

■ Optimal C Constructs for 8051 Microcontrollers

Nigel Jones ■

The limitations of an 8-bit microcontroller (MCU) can sometimes make conventional C constructs produce suboptimal code. In this article we look at common problems on the 8051 family and discuss workarounds in C.

The 8-bit MCU marketplace is massive. And, despite the inroads made by newer architectures, the ancient 8051 continues to hold significant market share. But old 8-bit architectures, such as the 8051 and HC11, were never designed to be programmed in a high-level language such as C.

Compiler vendors work hard to make good compilers for these 8-bit microcontrollers. But even with all the ingenuity in the world, compilers must eventually bump up against a target MCU's fundamental limitations, producing some surprising results.

Those of you familiar with the 8051 know that it's not exactly a great architecture from a programmer's perspective. Accordingly, a lot of 8051-based projects end up hungry for every CPU cycle and memory byte that can be saved. Since I enjoy working on 8-bit designs, I've spent many years wringing the last drop of performance out of such systems—and learning a lot in the process.

If you have a poor software design, inefficient algorithms, or haven't taken the basic steps advocated by the compiler vendor, nothing I can tell you here will make a difference. However, if you take all the steps advocated by the compiler vendor and have optimal algorithms and other necessary conditions, and you still need more performance (but don't want to resort to assembly language) finding the optimal C coding constructs can help you get the job done.

On 32-bit systems with a plethora of registers and addressing modes, functionally equivalent coding constructs will probably produce identical performance. On 8-bit systems, however, a subtle change in coding style can affect performance significantly. Consider the following blocks of code, which are functionally equivalent.

```
// 8 bytes and 42 cycles
for (i = 0; i < 10; i++)
{
    _nop_();
}

// 15 bytes and 120 cycles
i = 0;
while (i++ < 10)
{
    _nop_();
}
```

As the comments indicate, the difference between these two fragments is significant. Given the large difference, you'd think the compiler vendors would publish a nice list of optimal techniques to which one could refer. Well, I have poured over dozens of compiler manuals and have never come across such a list. I can only assume that either the vendors aren't aware of these differences (which seems hard to believe), or they consider it a sign of weakness in their product, and don't want to publicize the information.

Preliminaries

Working on the adage that “What they won’t tell me is probably really worth knowing,” I have created just such a list of optimal C constructs. This list was hard won and I intend to share it with you. However, a few caveats are in order.

First, this list is only definitive for an 8051 and version 6.02 of the Keil compiler using optimization level 8, favoring execution speed over size. I also used the no integer promote (NOIP) switch. Does this mean that if you are using something else that this article is a waste of your time? I hope not! Many of the results come about because of the 8051 architecture. Thus, if you are using another 8051 compiler, I suspect that the general results will hold true. If you are using another 8-bit machine, such as the HC11 or the PIC then at the very least, the article should give you some food for thought.

Second, I can’t hope to cover every possible permutation of each of the scenarios. If you think I’ve missed an important variant, please add it to the test code and let me know your results.

“Optimal” can mean many things. I think there is a tremendous case for claiming that optimal code is code that is easy to maintain. In this article, however, I use optimal to refer to code that is optimally fast, small, and consistent in its execution time; I err on the side of speed. Generally, however, I’ve found that the fastest code is also the smallest.

On many occasions I have seen compilers do weird things with subtle changes to code. As a result, the information I’ll present here should be taken as a “rule of thumb.” If you apply the techniques, most of the time you’ll get more optimal code. However, if you are trying to wring the last ounce of performance out of your system, you should really check the results for your specific case.

Finally, some of the tests reveal surprising weaknesses in the Keil compiler. It is not my intention to criticize their product. Indeed, I have used their compiler for over ten years, which says a lot about what I think of it. I also suspect that similar compilers would also show their idiosyncrasies when subjected to similar detailed inspection. I’d like to go on the record as saying I think Keil’s products are excellent—and put a lot of the bigger names in the industry to shame.

Coding conventions

A central problem in trying to demonstrate the various constructs is to keep the code very simple—but not so simple that the optimizer realizes the code is doing nothing useful, and hence optimizes it away to nothing. To this end, I make extensive use of the intrinsic “function call” `_nop()`.

In comparing various techniques, and to ensure that every technique gets a fair hearing, I make extensive use of conditional compilation. To try the various techniques, it is of course necessary to set the value of the various manifest constants in order to compile the code one way or another.

In most cases, I report the number of execution cycles, the code size, and the data size of the various fragments. The data size is instructive. However, given that this memory gets heavily overlaid on an 8051, don’t get too worked up about differences in this parameter. When relevant, I also report the size of the library code.

In some of the projects, I run a lot of variants in an attempt to deduce patterns and optimal constructs. I usually only report on my conclusions. I encourage you to study the results of the various test cases and see if you agree with my conclusions.

As a final note, the execution times are normally given in cycles assuming a standard 8051 (12 clocks per instruction cycle). When I am looking at the benefits of dual data pointers, I use a processor with a different number of clocks per cycle. In either case, what is really important is the ratio (or difference) of the execution times.

Code constructs

We’ll be looking at these areas:

- Looping
- Function calling

- Decision making
- Calling one of N functions
- Pointer manipulation
- Copying
- Bit-banging

The next to last section, on copying, builds on the earlier topics, and the bit-banging section builds on everything. That is, we'll apply some of the techniques to address the problem of addressing a serial device by twiddling port lines.

Looping

All nontrivial programs contain loops. Of these, the most common is the loop that has to execute a fixed number of times. C offers numerous ways to code this. To see if there is any significant difference between the various constructs, I put together the looping project. The looping project contains eight different constructs. After testing the eight variants, I achieved a variation in performance that is quite spectacular (2.5:1 in size and 3.5:1 in speed). Furthermore, the best performing construct is not exactly intuitive.

First, let's look at the worst performing loop construct:

```
// Code: 15. Cycles: 124.
void fne(void)
{
    unsigned char i;

    i = 0;
    while (i++ < 10)
    {
        _nop_();
    }
}
```

This is a fairly innocuous construct, but for various reasons it causes the compiler to generate a lousy solution. In contrast, three different constructs produced the optimal solution. The following is the most "standard" looking:

```
// Code: 6. Cycles: 35.
void fnc(void)
{
    unsigned char i;

    for (i = 10; i; i--)
    {
        _nop_();
    }
}
```

Note that this solution (along with the other two optimal solutions) involves counting down. In contrast, let's look at the results for a standard count-up for-loop: // Code: 8. Cycles: 46. void fna(void) { unsigned char i; for (i = 0; i < 10; i++) { _nop_(); } }

The above requires two more code bytes and an extra cycle per loop iteration. This is an interesting result, one in which the standard construct doesn't produce the optimal code. This result comes about because the 8051 has a DJNZ (Decrement and Jump if Not Zero) instruction that is designed for efficient looping. However, it requires the loop variable to count down—hence the for-loop that counts down gives the best performance.

Note, however, that the for-loop that counts down is usually problematic if the loop variable is used within the body of the loop. For example, if we wanted to set the elements of an array to some value, we'd have a problem because the values of the

looping variable would be 10 to 1 instead of 9 to 0. If you need to use the looping variable within the body of the loop (which is the more common case), then the standard count-up for-loop construct gives the best results.

It's also worth noting that if you were to use a 16-bit variable as the loop index, the optimal construct would probably change. Since I almost never have to use a 16-bit variable as a loop index, I've decided to leave this case as an exercise for the reader. If anyone does try it, I'd be interested in the results.

Function calling

Calling a function incurs overhead. On modern processors, this typically consists of pushing the required parameters onto a stack and then executing a call. The called function pops the required parameters off of the stack, performs the required work, and then returns. For many of the modern processors, the penalty is not so much the pushing and popping of parameters, as it is the fact that a function call often results in a cache miss and the attendant system slow down.

In the 8051 world, however, parameters are usually passed via registers. (It's horribly inefficient to use the compiler's reentrant model with parameters pushed onto an external stack. Consequently, it should never be used except in a case of true need.) However, the 8051 only has a few registers to use to pass parameters. Keil's rules for parameter passing appear in Table 1 (at <ftp://ftp.embedded.com/pub/11jones/tables>).

There are several things to note:

- At most, only three parameters will be passed in registers
- The order in which arguments are declared affects how many bytes are passed in registers
- At most, only one long or float and only one generic pointer can be passed in registers
- There is no provision for passing other data types such as a structure or a union in registers

Let's look at the implications of each of these limitations.

Three parameter limit

Passing lots of parameters to a function on an 8-bit machine is not a great idea. Here are some things you can try to get around this limitation:

- Group data into structures and pass a pointer to the structure
- If you need to pass four chars, you can group them as a union of four chars and one long and then pass the long (not pretty, but it works)

Parameter ordering

Here's a limitation that often surprises people. Consider a function `foo()` that requires a char, an int, and a generic pointer to be passed to it.

```
void foo(char a, int b, int *c);
```

This will result in all six bytes (one for a, two for b, and three for c) being passed in registers. However, rearranging the parameter order to:

```
void bar(int *c, int b, char a);
```

will result in only five bytes being passed in registers, with parameter a being passed in a fixed memory location. This results in a significant performance hit.

The code to demonstrate this is in Listing 1 (at <ftp://ftp.embedded.com/pub/11jones/listings>), which displays Test Case 1 of the function calling project. The first method requires three fewer cycles, three fewer code bytes, and six fewer data bytes than the second method. If this function is called often, the overall savings can be significant.

The bottom line is this: if you have a function that is called repeatedly, it is worthwhile to check the parameter order to make sure as many parameters as possible are passed in registers.

Passing pointers

Table 1 shows that only one generic pointer can be passed in registers, whereas up to three memory-space-specific pointers may be passed in registers. This is an unsubtle hint not to use generic pointers. To give you an indication of how severe the penalty is, consider Listing 2, which shows the difference between passing two generic pointers and passing two typed pointers.

The casts in `foo()` are intended to eliminate the time difference associated with the expression evaluation (the Keil compiler is smart enough to eliminate the overhead associated with the casts). Calling `foo()` required considerably more resources than calling `bar()`. This can all be attributed to the fact that `bar()` required more bytes (six versus four) to be passed, and that three of the bytes were passed in fixed memory locations. The obvious conclusion is to look very hard at any function that requires the passing of more than one generic pointer.

Passing other data types

The register argument rules are quite rigid. If you have the temerity to try and pass a bit or a structure as an argument (even if the structure consumes just two bytes), that parameter and all parameters to the right of it on the argument list are not passed in registers. Thus:

```
void foo(char a, int b, int *c, bit d);
```

will result in the first three parameters being passed in registers. However,

```
void bar(bit d, char a, int b, int *c);
```

will result in none of the parameters being passed in registers. The code to demonstrate this is Test Case 3 of the function calling project. The performance difference is significant and is shown in the Table 2.

Similarly, the following code:

```
typedef struct
{
    char e;
    char f;
} STYPE;

void foo(STYPE a, int b);
```

also results in none of the parameters being passed in registers, since `STYPE` defines an unsupported type. The code to demonstrate this is in Listing 3.

In this case, `foo()` uses a massive amount of resources compared to `bar()`. Now admittedly, this has a lot to do with the compiler calling a function to copy the structure—rather than being smart enough to realize the structure is so small that it could easily fit into the available registers. However, I think it demonstrates that passing structures to a function is a really bad idea.

On a related note, I have also discovered that in certain circumstances, even when all the parameters may be passed in registers, changing the ordering of parameters sometimes makes a difference. Unfortunately, I am unable to come up with a rule of thumb to guide you on this one. All I can do is suggest you experiment on critical functions.

Return types

No discussion of function calling would be complete without mention of the impact of the data type that is returned. The Keil documentation shows that all basic types (`char`, `int`, `float`, `*`) will be returned in registers. A bit will be returned in the Carry flag. However, they make no mention of what happens if you try and return a structure. We'll examine the case where you wish to get a copy of the data structure. Listing 4 examines this topic.

Thus, the simpler construct produced marginally better code. This is the first case of many where simplicity produces the better results.

As an aside, this example again demonstrates the inefficiency of the compiler's data structure copy routine. For a trivial case like this, it would of course be better to return a pointer and perform an explicit element-by-element copy. I just find it surprising that the compiler doesn't contain some form of heuristic algorithm for determining the optimal solution.

Function calling summary

Take a hard look at all of your function prototypes. The following changes will usually result in worthwhile performance increases:

1. Convert generic pointer arguments to typed pointer arguments
2. Make sure that any bit or structure arguments are passed last
3. Notwithstanding #2, don't pass structures!
4. Minimize the number of arguments-but not at the expense of violating rule 3
5. Look at the ordering of your parameters to see if a reordering will result in all (or more) of the parameters being passed in registers
6. Experiment! If you have a function that is called regularly, experiment with the order of its parameters. You may be surprised at the gains you realize.
7. If you need to assign a structure from a function call, return the structure instead of a pointer to the structure.

Decision making

All programs are littered with if-else and switch-case constructs. On the face of it, there isn't much one can do to change the system performance. However, certain questions that should be answered include:

- Is the ternary operator more efficient than an if-else?
- What's the most efficient way to test something for being zero?
- How does an index into an array of pointers to functions compare in performance to a switch statement?

The ternary operator

The ternary operator is terse, slightly cryptic and a favorite of mine. Examination of K&R shows that the authors use the ternary operator very liberally indeed. But does the ternary operator actually accomplish anything? Well, the answer with the Keil compiler is an emphatic no, since it produces larger code and longer execution time in every test case I tried. Listing 5 shows Test Case 1 of the decision making project.

The code in main() is designed to test the functions "both ways." Other tests, involving bits, floats, and so on, all produced similar results. This is the second example where a simpler coding style produces better code.

Evaluating a variable for TRUE / FALSE

This issue seems to occur frequently. In a nutshell, we are interested in determining if a variable is zero (FALSE) or nonzero (TRUE). This seems so trivial that you wouldn't imagine that performance would vary widely depending upon the constructs used. Think again!

Test Code 2 from the decision-making project is shown in Listing 6. In this case, I have three functions whose sole task is to test a byte variable and return 0 if it is zero, and 1 if it is not. The result is represented as a bit. The first function solves the problem by using the most straightforward representation. For simplicity and clarity, it is hard to beat. It also has the possible

benefit of having equal execution time regardless of the value of the parameter. The second example is more obfuscated while giving no real benefit (a byte saved versus increased and variable execution times). The third function is either incredibly simple or ridiculously obtuse, depending on your perspective. It does, however, give dramatically better performance and comes about from an exceedingly neat trick by the compiler.

This is an interesting example, since I think it demonstrates two points:

As we have seen in other examples, the compiler seems to generate better code when simpler constructs are used. Hence example 1 is better than example 2.

Example 3 gives the best result because it makes use of a type cast. Type casts occur so frequently that I suspect the compiler writers have put a lot of effort into generating optimal coding constructs for them.

Summary

From the preceding examples, we can come up with a few rules of thumb for decision-making:

- Avoid the use of the ternary operator
- When testing for TRUE / FALSE, omit all explicit tests and rely on the compiler to cast the variable to a Boolean

Calling one of N functions

Though this is a form of decision-making, the problem of calling one of N functions is interesting and complex enough to warrant its own section. Essentially, we wish to solve the problem of calling one of N functions based on some index value. There are three general approaches to this problem:

- if-else chain
- switch statement
- Array of pointers to functions

If you are unfamiliar with arrays of pointers to functions, you can read my article “Arrays of Pointers to Functions” to find out more.

The test project (Pointer to Function) contains four test cases. In all cases, we are calling one of N functions, where the functions `fna()` ... `fnk()` simply return a ... k respectively.

In this case, we wish to call one of five functions based on an index whose range of legal values is 0...4. This is a small enough number of choices that any of the three techniques is reasonable. The results of the three techniques are summarized in Table 3.

Note that index 6 is an illegal value, and hence tests how the technique handles this problem. There are a number of interesting results here:

- The else-if and switch techniques have identical code size. However, examination of the assembly language shows that completely different techniques are used.
- The else-if construct execution time increases with the value of the index. This seems reasonable, since we first test for 0, then for 1 and so on.
- The code is considerably more compact than the other two techniques.
- The switch construct execution time decreases with the value of the index. This comes about from a rather elegant code construct that the compiler generates. However, I'd consider this counter-intuitive.
- The pointer to function code execution time is constant for all valid indices.
- The pointer to function code execution time is considerably longer than the other techniques. However, a large portion of this is due to the limit checking on the index. If this can be eliminated (because the index is guaranteed to be within a

valid range), then the penalty is less significant.

If the number of elements in the switch statement exceeds six, the Keil compiler uses a different technique for evaluating the switch statement. Test Case 2 examines this point.

Test Case 2 is identical to Test Case 1, except that the list of possible functions is extended to 11. I suspect that most people would eschew an else-if chain at this point, and so I have dropped it from the test.

The results of the two remaining techniques are summarized in Table 4.

Note that index 20 is an illegal value, and hence tests how the technique handles this problem.

Again, there are a number of interesting results.

- The switch statement now produces consistent execution time (except for the default case).
- The code size of the switch statement is significantly larger than the pointer to function approach.
- The switch statement still has a significant edge in execution time. Again, if the limit checking code can be removed then this difference becomes much smaller.
- Test Case 3 reproduces Test Case 2, except the indices are offset. The results are basically the same as Test Case 2, showing that the compiler is smart enough to handle nonzero-based contiguous indices. However, what happens when the indices are non-uniformly spaced? Test Case 4 addresses this question

In Test Case 4, we still have 11 possible function calls. However, the indices for the functions are non-uniformly spaced. For the switch-based code, we simply fill in the requisite values as shown below:

```
char switch_fn(unsigned char i)
{
    char res;

    switch(i)
    {
        case 0x30:
            res = fna();
            break;

        case 0x35:
            res = fnb();
            break;

        case 0x36:
            res = fnc();
            break;

        case 0x39:
            res = fnd();
            break;

        case 0x3C:
            res = fne();
            break;

        case 0x41:
            res = fnf();
            break;

        case 0x42:
            res = fng();
            break;

        case 0x48:
```



```

        res = fnh();
        break;

    case 0x4A:
        res = fni();
        break;

    case 0x4D:
        res = fnj();
        break;

    case 0x4E:
        res = fnk();
        break;

    default:
        res = 0;
        break;
}

return res;
}

```

For the pointer to function approach, we fill in the gaps with calls to a default function; in other words, a function that returns the default value of zero. Thus, the executable code looks the same. It's just the lookup table that has increased in size. The code to do this follows:

```

char ptof_fn(unsigned char i)
{
    static char (code * code pf[])(void) = {
        fna, fnDefault, fnDefault, fnDefault,
        fnDefault, fnb, fnc, fnDefault,
        fnDefault, fnd, fnDefault, fnDefault,
        fne, fnDefault, fnDefault, fnDefault,
        fnDefault, fnf, fng, fnDefault,
        fnDefault, fnDefault, fnDefault, fnDefault,
        fnh, fnDefault, fni, fnDefault, fnj, fnk
    };

    char res;

    if ((i >= 0x30) && (i <= 0x4E))
    {
        res = pf[i-0x30]();
    }
    else
    {
        res = 0;
    }

    return res;
}

```

The results of this test are shown in Table 5.

Note that index 49 is an illegal value, and hence tests how the technique handles this problem.

Again, there are a number of really interesting results here:

- The switch statement code now produces variable execution times
- The execution time of the switch increases significantly with increasing index
- The default value execution time of the switch is dramatically increased, which would be problematic if the default case

was the most common

- The switch footprint is considerably larger than the function pointer approach. However, much of this is attributable to library code, which presumably would be reused in other switch evaluations.

Summary

From the preceding examples, we can come up with a few rules of thumb for calling one of N functions:

- If consistent execution time is important, use an array of function pointers. But if the number of indices is greater than six and the indices are contiguous, use a switch statement.
- If code size is important, always use an array of function pointers.
- If average execution time is of paramount concern, then you have to do some analysis:
 - For six or fewer choices, use an if-else chain.
 - For more than six choices, where the choices are contiguous, use a switch statement.
 - For all other cases, use an array of function pointers.

If you have a situation where the number of choices may be altered over time, and especially if the choices may become noncontiguous, then use an array of function pointers. This will prevent big changes in performance from small changes in code.

Pointer manipulation

Pointers are an integral part of most C programs. In addition to dereferencing them, it is common to increment or decrement them within the body of a loop. Anything that is regularly performed within a loop is worth looking at carefully to see if any optimizations are possible. To this end, I created the pointer manipulation project. This project attempts to answer the following questions:

- How does combining the pointer manipulation with the dereference affect performance?
- Is there any difference between incrementing and decrementing a pointer?
- What effect does the memory space have on these issues?

The answers to these questions are not intuitively obvious because the 8051 has some weird characteristics:

1. When accessing data space via a pointer register (for example, R0), the register can be incremented or decremented with equal ease.
2. In contrast, when accessing external data space via the DPTR register, the instruction set only defines an increment operation. Thus, we might expect pointer decrements to be more problematic in this case.

The pointer manipulation project contains a dozen functions that all perform the equivalent of `memset()`. In each case we are trying to set the contents of a 20-byte array to a specified value.

Function calls `att1()`, `att2()`, and `att3()` examine the optimal way to combine a data space pointer dereference with a pointer increment. The results are summarized in Table 6, and are very interesting. There is a massive penalty to be paid for combining the pointer increment with the dereference as done in `att1()`. Surely, I thought to myself, this must be some manifestation of dealing with internal data space pointers? So I performed the same test using external data space pointers. Function calls `att7()`, `att8()`, and `att9()` were the test code. The results are summarized in Table 7. As you can see, we get the same basic result. This was quite a shock to me, since my coding style preference is to use `*ptr++`.

I know that there is a school of thought in which my preferred way of doing things is bad, as I'm doing too much in one line of code. Well, for those of you who subscribe to this school, here's evidence that (at least for this compiler) your way of doing

things is also more efficient.

It's worth commenting on the fact that the pre-increment (`att9()`) offers a one cycle saving over the post-increment (`att8()`). This was also a surprise to me. For those of you who don't like pre-increments (because it looks strange), go ahead and use the post-increment, since the penalty is miniscule. For those of you who absolutely have to have the best performance, use the pre-increment.

Here's `att9()`'s implementation:

```
void
att9(unsigned char xdata *ptr,
      unsigned char val, unsigned char len)
{
    unsigned char i;

    for (i = len; i; i--)
    {
        *ptr = val; ++ptr;
    }
}
```

Note the use of the count-down for-loop to achieve optimal looping time.

Having obtained such a surprising result, I was obviously intrigued about other pointer manipulations. How does pointer incrementing compare to pointer decrementing?

Functions `att4()`, `att5()`, and `att6()` are identical to `att1()` ... `att3()` except that we start at the top of the array and decrement the pointer. The results were identical. This was expected since the index register for data space access can be as easily decremented as incremented.

The same is not true for external data space access, however, where there is no provision for decrementing the DPTR register. Functions `att10()` ... `att12()` are the equivalents of `att7()` ... `att9()`. Once again the results were surprising; they're summarized in Table 8.

In the case where we combine the pointer increment/decrement with the dereference, (`att7()` and `att10()`), we are better off using a pointer decrement. In contrast, where we split the instructions apart there is a significant penalty to be paid for decrementing the pointer. Finally, note that the pre-decrement also holds a one cycle advantage over the post-decrement, just as for the increment case.

A reasonable question to ask is "Do these results hold when we are pointing to anything else other than a character?" I don't know. Most of my 8051 pointer manipulation seems to occur on communication buffers, so I haven't been motivated to find out the answer to this question.

As a final note on this topic, I also compared calling `memset()` to the optimal solutions. `memset()` was considerably faster, but required more code space. The copying memory project that we look at next examines this issue in more detail.

Copying memory

Copying one area of memory to another is a common requirement. Indeed, it is so common that ANSI defined `memcpy()` to do this for us. In just about every environment, one would expect that `memcpy()` would be the fastest way to copy memory. However, this isn't always the case with the 8051. The reason is a combination of the ANSI standard, the 8051 architecture, and implementation details of the compiler.

Let's start with our friends at ANSI. First, the `memcpy()` library routine is required to return a pointer. Although I understand the rationale for this, I almost never use the result of a call to `memcpy()`. Thus the compiler writers are forced to generate and execute code that is of no interest to me. Secondly, `memcpy()` is required to take a length argument that is a signed int (why anyone would want to pass a negative length to `memcpy()` is beyond my comprehension). Signed integers carry a heavy penalty

on the 8051. This would be OK, except that on most of my 8051 projects, I normally need to copy much less than 256 bytes, so that the length parameter can be an unsigned char. Thus, ANSI forces me to pass two bytes to `memcpy()` where one byte would do fine. In summary, ANSI handicaps the `memcpy()` writer in three ways:

- Requires a pointer to be returned
- Requires the length parameter to be an integer, resulting in more bytes being passed than is normally necessary
- Requires the length parameter to be an integer, resulting in the looping variable being unnecessarily large

Of course, if a particular invocation does require copying more than 256 bytes, then the last two complaints are moot.

The length issue is nothing compared to the problems caused by the 8051. The 8051 has four different address spaces that make sense from a `memcpy()` perspective (using the Keil nomenclature: IDATA, PDATA, XDATA, and CODE). Three of these are legal destination spaces, while all four are legal as the source of the data. Thus, there are 12 possible memory space combinations that have to be supported. To accommodate this, Keil defines `memcpy()` to take two generic pointers. For those of you that are really paying attention, you'll remember that the Keil parameter passing rules don't allow two generic pointers to be passed in registers. However, examination of the `memcpy()` assembly language shows that Keil doesn't follow its own rules. In fact it passes both pointers and the length integer in registers. (The cynic in me wants to know why they are allowed to do this and I'm not.)

Anyway, at this point, the `memcpy()` routine has two generic pointers it has to deal with. Keil's solution is to determine at run time the memory spaces that the pointers point to and to call the appropriate function. I'm a little surprised that Keil cannot determine this information at compile time and simply call the requisite routine. However, I don't write compilers for a living, so I'm sure there is a good reason.

To summarize, the `memcpy()` routine has the following handicaps:

- Needs eight bytes to be passed in registers
- Uses a 16-bit looping index-even when eight bits will do
- Determines at run time the memory spaces to use
- Has to return a value

Against these, the compiler has two advantages:

- Hand optimized assembly language
- If the 8051 derivative has two DPTR registers (a common extension to many 8051 families), the function can make use of this to dramatically improve performance for XDATA-XDATA or CODE-XDATA moves

Based on this information, it would seem reasonable that `memcpy()` may not be the optimal solution in all cases. The `memcpy` project was put together to examine this very issue. The project contains 12 tests. Some of these use `memcpy()`, while others use variations on for loops and array/pointer manipulations. These 12 tests are not exhaustive, but do serve to demonstrate a reasonable pattern. Tests from code space to data space are not shown, since this doesn't seem to occur very frequently.

The detailed results are given in Table 9. A few comments are needed on some of the columns.

># DPTR registers: This specifies how many DPTR registers I told the compiler were available. The only time this is relevant is when doing an XDATA – XDATA (or CODE – XDATA) copy using `memcpy()`. In all other cases, the compiler simply ignores the availability of a second DPTR register.

Implementation: This gives a very brief summary of the basic technique used to achieve the result. For-loop/arrays means that the code looked something like this:

```
for (i = 0; i < SIZE; i++)
{
```

```

    to[i] = ex[i];
}

```

For-loop/pointers means that the code is based on constructs similar to:

```

for (i = 0, toptr = to, exptr = ex; i++)
{
    *toptr++ = *exptr++;
}

```

You'll have to look at the source code to get the definitive statement on each implementation.

Times: The 1-byte and 11-byte times allow me to compute the setup time (the time taken to set up all the looping constructs) together with the per byte time (the delta time over the setup time for each additional byte to be copied). This information allows me to compute the break-even point for comparing various techniques.

Several conclusions may be drawn from these results:

1. The `memcpy()` function requires about 56 to 64 cycles of setup time. This is a lot, and is attributable to the number of registers to be loaded and the fact that the memory spaces are determined at run time.
2. The most efficient method of coding one's own copying data routine seems to be the most straightforward, namely a for-loop with arrays (test 8). This is another example of where the simplest coding construct works best. It's also notable that the array-based code is faster than pointer-based code. So much for the adage of "pointers are faster than array subscripting."
3. If we compare the for-loop to `memcpy()`, the break-even point occurs around seven bytes in most cases. So if you have to copy more than seven bytes, use `memcpy()`. Otherwise use a for-loop.
4. The big exception is when doing XDATA-XDATA moves on an 8051 with a single DPTR register. In this case, we are comparing the results of test 4a with test 8. Here `memcpy()` is always slower than the for-loop. Of course, if you need to copy more than 256 bytes, this result doesn't hold. Notwithstanding this, I still find this result pretty amazing.

If speed is what you need, then use these results. However, if you are copying blocks of data around as part of system initialization, I urge you to consider code size, readability and maintainability before making your decision. Indeed, if code size is your prime consideration, you may want to avoid `memcpy()`, even if you use it in multiple places. The reason for this is quite subtle. It takes about 19 bytes of code to call `memcpy()` (the compiler has to load eight registers). In addition, the actual library code is 246 bytes. By comparison, the code size for Test 5 is only 16 bytes, while the code size for Test 8 is 30 bytes. You'd have to duplicate this code numerous times to equal the code required by `memcpy`.

Bit banging

By way of a conclusion, I thought we'd take a look at a reasonably realistic example that allows me to demonstrate a few of the techniques that have been discussed. The example consists of an external peripheral—perhaps a 16-bit digital to analog converter, which requires 24 bits of command and data to be clocked into it.

The bottom line: fancy coding tricks are no substitute for good design.

Source code

The code is available at <http://ftp.embedded.com/pub/2002/11/jones>, where you'll also find all of the other test code referenced in this article. In the final project, there are eight different C programs and one assembly language program. There is a 16:1 variation in speed and a significant variation in code size. However, if you bother to examine the code, you'll find that the biggest improvement in speed came from a change in algorithm. The techniques described here are the icing on the cake.

This article was published in the October 2002 issue of Embedded Systems Programming. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Jones, Nigel. "Optimal' C Constructs for 8051 Microprocessors," Embedded Systems Programming, October 2002