



**TIME-TO-MARKET PRESSURES HAVE FORCED EMBEDDED-SYSTEM DEVELOPERS TO PROGRAM MICROCONTROLLERS IN HIGH-LEVEL LANGUAGES. ALTHOUGH SOME COMPILER VENDORS CONTINUE TO SQUEEZE OUT EXTRA PERFORMANCE AND HIGHER CODE DENSITY, SOME SEMICONDUCTOR VENDORS ARE DEVELOPING NEW COMPILER-FRIENDLY 8-BIT ARCHITECTURES. YOU'LL FIND IT USEFUL TO EVALUATE THOSE ANGLES AND LEARN SOME USEFUL PROGRAMMING TIPS TO HELP YOU GET THE RESULTS YOU'RE AFTER.**

# C for yourself: PROGRAMMING 8-BITTERS

**A**TENTION ASSEMBLY-LANGUAGE PROGRAMMERS: C compilers for 8-bit microcontrollers can generate code as well as—if not better than—you can. Sure, hand-coding is in some cases necessary, such as for small programs or when you must have 100%-deterministic behavior. But, as applications get more complex, micros get faster and

more compiler-efficient, and compiler technologies improve, C prevails. In other words, high-quality C code results from a good compiler, a C-friendly microcontroller, and a good C programmer. What constitutes a good C compiler? Legendary microcontroller architectures, such as the 8051, 68HC05, and Z8, have been around for so many years that you would expect the compiler technology to be very mature, reliable, and efficient. However, programmers tell me that a need exists for new optimization capabilities, new language extensions, and improved graphical user interfaces. And, for the most part, compiler vendors are making these improvements. Otherwise, what would be your incentive to upgrade and provide them with continued business?

## SILICON TECHNOLOGY REMOVES LIMITATIONS

Improving silicon process technology drives new capabilities in microcontrollers, with the most obvious benefits

showing up as increased operating frequency and greater memory capacity. These improvements directly affect compiler technology. First, increased operating frequency allows compiler vendors to more easily overcome limitations of the microcontroller's architecture. For example, accommodating standards is the most difficult challenge for any compiler vendor. (It's amazing how many compilers can't seem to master this challenge.) When Intel ([www.intel.com](http://www.intel.com)) unveiled the 8051, the ANSI C standard didn't even exist. Compiler vendors have welcomed the newer, faster 8051s from such companies as Dallas, Philips, and Silicon Storage Technology to help overcome the architecture's limitations. Whereas a "C-friendly" processor may help a vendor quickly implement a good C compiler, it doesn't necessarily imply that you can't create a good compiler with another processor. Furthermore, higher clock frequencies can help overcome an "old" architecture's shortcomings.

*At a glance*..... 102

*Exploring compiler-friendly features* ..... 102

*For more information* .... 106

Greater memory capacity has also created a more C-friendly environment, making it easier for compiler vendors and programmers. Nevertheless, code size still is one of the highest priorities. One compiler feature to improve code size, implemented by a variety of compiler vendors, including Crossware, IAR, and Keil, is common-block replacement, or common-code merging. To accomplish common-block replacement, Keil Software's 8051 compiler scans compiled code for common blocks that it can replace with a subroutine. This function, which Keil implemented in its Version 6 release, requires large amounts of host-system memory because the compiler

### AT A GLANCE

▶ Although most compilers for 8-bit microcontrollers are stable and efficient, vendors continue to make improvements to take advantage of architectural improvements.

▶ When possible, check the compiler's output to make sure it takes advantage of a microcontroller's instruction set and underlying architecture.

▶ Be careful. No two compilers are alike, and many may not generate the proper instructions for the respective C code.

▶ Certified benchmark scores from the EDN Embedded Benchmarks Consortium ([www.eembc.org](http://www.eembc.org)) will help you compare compiler performance for speed and code size.

must simultaneously compare many instantiations of the code. Conveniently, Keil's Version 6 also represents a move to Microsoft ([www.microsoft.com](http://www.microsoft.com)) Windows, which, theoretically, has no mem-

ory limits. (MS DOS limits a program's memory to 640 kbytes and therefore requires you to use DOS extenders, potentially creating integration problems with some third-party make utilities.)

## EXPLORING COMPILER-FRIENDLY FEATURES

You could probably successfully compile 90% of the embedded applications on at least one member of any microcontroller vendor's products. But that's not to say that that product will produce efficient code. Vendors designed some microcontrollers without any thought that you might ever use it with a high-level language, whereas others included as many C-friendly features as possible in their minimal resource architectures. It's impossible to discuss every microcontroller family and its associated "compiler-friendly" and "unfriendly" features. However, the following provides the basic features of what you should look for as you select a microcontroller and corresponding compiler.

As mentioned, the 8051 arrived before the introduction of the ANSI C standard, which presented a challenge for C compiler vendors. However, as you know, the 8051 has been among the most popular microcontroller architectures. A further testimony to the 8051 is that each year, one or more semiconductor vendors joins the flock of companies providing a product with an 8051 core. (You can check out [www.ednmag.com/ednmag/reg/micro.asp](http://www.ednmag.com/ednmag/reg/micro.asp) for a listing of 8051s.) As a result,

the 8051 also has garnered the widest selection of compilers.

From the viewpoint of many compiler vendors, the 8051 microcontroller has some unusual features that make it an order of magnitude more difficult to write a compiler for than most other architectures. Program memory, internal RAM, external RAM, and the special-function registers have overlapping address ranges. This approach complicates matters for C pointers because a pointer to an address may be a pointer to any of these memory areas. In addition, the 8051 lacks an indexed addressing mode to access RAM. Therefore, to access a dynamic stack frame, you have to calculate the address within the stack frame and load the result into the memory pointer. You would ordinarily accomplish this task by loading an index register with the value of an offset relative to the stack frame pointer. Most 8051s have only one general-purpose pointer to external RAM, further complicating matters. Several 8051 providers, such as Atmel, Dallas, and Philips, have modified the architecture to include two data pointers; Infineon's newer 8051 implementations have eight data pointers.

Microchip's PIC is another

popular 8-bit architecture with a variety of compiler support. The early PIC cores had limitations that detracted from their C-friendliness, but the company designed its new PIC18CXXX core specifically with C in mind. Some of the enhanced features of this architecture include a linear 21-bit program memory space that helps to eliminate paging. Earlier PICs used paged ROM, which required special techniques to perform calls and jumps across pages, although the compiler should automatically perform this task. The PIC18CXXX also has a linear 4-kbyte RAM space that eliminates the need for banking and allows you to access it via indirect addressing. Again, earlier PICs required banking; hence the programmer's need to set and reset bank selection bits. The compiler saves you from having to work out the optimal sequence of RAM accesses to minimize bit twiddling.

The PIC architecture lacks a hardware stack, which explains the absence of any RTOS support. Compiler vendors partially overcome this limitation by creating a pseudostack for the allocation of memory. Another feature that was absent on PIC architectures before the PIC18CXXX is the ability to do preincrement,

postincrement, predecrement, and postdecrement. This ability is useful for fast, efficient pointer manipulation. Additional well-known architectures include Motorola's 68HC05 and 68HC08. The HC05's 8-bit index register with an offset as large as 16 bits works well for most data structures because the 16-bit offset allows you to locate structures anywhere in the microcontroller's address space. If the structure is greater than 256 bytes, you need extra code to move between the 256-byte data structures, so the HC08 extended the index register to 16 bits to eliminate this limitation. The HC08 also added stack-relative addressing to help support parameter passing. You can also take advantage of indexed addressing to access multiple data tables using a single index pointer with different offsets. Without this feature, the compiler would have to generate additional instructions to load a new pointer for each table. Another performance-enhancing feature of the HC08 is its *dbnz* instruction, which increases the performance of loops that end in zero. Ending loops with anything other than zeros requires additional decrement, compare, and branch instructions.

(continued on pg 104)

**EXPLORING COMPILER-FRIENDLY FEATURES (CONTINUED)**

Toshiba provides compiler support for its TLCS-870 series with its Toshiba ANSI C compiler. The microcontroller family has several features that assist the compiler in generating efficient code. These features include eight general-purpose 8-bit registers that allow it to pass function arguments in registers and a program stack implemented with a 16-bit stack pointer register. The 870/C and 870/X series also include 16-bit index registers that allow you to implement efficient addressing modes involving pointers. The TLCS-870 and TLCS-870/X series also include 16 banks of general-purpose registers, which support high-speed interrupt handling. Toshiba provides a C-like compiler to help you achieve better code efficiency than the C compiler offers. Although the language has many similarities to C, the Toshiba language allows you to directly manipulate the

microcontroller's registers and do coding in machine instructions, helping to eliminate the need for any assembly language.

**C-FRIENDLY ARCHITECTURE**

Just a few years ago, Atmel introduced its 8-bit AVR architecture. It has 32 working registers with the same functions as a traditional accumulator—a bonus for efficient C coding. You can combine some of these registers to yield 16-bit pointers to access data in data and program memory. For large memories, you can add a third register to yield 24-bit pointers and access as much as 8 Mbytes of data without page switching.

The AVR architecture has four memory pointers. One is a dedicated stack pointer for storing a function's return address. IAR's C compiler allocates one pointer for a software stack. The compiler uses the two remaining point-

ers as general-purpose pointers to load and store data. **Listing A** provides an example of using two data pointers to move data from one memory location to another. The AVR microcontroller can also access data memory by direct addressing; this feature gives a programmer access to the entire data memory in a two-word instruction without page switching. Next year, IAR Systems will deliver a C++ compiler for the AVR architecture, further underlining this architecture's high-level-language friendliness.

Mitsubishi's 740 family microcontroller architecture allows absolute reads and writes, giving programmers direct access to absolute memory locations at the C level. The 740 microcontroller architecture has a 256-byte, zero-page structure that can accommodate small memories in which all the code and variables for single-chip applications are contained within the zero-page space. The architecture also accommodates large models in which the code exceeds the space available in RAM and variables must reside

outside the zero-page space.

STMicroelectronics' 8-bit microcontrollers also support multiple index registers and general-purpose registers to make life easier for the compiler. The devices provide multiple addressing modes to increase flexibility and efficiency. Hitachi's H8 uses a general-purpose, register-based, load/store architecture with eight 16-bit registers that a programmer can use as address pointers, data storage, or index registers.

Last but not least, for all you Commodore fans, the Western Design Center (the original source and intellectual-property owner of the patented W65-C02TM 8-bit and W65C816TM 16-bit microprocessors) forged ahead with the introduction of its Core Alliance Program. IAR is one member of this program, so you should see better compiler support for these devices. (And, just FYI, Creative Micro Devices ([www.cmdweb.com](http://www.cmdweb.com)) has designed the 20-MHz Super CPU accelerator cartridge for the Commodore 64 and 128. This cartridge uses Western Digital Design Center's W65C816S microprocessor.)

**LISTING A—TWO DATA POINTERS MOVE DATA FROM ONE MEMORY LOCATION TO ANOTHER**

```
char table[50];
char *pointer1 = table[0];
char *pointer2 = table[49];
*pointer1++ = *--pointer2;
```

This generates the following assembly code and executes in two cycles:

```
LD    R16,-Z ; Pre-decrement Z pointer and load data
ST    X+,R16 ; Store and post increment
```

Keil's common-block replacement can replace any block larger than 5 bytes; however, the company is still tuning the optimizer, so that number may later change. Keil has found that its Version 6 compiler can reduce program size as much as 25% (see a sample listing on EDN's Web-site version of this article at [www.ednmag.com](http://www.ednmag.com)). As you might expect, the larger the C source file, the better job the compiler can do at common-code merging. Although it may not be the best programming practice, some programmers are lumping all of their C modules together to get better code savings. The nonorthogonal architecture of the 8051 provides many opportunities for common-code blocks because many standard operations require multiple instructions that you can optimize.

Crossware Products' second-generation 8051 compiler also implements common-code merging. Typically, a 32-bit microprocessor can perform 4-byte integer arithmetic and comparisons in a single instruction. But the compiler for an 8-bit microcontroller must generate a sequence of instructions to accomplish the same function. This sequence of instructions or code fragment can occur repeatedly, and it may be tempting to put these sequences in a library subroutine so that the compiler can instead generate a subroutine call. Crossware's original 8051 C compiler worked in this way, and it generated compact code. However, the logistics of maintaining both a library of subroutines and the compiler code that called them considerably impeded the speed at which the

company could develop enhancements.

Therefore, the only subroutine calls Crossware built into its new compiler are to floating-point arithmetic and multiply-and-divide routines. The compiler generates subroutine calls for common fragments, and the user controls the size of the code fragment that the compiler should consider for merging. This approach allows you to fine-tune the program size to fit the available space and minimize the number of subroutine calls.

**IMPROVING CODE QUALITY**

In addition to common-code merging, good compilers can reduce memory-space requirements by constructing a call tree for all functions in a program. The compiler uses this call tree to determine which static stack frames can safely share

memory. Static stack frames allow the compiler to calculate the absolute addresses of variables on the stack at compilation time. Functions with static stack frames are non-re-entrant because a second recursive call to a function would overwrite the variables of the first call. Crossware Products' and Byte Craft's compilers, for example, carefully analyze interrupt functions and functions called by function pointers to avoid this problem. Crossware notes that one "leading compiler" requires users to manually specify pointer-called functions to prevent the linker from incorrectly overlaying the stack space.

Functions, such as `printf()`, having a variable number of arguments, may also pose a problem because the function does not know how many arguments the calling routine will pass to it. Some compilers require users to specify how much space to reserve for these cases. Other compilers, such as Crossware's, combines global information for all calls to a function of this type and calculates the exact amount of stack space to reserve.

Another compiler capability to watch for deals with function pointers involving static stack frames. In this situation,

the function pointer must point to the function code and to that function's static stack frame. With some compilers, the function pointers to non-re-entrant functions hold this additional information. Be careful, because some compilers do not support pointers to non-re-entrant functions that have parameters. Others require that all the function's parameters fit into registers. (As a side note, most applications for 8-bit microcontrollers don't have re-entrant routines due to limited available memory. Allocating memory during compilation has the advantage of no runtime overhead. On the other hand, runtime allocation means that in addition to the allocation/deallocation overhead, the code must make at least one more memory access for variable dereferencing, thus further increasing execution time. Another issue occurs when a re-entrant routine runs out of space. How does the system handle this situation?)

Harvard architectures, such as the 8051, generally have a more complicated way of handling pointers (see **sidebar** "Exploring compiler-friendly features"). You would typically make all unqualified pointers generic pointers, using an extra

### LISTING 1—SMART-POINTER CODE REQUIRING NO KEYWORD

```
// How to avoid 3 byte generic pointers:

// Without Smart pointer
// Large memory model

int func(char _xdata* pch1)
{
    char _xdata* pch2;
    pch2 = pch1;
}

// Without Smart pointer
// Small memory model

int func(char _data* pch1)
{
    char _data* pch2;
    pch2 = pch1;
}

// With Smart pointer
// Any memory model

int func(char* pch1)
{
    char* pch2;
    pch2 = pch1;
}
```

byte of information to hold a value representing the memory area that the pointer points to. Allowing a user to qualify a pointer with keywords such as

## FOR MORE INFORMATION...

For more information on the products discussed in this article, circle the appropriate number on the Information Retrieval Service card or use EDN's InfoAccess service. When you contact any of the following manufacturers directly, please let them know you read about their products in EDN

#### Atmel Corp

www.atmel.com  
1-408-441-0311  
Circle No. 305

#### Byte Craft Ltd

1-519-888-6911  
www.bytecraft.com  
Circle No. 306

#### CMX Co

1-508-872-7675  
www.cmx.com  
Circle No. 307

#### Crossware Products

+44 1223 421263  
www.crossware.com  
Circle No. 308

#### Dallas Semiconductor

1-972-371-0448  
www.dalsemi.com  
Circle No. 309

#### Franklin Software Inc

1-408-296-8051  
www.fsinc.com  
Circle No. 310

#### Hitachi

1-800-285-1601, ext 21  
www.halsp.hitachi.com  
Circle No. 311

#### Hi-Tech Software

1-800-735-5715  
www.htsoft.com  
Circle No. 312

#### IAR Systems

1-800-427-8868  
www.iar.com  
Circle No. 313

#### Infineon Technologies

1-408-777-4910, ext 166  
www.sci.siemens.com  
Circle No. 314

#### Keil Software

1-800-348-8051  
www.keil.com  
Circle No. 315

#### Microchip

1-602-786-7668  
www.microchip.com  
Circle No. 316

#### Mitsubishi

1-408-730-5900  
www.mitsubishi.com  
Circle No. 317

#### Motorola

1-800-765-7795, ext 60  
http://motosps.com/sps/  
General/chips-nav.html  
Circle No. 318

#### Philips Semiconductors

1-408-991-3518  
www.semiconductors.  
philips.com  
Circle No. 319

#### Silicon Storage Technology Inc

1-408-735-9110  
www.ssti.com  
Circle No. 320

#### STMicroelectronics

1-617-259-2516  
www.st.com  
Circle No. 321

#### Tasking Inc

1-800-458-8276  
www.tasking.com  
Circle No. 322

#### Toshiba

1-714-455-2000  
www.toshiba.com/teac  
Circle No. 323

#### Western Design Center

1-480-962-4545  
www.wdesignc.com  
Circle No. 324

#### Zilog

1-408-370-8000  
www.zilog.com  
Circle No. 325

#### SUPER CIRCLE NUMBER

For more information on the products available from all of the vendors listed in this box, circle one number on the reader service card. Circle No. 326

code, data, and xdata is the traditional method of helping the compiler deal with this difficulty. However, a compiler should be able to determine the memory area based on how the programmer referenced it. Some compilers use what Crossware calls smart pointers, because, instead of defaulting to generic pointers, they default to a to-be-determined memory area; thus, they require no keywords (**Listing 1**). A compiler can easily calculate an expression such as `sizeof(ArrayOfPointersToStructuresContainingPointers)` when the compiler knows the size of all pointers. Without this information, the compiler still does not know the size of the structure and hence the size of a pointer to the structure. Having pointers in the structures point to other structures containing pointers complicates the situation. Further, if the compiler lacks smart-pointer support, you must use the `_xdata` or `_data` keywords to specify the pointer type. Then, if you change the memory

**LISTING 2—FRAGMENTS SHOWING THE DIFFERENCE BETWEEN 10 AND 21 BYTES OF CODE**

```
Code compiled to reference data in any memory location
0200 A6 19 LDA #19 a = 25;
0202 C7 04 4C STA $044C
0205 C6 04 4D LDA $044D b++;
0208 4C INCA
0209 C7 04 4D STA $044D d = a + c;
020C C6 04 4C LDA $044C
020F CB 04 4E ADD $044E
0212 C7 04 4F STA $044F

Code compiled with knowledge of the final data location.
0200 A6 19 LDA #19 a = 25;
0202 B7 50 STA $50
0204 3C 51 INC $51 b++;
0206 BB 52 ADD $52 d = a + c;
0208 B7 53 STA $53
```

model, you must also change the keywords. On the other hand, if the compiler supports smart pointers, you can switch memory models without changing any code.

**ALL MEMORY IS NOT CREATED EQUAL**

Obvious differences exist between RAM and ROM in a system. These differences are even greater in microcontrollers in which ROM access, if possible, requires different instructions and in

some cases, device drivers, to access ROM-stored data. A more subtle difference relates to the RAM areas on many small microcontrollers. Motorola's 68HC05 and 08 family, for example, allocates about 30% of its instruction set to handle instructions unique to the first 256 memory locations. A compiler must be aware of these differences because code size and execution times can differ significantly. **Listing 2** shows code fragments that illustrate the difference between 10 and 21 bytes of code. A code generator that knows the address of the data can produce significantly better code. As the **listing** shows, the bytes of code for the increment go from 7 bytes to 2, but the compiler eliminates 3 more bytes of code by making unnecessary the subsequent accumulator load. Byte Craft's smart linking technology supports this capability.

Although the code can lose its portability, compiler vendors improve the quality of their code output by using

*pragmas*. A “pragma” is an ANSI C-standard method for implementing features that vary from one compiler to the next. For example, the “#pragma pack” pre-processor director specifies the byte boundary for packing members of C structures. Other typical uses are to control pointer sizings or linkage directives. Byte Craft uses pragmas to uniformly declare memory to the compiler and to help determine how to use the given resources. The company further extends this process by allowing you to declare memory that the program can access only through the use of device drivers. For example, you can use internal EEPROM in many processors, memory located on I<sup>2</sup>C and SPI buses, or unused buffer space in LCDs as application variables.

Pragmas also include language extensions that support each 8-bit architecture. Examples of pragmas from Tasking include multiple-data-pointer support, optimization of switch statements, and bank-switching

support to expand the addressable-memory limits of an 8-bit architecture. Other vendors use these and other pragmas.

One other compiler/development tool feature that doesn’t necessarily improve code quality but makes it easier for the programmer is the automatic memory-model designator. For example, Franklin Software’s tools by default determine the “best” memory model to apply. In the past, you had to know the various memory models and their impact. (One programmer once said that he always used large memory models because he is a great programmer and writes large programs!)

No matter how great a compiler and its corresponding microcontroller, situations may exist in which you have to use assembly programming. You may need to optimize the timing for a critical task, or you may run out of ROM space. Also, a compiler tends to perform better as your code gets larger. Although compilers for

8-bit processors are mature and stable, you will continue to see incremental improvements over time. The emerging widespread acceptance of the EDN Embedded Benchmarks Consortium ([www.eembc.org](http://www.eembc.org)) benchmarks will pressure compiler vendors to demonstrate their competitiveness. This in turn will raise the bar on a compiler’s ability to produce the highest performing code. □

---

#### ACKNOWLEDGMENTS

*Special thanks to Alan Harry (Crossware Products). Significant contributions also came from Clyde Smith-Stubbs (Hi-Tech Software), Walter Banks (Byte Craft), and Jon Ward (Keil Software).*

---

#### REFERENCES

1. Chan, Wilson, *Writing Optimized C Code for Microcontroller Applications*, Toshiba America Electronics Components Inc.
2. *A Beginners Guide to Low Integrity Software*, [www.hitex.co.uk/general/lowint.html](http://www.hitex.co.uk/general/lowint.html).

