values.
ADD.W   R1,R0
MOV.W   R0,@R4   ; store the sum.

ADDS.W  #2,R5    ; increment the
ADDS.W  #2,R3    ; pointers
ADDS.W  #2,R4

ADDS.W  #1,R6    ; increment the
MOV.W   @(SP),R0 ; index and loop
CMP.W   R0,R6
BCS     LoopTop
```

Listing 16 Pointer Access to an Array (H8)

By switching to a count-down loop control, there are enough available registers to avoid using the stack (Listing 18). There is a direct correlation between the C/C++ sample code and the generated assembly. One less C/C++ variable means one less machine register is consumed. Proof that even with all optimizations turned on, the C/C++ source feed into the compiler directly influences the efficiency of the assembly output.

```
LoopTop:
MOV.W   @R4,R0   ; load the
MOV.W   @R3,R1   ; array values.
ADD.W   R1,R0
MOV.W   R0,@R5   ; store the sum.

ADDS.W  #2,R3    ; increment the
ADDS.W  #2,R4    ; pointers.
ADDS.W  #2,R5

SUBS.W  #1,R6    ; decrement the
CMP.W   R6,R6    ; count and loop
BCS     LoopTop
```

Listing 18 Loop by Counting Down (H8)

The ARM Exception

Accesses by index have a major disadvantage, the costly index-to-offset conversion. Pointer accesses also have a disadvantage (though not as potent as index accesses): their need to be incremented. The ARM instruction set provides an improvement for some index accesses and all pointer accesses (Figure 1). As long as the size of an array's data elements is a power of two, the

ARM load instruction can perform the index-to-offset conversion, combine it with the base address and load the data element. Again, should the element size be 11 or something like it, a costly multiplication will be required. The load instruction can improve a pointer's performance too by loading the pointer's data and incrementing the pointer by the size of the data, whether it is a power of two or not. Fortunately enough, the store instruction has this same capability.

Figure 1 The ARM Load Instruction

Summary

Though modern microprocessor architectures are more array-friendly, the high cost of an index-to-offset conversion is unavoidable when the data element size is not a power of two. Therefore, pointer access still cannot be beat when iteratively processing an array. Also, there is no compelling reason to use the traditional "increment the index up to the count value". The *count* variable itself can be used to control the iteration of the loop – without the extra cost of another register.

Device Input/Output

One of the benefits of C/C++ is a layer of abstraction from the microprocessor. Theoretically, firmware can be recompiled for any microprocessor without changing a single line of source code. The peculiarities and burden of stack implementation, procedure calls, instruction set restrictions, and etc. (even the machine word size is an abstraction) are conveniently removed from the engineer's shoulders and tended to by the compiler. Also in that list of abstractions is device access. In fact, if devices are not mapped into memory space, standard ANSI C/C++ provides no means for reading or writing. For memory-mapped devices, access can be gained through one of two methods: bit-field structures and bit-masking techniques.

Bit-masking and Bit-fields

C/C++ allows any location in the memory address space to be accessed through pointers. All that is required is that the pointer be given the address of the location of the device and it can be read from and written as though it were any other data object (Listing 19). Among other things, the keyword *volatile* ensures that each C/C++ access directly corresponds to an access in assembly. Devices can be read, bit-masked with a desired value, and written back – much like assembly programming.

Use of a pointer to access devices incurs two types of overhead that can be seen from the C/C++ source code. First a pointer requires storage space to hold the address of the device. Second, before accessing the device, the pointer must be dereferenced and then reads and writes can be performed. Both of these penalties can be avoided by using a macro that contains the device address (Listing 20). Such a macro more closely shadows the assembly load/store sequence.

```
int volatile * const device = DEVICE_ADDRESS;
.
.
.
*device = (*device & ~0x8 & ~0x70) | 0x1 | 0x6 | 0x80;
```

Listing 20 Device I/O by Pointer

```
#define device (*(int volatile * const)DEVICE_ADDRESS)
.
.
.
device = (device & ~0x8 & ~0x70) | 0x1 | 0x6 | 0x80;
```

Listing 19 Device I/O by Macro

Finally, the C/C++ bit-field construct allows any bit or group of bits anywhere to be individually read or written - or so it seems (Listing 21). Bit-fields are fraught with hidden costs, which will be discovered when the assembly output is examined.

```
struct BitfieldStruct
{
  unsigned bit0   : 1;
  unsigned bits12 : 2;
  unsigned bit3   : 1;
  unsigned bits46 : 3;
  unsigned bit7   : 1;
};
.
.
.
device.bit3   = 0;
device.bits46 = 0;
device.bit0   = 1;
device.bits12 = 3;
device.bit7   = 1;
```

Listing 21 Device I/O by Bit-field Structure

Analysis

The efficiency comparison results confirm the efficiency of macros over pointers and also confirm that bit-field structures have hidden costs for they are the largest and slowest of the bunch (Chart 3).

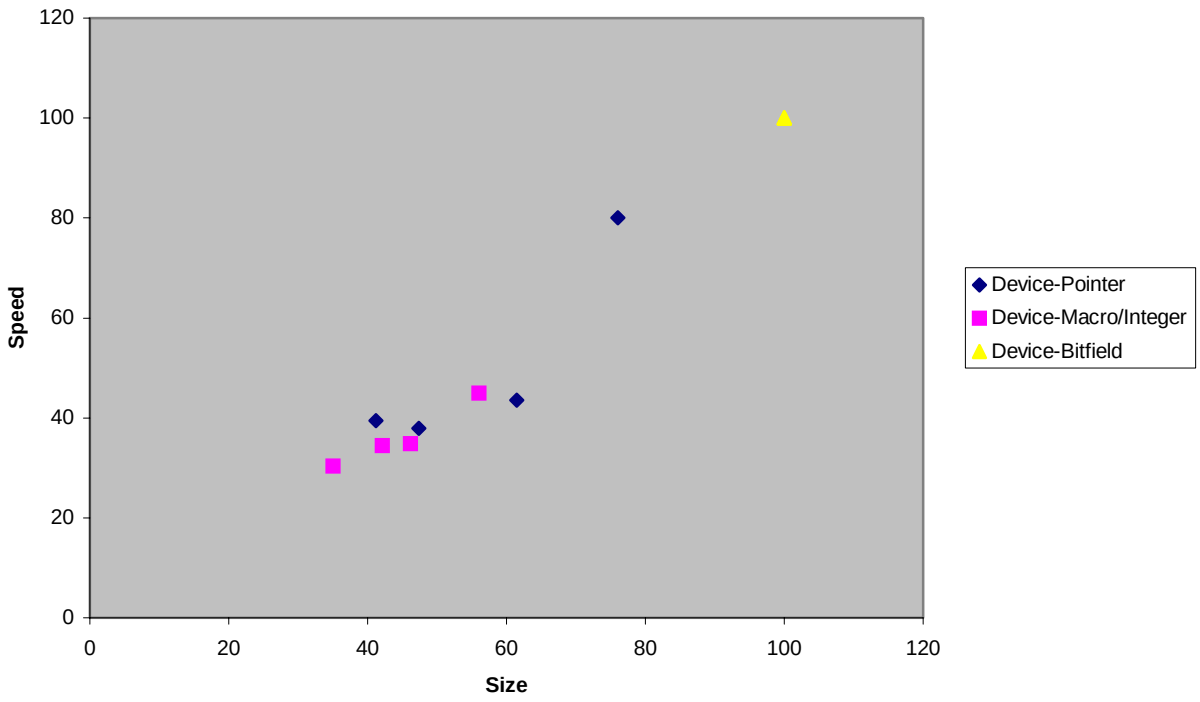

Chart 3 Device Input/Output Efficiency Results

As predicted, the pointer requires space for storing the address of the device. Again, it requires an instruction to load the device address from the pointer and then can read and write the device at will. Besides that, the assembly code generated by the compilers looks very much like code that would be expected of an assembly programmer (Listing 22).

If the pointer method has the appearance of being handmade by an assembly programmer, the macro method is exactly what should be expected. By using a macro in the C/C++ source code, the device address was basiclly placed “inline” the instruction stream. This output definitely demonstrates the efficiency that can be expected from a modern compiler (Listing 23).

```
ADR  R0,device   ; load the ptr.
LDR  R0,[R0,#0]  ; load its val.
LDR  R1,[R0,#0]  ; read the dev.
BIC  R1,R1,#0x78
ORR  R1,R1,#0x87
STR  R1,[R0,#0]  ; write the dev.
```

Listing 23 Device I/O by Pointer (ARM)

```
MOV.W  #65415,R0 ; read the dev.
AND.W  65280,R0
OR.W   #135,R0
MOV.W  R0,65280  ; write the dev.
```

Listing 22 Device I/O by Macro (M16C)

The assembly generated for the bit-field structure is undoubtedly the largest and slowest. These are the types of constructs that make C/C++ infamous for code-bloat. It is twice as large and more than twice as slow as its peers. From inspection of the generated assembly, it is clear that the C/C++ compiler generates a load/modify/store sequence for every access of a bit group in the structure (Listing 24). It would be hard to be more inefficient.

```
BCLR.B  #4,@65280:8  ; clear a single bit.

MOV.B   @65280:8,R0L ; clear/set a bit group.
AND.B   #-15:8,R0L
MOV.B   R0L,@65280:8
.
.                    ; repeat for other bits.
.

BSET.B  #0,@65280:8  ; set a single bit.
```

Listing 24 Device I/O by Bit-field (H8)

Summary

Pointers require extra storage space and bit-fields are simply too inefficient to even be considered. Macros are definitely the method of choice. The assembly generated for it is exactly what should is to be expected, concise and fast. The corresponding C/C++ macro definition can be confusing. But if all devices are accessed in that manner, then a standard macro format can eliminate the potential for mishap.

Structure Inheritance

Just as most new firmware was once done in assembly and is now done in C, it appears that C++ is being poised to follow suit. An embedded C++ standard is emerging to fit C++ to the embedded/firmware world. Therefore, now is the time to investigate C++ and discover its strengths and weaknesses.

Inheritance is a C++ construct that is not truly available in C; therefore it should be investigated and compared with nearest match in the C language. Conveniently, the C++ standard treats

structures as a type of an object class in which the default scope of its members is public. Therefore, while focusing on inheritance, this study uses structure constructs to generate the assembly listings. What is discovered for structure inheritance will be the same for class inheritance.

Inheritance: C Style, C++ Style, and Virtual

When, in C, the combination of two of more data is logical, a structure is formed. Should one or more the data that will be included into the structure is itself a structure, there is only one-way to include it: make it a member of a new data structure (Listing 25 and Figure 2). The data in the structure is organized in memory in the same order that it is listed in the C source code. Sounds simple enough, and it is.

```
struct BlockStruct
{
  int left;
  int right;
  int top;
  int bottom;
};

struct ColorStruct
{
  unsigned char hue;
  unsigned char saturation;
  unsigned char luminance;
};

struct ListStruct
{
  struct ListStruct * next;
  struct BlockStruct  block;
  struct ColorStruct  color;
  int                 zorder;
};
```

Listing 25 Inheritance: C Style

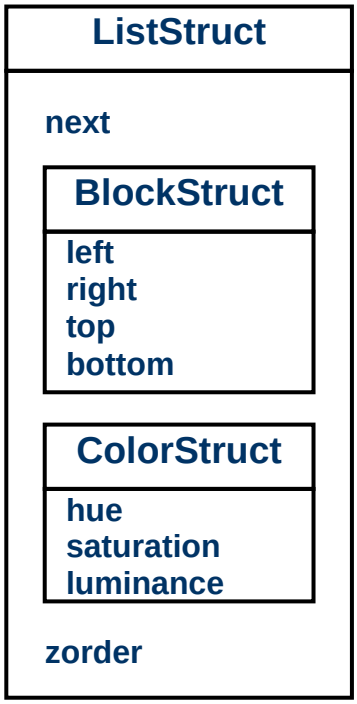

Figure 2 Inheritance: C Style

If it is logical to combine two or more data together and it makes sense to view the combination of that data as base type – derived type relationship, then C++ style inheritance seems to be the proper answer (Listing 26 and Figure 3). Inheritance allows a new type of inclusion or exclusion: some of the inherited data, that should not be used be the derived structure, can be hidden away from it via the *private* keyword. Now the prickly-part: the programmer cannot determine the organization of the inherited data. Another layer of abstraction – it just might be getting too thick.

```
struct BlockStruct
{
  int left;
  int right;
  int top;
  int bottom;
};

struct ColorStruct
{
  unsigned char hue;
  unsigned char saturation;
  unsigned char luminance;
};

struct ListStruct :
BlockStruct, ColorStruct
{
  ListStruct * next;
  int          zorder;
};
```

Listing 26 Inheritance: C++ Style

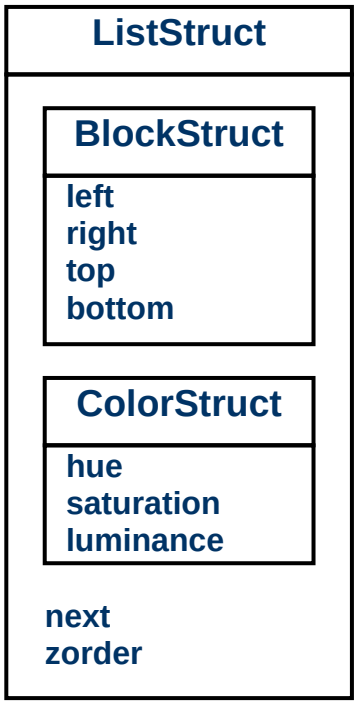

Figure 3 Inheritance: C++ Style

Virtual inheritance is pretty much like black magic. When two classes are derived (by normal methods) from the same base class and are inherited by a third class, two copies of the original base class will exist. If those same two classes are “virtually” derived from the same base class, then when inherited by the third class, only one copy of the original base class will exist (Listing 27 and Figure 4). The trick to virtual inheritance is that a no classes actually contain the virtual base class, but instead hold a pointer to a virtual base class table that indicates where in the object the virtual base class exists. This is important because multiple classes can inherit those same derived classes and therefore the location of the virtual base class will change from class to class. Complicated? Yes, it is.

```
struct BlockStruct
{
  int left;
  int right;
  int top;
  int bottom;
};

struct ColorStruct :
virtual BlockStruct
{
  unsigned char hue;
  unsigned char saturation;
  unsigned char luminance;
};

struct ListStruct :
virtual BlockStruct
{
  ListStruct * next;
};

struct ColorListStruct :
ColorStruct, ListStruct
{
  int zorder;
};
```

Listing 27 Inheritance: Virtual

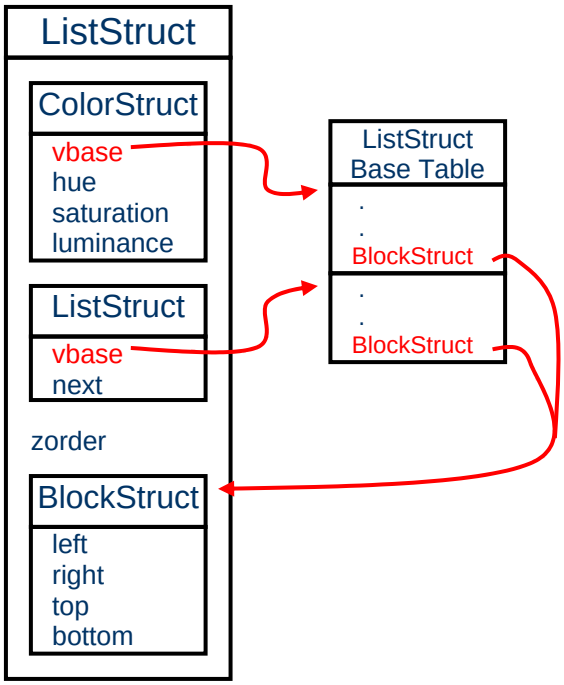

Figure 4 Inheritance: Virtual

The complications add upon themselves as the levels of virtual base class inheritance increases. Each level of virtual inheritance requires its own virtual base class table. To traverse an object into its deepest virtual base class – say there are two levels of virtual inheritance, two separate virtual base class table lookups are required (Listing 28 and Figure 5). Code-bloat? Slow? Yes and yes.

```
struct BaseStruct1
{
  .
  .
};

struct BaseStruct2 :
virtual BaseStruct1
{
  .
  .
};

struct BaseStruct3 :
virtual BaseStruct2
{
  .
  .
};

struct DerivedStruct :
public BaseStruct3
{
  .
  .
};
```

Listing 28 Multiple Virtual Inheritance

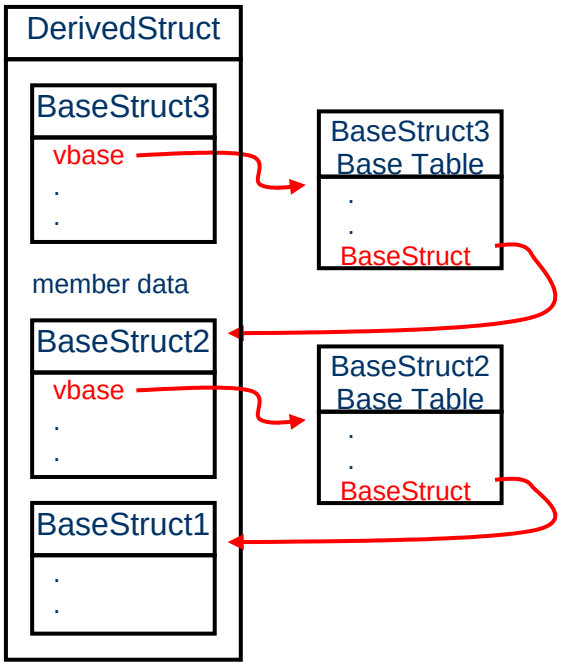

Figure 5 Multiple Virtual Inheritance

Analysis

As can be predicted from the corresponding C/C++ source code, C structures and simple C++ inheritance are nearly identical in the terms of size and speed. And as has been discussed, virtual inheritance is the most inefficient form of inheritance (Chart 4).

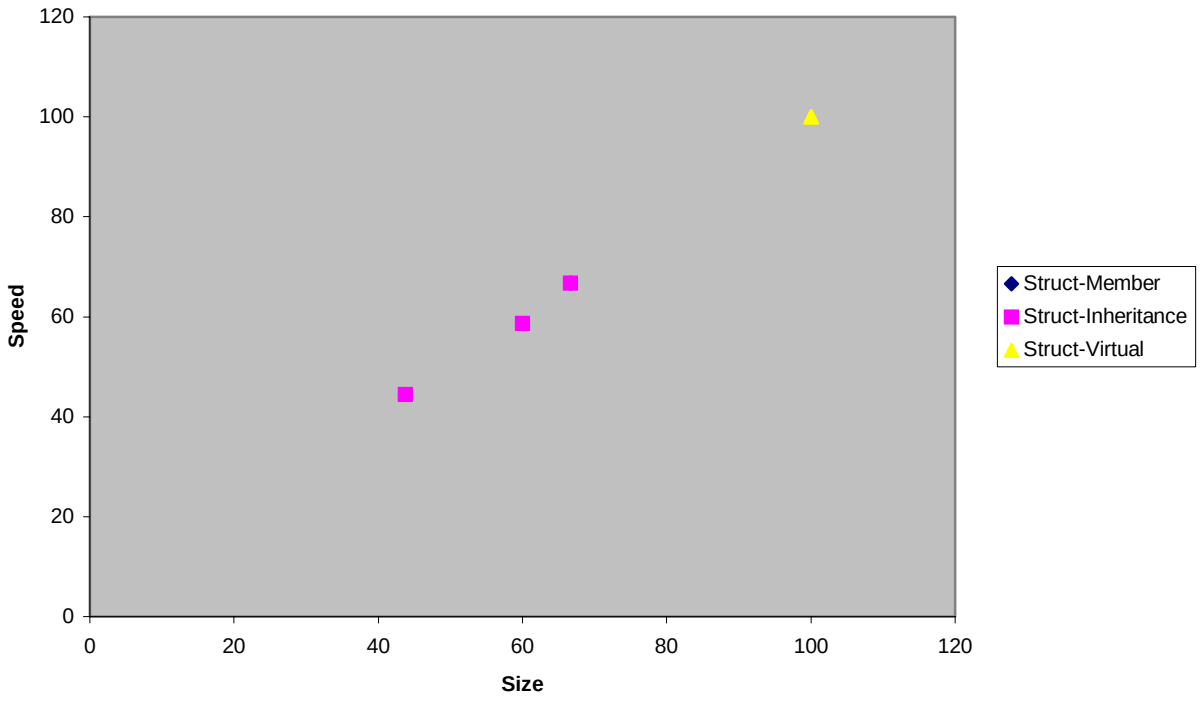

Chart 4 Structure Inheritance Efficiency Results

Both the C style inheritance (by membership) and the simple C++ inheritance are straightforward enough to allow for highly efficient access of their member data, no matter how deeply buried it may be (Listings 28 and 30). Concerns of efficiency cannot dictate the choice of one over the other; instead terms of readability and maintainability enter in at this point.

```
MOV.W  @(4:16,R0),R0  ; fixed-offset
                      ; member access.
```

Listing 30 Inheritance: C Style (H8)

```
LDR  R0,[R0,#4]  ; fixed-offset
                 ; member access.
```

Listing 29 Inheritance: C++ Style (ARM)

Virtual inheritance is a beast when it comes to efficiency. Slower execution time and more ROM space is consumed by data accesses; and the deeper the data's buried, the harder it is to get to it. If the data is only buried one level deep, then the pointer to the virtual table is loaded, the entry for the desired virtual class is accessed, and the address of the base class is computed. After all of that is done, then the data that was originally desired can be accessed (Listing 31). Definitely not a pretty sight – firmware will do well to avoid virtual inheritance.

```
LW    $9,0($4)   ; vbase pointer.
LW    $10,4($9)  ; base class offset.
ADDU  $11,$4,$10 ; calc. class pointer
LW    $12,0($11) ; load desired data.
```

Listing 31 Inheritance: Virtual (MIPS)

Summary

The choice between C style inheritance (by membership) and simple C++ inheritance is not one of efficiency – the same type of assembly is produced for both. Other factors, such as the benefit of private, protected, and public data scope, must govern that decision. Virtual inheritance incurs too much overhead and is not suitable for current firmware systems. If C++ is not used anywhere in a project, then the possibility of a virtual base class "sneaking" into the code base is eliminated.

Object Constructs

Another C++ concept is the organization of data and functions into objects. By and large, object-oriented design and development of software is the buzzword of the day. Although the "data plus functions" or "properties and methods" concept can be expressed in C as data structures and its related functions, it is undoubtable that firmware will eventually employ the C++ style of objects.

Objects: C Style, C++ Style, and Virtual

As mentioned above, grouping a data structure with the functions that access, control, and modify it is a simple type of object-oriented design and can be done in C (Listing 32 and Figure 6). How the full-blown C++ language compares with this simple type of object-oriented programming is what should be determined.

```
struct DataStruct
{
  struct DataStruct * parent;
  struct DataStruct * leftchild;
  struct DataStruct * rightchild;
};

struct DataStruct *
DataChildLeftmost(struct DataStruct * node);

unsigned int
DataParentCount(struct DataStruct * node);
```

Listing 32 Objects: C Style

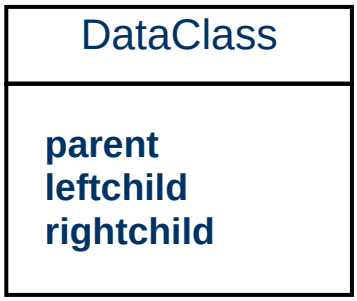

Figure 6 Objects: C Style

A very common C++ object style is to make the member functions public and the data private (Listing 33 and Figure 7). It is important to note that most class functions have one hidden parameter, the *this* pointer. The *this* pointer contains the address of the object that the function is to modify. Whenever member data is accessed, it is done so by dereferencing the *this* pointer (i.e. this->data). This simple C++ class is hardly different from the C structure and the two should produce similar assembly listings.

```
class DataClass
{

  public:
  DataClass *  ChildLeftmost(void);
  unsigned int ParentCount(void);

  private:
  DataClass * parent;
  DataClass * leftchild;
  DataClass * rightchild;

};
```

Listing 33 Objects: C++ Style

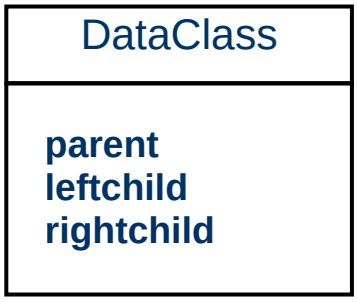

Figure 7 Objects: C++ Style

Another common object-oriented feature is the virtual function. Rare is the C++ object that does not contain at least one virtual function (Listing 34 and Figure 8). As with virtual inheritance, there are hidden virtual function tables (a.k.a. hidden costs) that make virtual functions possible.

```
class DataClass
{

  public:
  virtual DataClass *  ChildLeftmost(void);
  virtual unsigned int ParentCount(void);

  private:
  DataClass * parent;
  DataClass * leftchild;
  DataClass * rightchild;

};
```

Listing 34 Objects: Virtual Functions

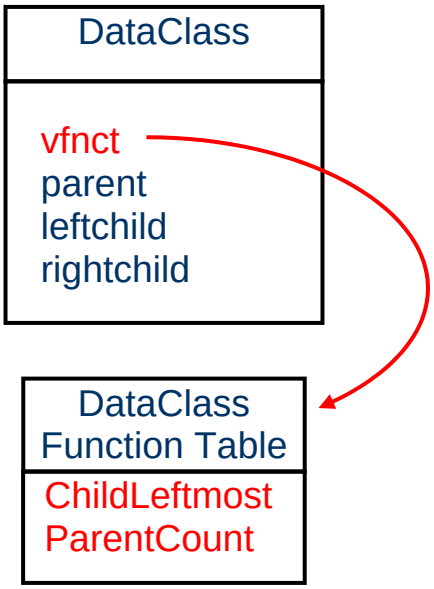

Figure 8 Objects: Virtual Functions

Can virtual functions be as complicated as virtual tables? Yes. Multiple inheritance requires multiple virtual function tables (Listing 35 and Figure 9). The reason is that an outside caller may only have a pointer to one of the base objects and not even know what type of class the derived object actually is. So there must be a virtual function table with the scope of the base class so that the proper virtual function can be executed.

Most member functions require that the *this* pointer be passed in to them as a hidden parameter. If a virtual function, which expects a pointer to the derived object, is called from a base class pointer, that pointer must be adjusted to point to the derived class. Conversely, if a virtual function, which expects a pointer to the base class, is called from a derived class pointer, that pointer must be adjusted to point to the base class. This conversion is usually referred to as a logical *this* adjustment or is performed by an adjuster thunk/veneer depending on the particular implementation. Such adjustments are only required for virtual functions and multiple inheritance.

```
class BaseClass1
{
  virtual VFnct1();
  .
  .
};

class BaseClass2
{
  .
  .
};

class DerivedClass :
public BaseClass2,
public BaseClass1
{
  virtual VFnct2();
  .
  .
};
```

Listing 35 Multiple Inheritance and Virtual Functions

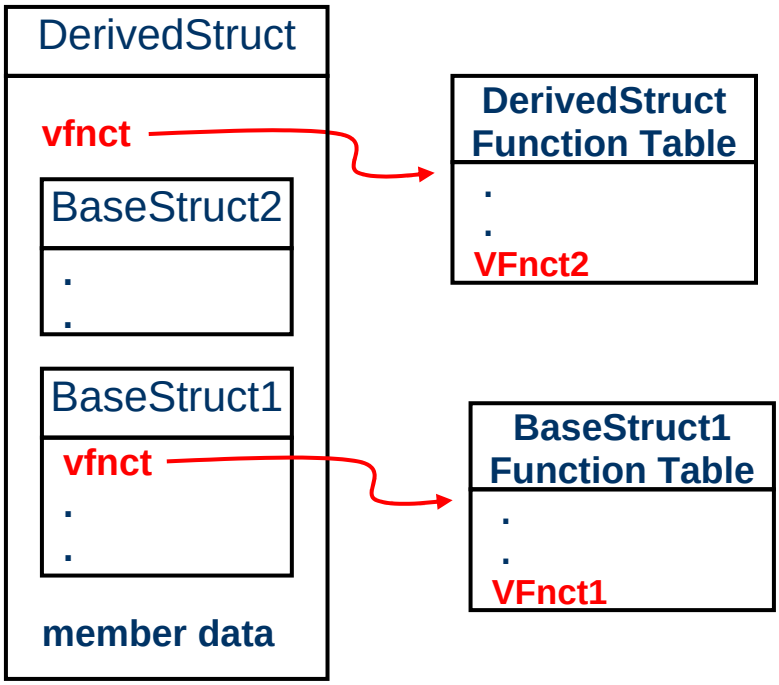

Figure 9 Multiple Inheritance and Virtual Functions

The object-oriented stew is stirred-up a bit more when both virtual functions and virtual inheritance enters into the pot (Listing 36 and Figure 10). Not only does a function call have to pass through the virtual function table look-up and a possible adjuster thunk, but it could also pass through one or more levels of virtual inheritance. The amount of possible overhead is high and costly and, to say the least, should be avoided.

```
class BaseClass1
{
  virtual VFnct1();
  .
  .
};

class DerivedClass :
virtual BaseClass1
{
  virtual VFnct2();
  .
  .
};
```

Listing 36 Virtual Inheritance and Virtual Functions

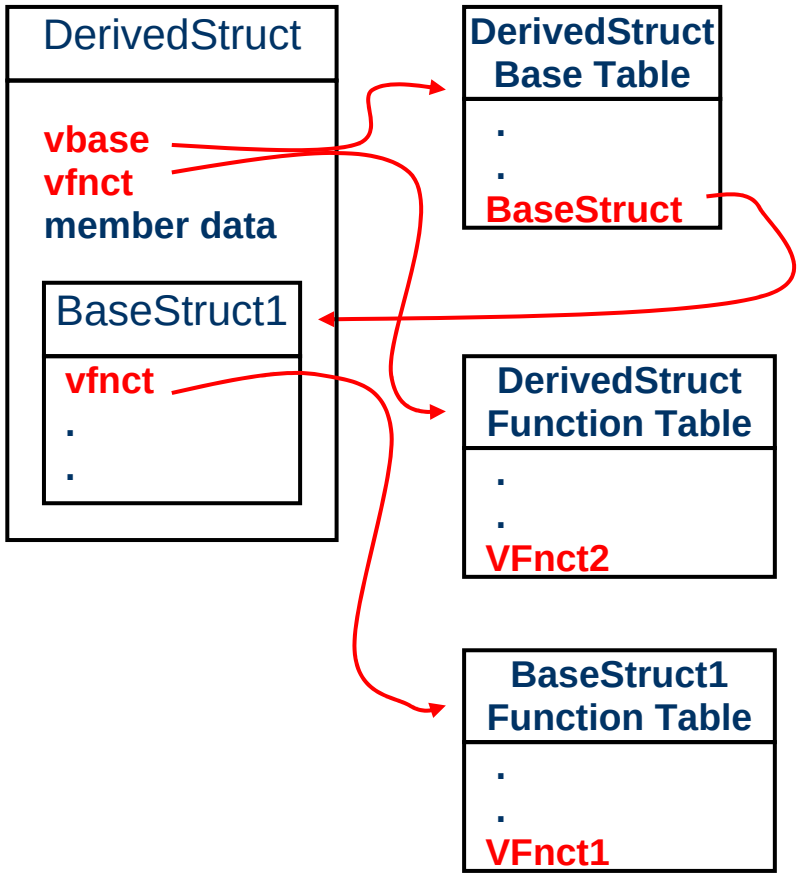

Figure 10 Virtual Inheritance and Virtual Functions

Analysis

Once again, the C style object organization and the simple C++ class produce equivalent assembly listings. The real “dogs” are the virtual functions (Chart 5).

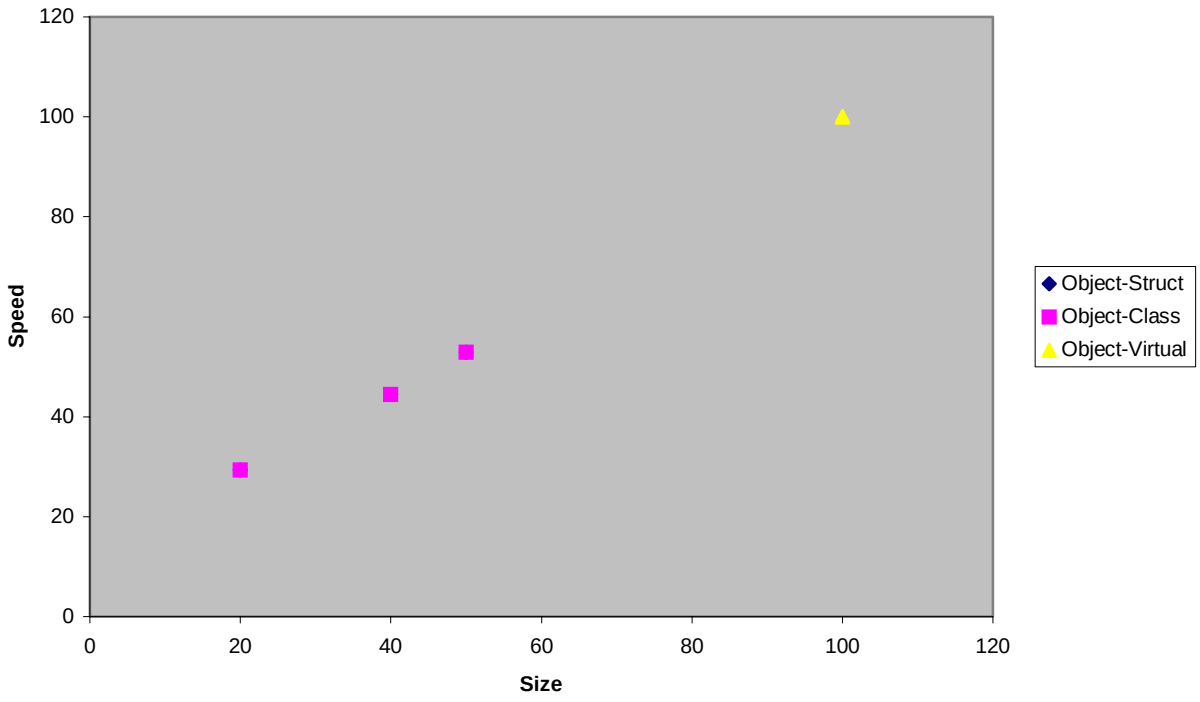

Chart 5 Object Constructs Efficiency Results

Calling either a regular C function or a C++ class function is equivalently the same as far as the assembly language is concerned. Simply pass the parameters onto the stack and call the function (Listings 37 and 38). Once again, terms of efficiency are not involved in selected between these two styles.

```
BSR  DataParentCount
```

Listing 38 Objects: C Style (H8)

```
BL  ParentCount_DataClass
```

Listing 37 Objects: C++ Style (ARM)

Virtual functions prove themselves to be beasts just like their cousin, virtual inheritance. Virtual function calls require twice as much code and execute less then half as fast. The virtual function table must first be accessed, then the particular function is looked-up, and finally the function is called (Listing 39). The whole concept of virtual functions is quite interesting, but cost more than they're worth to firmware.

```
LW   $8,0($25)  ; vfnct pointer.
LW   $9,0($8)   ; function address.
JAL  $24        ; call the vfnct.
```

Listing 39 Objects: Virtual Functions (MIPS)

Summary

Again, efficiency is not an issue when dealing with C style objects and simple C++ object classes. Virtual functions are the real issue. If C++ is avoided throughout a project's code base the inefficiencies of virtual functions will not rear their ugly head.

Conclusion

The objective of this entire treatment is to determine if the speed and size disadvantages of C/C++ can be minimized for a range of compiler/microprocessor platforms. This study resoundly says: yes. The assembly output of common C/C++ constructs demonstrate that the correct selection of coding techniques does guide the compiler to produce efficient code. By reviewing the output of questionable C/C++ constructs not covered by this study will give the engineer an intimate understanding of how his or her techniques affect efficiency.