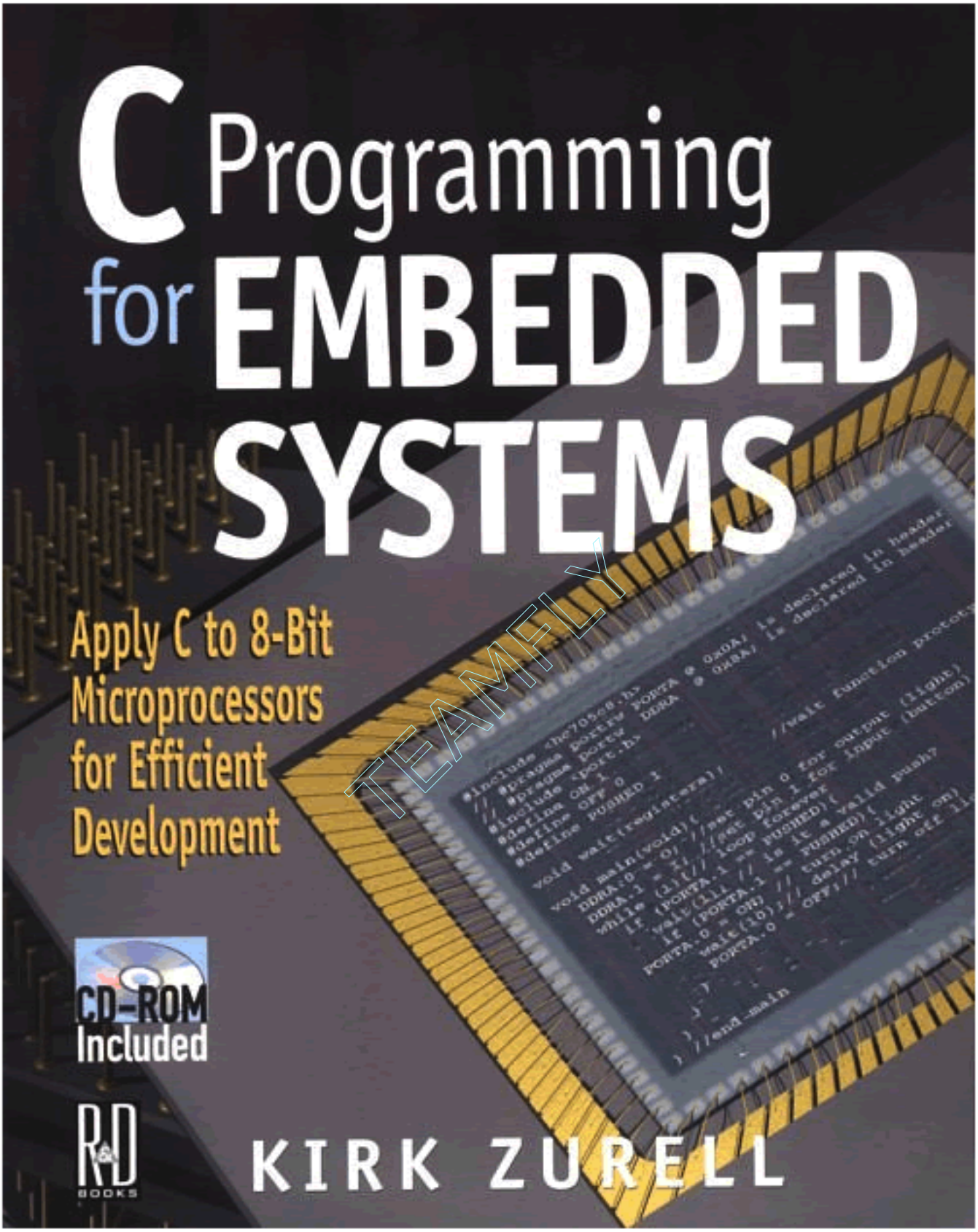

C Programming for Embedded Systems

Kirk Zurell

R&D Books
Lawrence, Kansas 66046

R&D Books
CMP Media, Inc.
1601 W. 23rd Street, Suite 200
Lawrence, KS 66046
USA

Cover art created by Robert Ward.

Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, CA 94710
ISBN 1-929629-04-4

BYTE CRAFT LIMITED
421 King Street North
Waterloo, Ontario
Canada N2J 4E4
Telephone: (519) 888-6911
Fax: (519) 746-6751
E-mail: info@bytecraft.com
http://www.bytecraft.com

Acknowledgments

I would like to thank Walter Banks at Byte Craft Limited for dropping me head-first into the world of embedded programming. Walter and Andre have provided copious expertise in the very finest points of C programming and code generation.

I would also like to thank my parents, who went out on a limb and purchased that Commodore 64 all those years ago. I hereby disclose publicly that I did not wash the dishes forever, as promised.

Table of Contents

Chapter 1—
Introduction

1.1—
Role of This Book

This book provides a complete intermediate-level discussion of microcontroller programming using the C programming language. It covers both the adaptations to C necessary for targeting an embedded environment, and the common components of a successful development project.

C is the language of choice for programming larger microcontrollers (MCU), those based on 32-bit cores. These parts are often derived from their general-purpose counterparts, and are both as complex and feature-rich. As a result, C (and C++) compilers are necessary and readily available for these MCUs.

In contrast, designers who have chosen to use 8-bit controllers have usually resorted to hand-coding in assembly language. While manual assembly programming for precise control will never go out of style, neither will the push to reduce costs. There are advantages in compiling high-level C language to even the limited resources of an 8-bit MCU.

• Automatic generation of code for repetitive coding tasks, such as arithmetic for 16-bit or longer data types.

• Intuitive treatment of hardware peculiarities. Reading from or writing to a serial flash memory device can be represented in C as a simple assignment statement, although the store operation requires some coding.

• Platform-independence. The same cross-platformcapabilities that C brings to desktop computing are available for the range of 8-bit microcontrollers on the market today.

This text shows you how to use C to program an 8-bit embedded MCU. We hope you are familiar with C, but require in-depth information about microcontroller programming.

The main example project in this text is a computer-controlled thermostat. From an initial specification, we progressively refine and augment the device in the same manner as any other consumer or control product. With software development as our focus, we make choices and trade-offs that any designer will need to make.

1.2— Benefits of C in Embedded Systems

The direct benefits of using C in Embedded Systems design are as follows.

You will not be overwhelmed by details. 8-bit microcontrollers aren't just small: microcontrollers include only the logic needed to perform their restricted tasks, at the expense of programmer "comfort". Working with these limited resources through a C compiler helps to abstract the architecture and keep from miring you down in opcode sequences and silicon bugs.

You will learn the basics of portability. Embedded applications are cost-sensitive. There may be great incentive to change parts (or even architectures) to reduce the per-unit cost. However, the cost of modifying assembly language code to allow a program written for one microcontroller to run on a different microcontroller may remove any incentive to make the change.

You can reduce costs through traditional programming techniques. This book emphasizes C code that generalizes microcontroller features. Details relating to specific hardware implementations can be placed in separate library functions and header files. Using C library functions and header files ensures that application source code can be recompiled for different microcontroller targets.

You can spend more time on algorithm design and less time on implementation. C is a high level language. You will be able to program your applications quickly and easily using C. C's breadth of expression is concise and powerful; therefore, each line of code written in C can replace many lines of assembly language. Debugging and maintaining code written in C is much easier than in code written in assembly language.

1.3— Outline of the Book

Determining the goals of software development is the first step, and is covered in Chapter 2. It includes embedded-specific commentary about the regimen of predesign documentation crucial to effective software development.

Chapter 3 provides an introduction to 8-bit microprocessors for those who have not dealt with them on a low level before.

With a good plan and in-depth information about the central controller, the design process (covered in Chapter 4) finalizes what was previously estimated. The processor-specific details about implementing the thermostat are introduced.

Chapter 5 details hardware representation in C. It catalogs all the required set up for your program source.

Chapter 6 provides insight into embedded data. The `near` and `far` variable storage modifiers mean different things on an Intel PC running Microsoft Windows and on an embedded processor running your code.

Chapter 7 completes the C portion, with embedded-specific information on functions, statements, and operators.

Chapter 8 introduces libraries. Even in environments with a pittance of ROM and a very specific task to do, libraries of prewritten functionality are a great help.

Chapter 9 provides insight into optimization, and helps you test your creation thoroughly.

Chapter 10 sums up with more information about the sample project. Though some information is presented throughout the book, this chapter includes content not previously discussed.

1.4— Typographical Conventions

Typography is used to convey contextual or implied information. The following examples provide a guide to the conventions and their meanings.

Table 1.1 Typographical usage

Bold	identifies key terms.
Italic	provides emphasis.
`Letter Gothic`	denotes elements of programming language: identifiers, variable types, keywords, file names, sample code and code excerpts.
`Letter Gothic Italic`	indicates replaceable elements in user input or in computer output.
0x	is used to denote a hexadecimal number. For example: 0xFFF
0b	is used to denote a binary number. For example: 0b010101

1.5— Updates and Supplementary Information

If you are looking for more information on the thermostat project, please consult our supplementary information via web:

http://www.bytecraft.com/embedded_C/

Chapter 2—
Problem Specification

The problem specification is the initial documentation of the problem that your device and software will solve. It should not include any specific design questions or product solutions. The main aim is to explain in detail what the program will do.

Of course, there are as many ways to conduct project planning as there are workplaces on the planet. Even the most standardized phases are observed in different fashions or in a different order. The following sections are included because they add information about the embedded software realm, or they pertain to the sample project specifically.

2.1—
Product Requirements

Often, this document is written from the users' point of view, as a series of user requirements. In the case of an embedded system designed for a single task, you can be quite explicit and certain of the extent of the product's intended functionality.

General decisions about hardware form part of the problem specification, especially in embedded projects in which the hardware will be well controlled.

Results

• Program will measure and display current temperature.

• Program will count real time on a 12- or 24-hour clock, and display hours and minutes on a digital display.

• Program will accept time settings and set clock.

• Program will accept and store time settings for three daily usage periods.

• Program will switch between heating control and cooling control. Note that some HVAC experts will see the need for occasionally operating both heating and cooling at the same time, but this requirement more closely resembles traditional thermostat operation.

• Program will compare current temperature with settings for current time period, and turn on or turn off external heating or cooling units as needed.

• Program will refrain from changing state of external units twice within a short period of time, to permit the HVAC equipment to operate well.

• Program will accept manual override at any time, and immediately turn off heating or cooling unit.

2.2— Hardware Engineering

This book does not deal directly with hardware, except for the example project. Nevertheless, the target platform influences everything about the product. It determines the ease with which code is generated by the compiler, and it determines some overall software design decisions.

If software developers are so lucky as to be involved in the hardware development process, the opportunity to influence the design is too important to pass over. Wish-list items to ask for include the following.

A Built-in Debug Interface Another method of field-programmability would also suffice. When a device must be installed, customized, or repaired on site, a Flash-RAM part makes more sense than an EEPROM or ROM device.

ROM Code Protection Embedded processors often provide protection against casual examination of your ROM code. A configuration bit inhibits reading of ROM through the programming interface. While there are sev-

eral exploits against this protection, only a determined opponent will succeed in reading your programming.

Rational Peripheral Interfaces The temptation to route circuits according to convenience can overwhelm software performance quite quickly when it affects I/O organization. Does the desired processor have bit-manipulation instructions to change port bits independently? Will multiplexed interfaces require too much data direction switching?

Some peripherals can be replicated using generic I/O port lines and driver software. This saves money but adds complexity to the programming challenge. Typically described as "bit-banging", software must quickly and repeatedly write sequences of bits to port output lines, to imitate the logic signals of a dedicated peripheral circuit.

Standard libraries, which might not contemplate a particularly-optimized hardware solution, can pay for the added hardware cost in reduced software cost.

The central decision in hardware design is processor selection. The choice of a processor is a negotiated decision, weighing factors such as the resources needed by the intended application, the cost and availability of the part supply, and the development tools available. For an in-depth treatment of microcontrollers, see the next chapter. Memory estimation does form part of our problem specification, so estimation of RAM and ROM sizes is discussed in Section 2.3.5, Resource Management.

Results

While we don't deal with hardware engineering in this book, we include some sample product specification information for hardware to complete the information set.

Table 2.1 Initial hardware specifications

Engineering Factors	Estimate
Operating Environment	• domestic environment
	• medium-power, medium-noise electrical connections
	• occasional power loss

(table continued on next page)

(table continued from previous page)

Engineering Factors	Estimate
Interfaces	• one multi-bit port for switching HVAC: probably only 3 pins necessary
	• one multi-bit I/O interface for display
	• one multi-bit I/O interface for keypad
	• one A/D device for temperature sensing
	• real time clock source: one second granularity
Memory Size	(See the following text.)
Special Features	• clock/counter or real time clock
	• use of NVRAM depends upon whether and how the processor might sleep
	• watchdog timer might be helpful
Development Tools	• C compiler
	• simulator or emulator
	• development board

2.3— Software Planning

The software plan should say something about the choice of programming language. With embedded systems, there are three general choices of development language: machine language, C, or a higher-level language like BASIC. Of the three, C balances two competing needs.

• C approaches the performance of hand-coded machine language, compared to an interpreted system like many BASICs. If a BASIC system ceases to be basic by exposing pointers or by precompiling the source, the difficulty in testing begins to match that of C.

• C provides device-independence not offered by machine language. If you hand-code a program in assembly, you run the risk of wasting it all with a change in microcontroller. Changing processors in a design programmed in C can incur as little extra effort as changing a header file in your software modules.

The first step in the software plan is to select an algorithm that solves the problem specified in your problem specification. Various algorithms should be considered and compared in terms of code size, speed, difficulty, and ease of maintenance.

Once a basic algorithm is chosen, the overall problem should be broken down into smaller problems. The home thermostat project quite naturally breaks down into modules for each device:

• HVAC interface,

• keypad,

• LCD, and

• temperature sensor;

and then each function of that device.

Working from the block modules, you can write traditional **pseudocode.** This helps form the identifiers and logical sections you will implement in your code.

The **flowchart** begins to make the transition from natural language pseudocode to actual code. In the flowchart, we can begin to speculate about the data that functions will accept and provide. Most importantly, we can begin to plan library usage. Even if there are no prewritten peripheral or data conversion libraries available, we can write original code in library form and much more easily re-use it later.

It is likely that different states have been introduced into the plan. A **state diagram** maps the transitions, as a complement to the flowchart.

From the pseudocode, we can build a **list of variables** and make estimates about RAM and ROM needs. The restriction of memory resources will come as a shock to some. Programmers working with modern desktop environments are comfortable with huge memory spaces. Great fields of RAM are available to create large data structures or arrays that may never actually be initialized or used.

In contrast, microcontrollers sport only as much RAM and ROM as is projected to be needed for a specific class of target applications. Vendors strive to provide a range of similar parts, each variant contributing only a small increase in on-chip resources.

Results

2.3.1—
Software Architecture

The language for programming the thermostat device will be C.

The main architectural dilemma involves the use of interrupts versus polling. Part of this dilemma will be resolved in part selection: some processor variants do not include interrupts at all. Other choices include explicit

support for interrupt-driven keypads, or timers that generate interrupts upon timeout.

A serious facet of an interrupt-based solution is the protocol for communication between the interrupts and main-line code. Since interrupts and main line are as independent as possible (an interrupt may occur during any main-line instruction), race conditions are one consequence.

We have chosen the simplest of several alternative algorithms: a clock/counter interrupt will calculate time, request a display update and set target temperatures. The main line will loop to poll the keyboard, to sample environment temperature, to update the display, and to switch the HVAC machinery. This requires only a precise timing interrupt, which is essential for 24-hour timekeeping.

2.3.2— Pseudocode

Pseudocode presents in natural language the imperative steps of the program. It is especially useful in embedded programming because every aspect of execution can be planned together: there is no need to account for operating system oddities.

In the following example, we assume that time is kept with a counter and software.

1. Initialization

    (a) Set clock counter to 0.

    (b) Set time and temperature target variables to defaults.

    (c) Enable time interrupt.

2. Clock/counter triggers an interrupt each second

    (a) Increment clock counter.

    (b) Request display update.

    (c) Loop through the preset cycles. If clock is at or past the indexed cycle time, set target temperature to that cycle.

3. Main loop

    (a) Sample environment temperature.

        (1) If environment temperature is outside target temperature, turn on heat or cool.

        (2) If environment temperature is inside target temperature, turn off heat or cool.

    (b) Write time, environment temperature, and status to LCD.

(c) Wait for keystroke

(1) If key is pressed, wait for debounce period and check again.

(d) Parse keystroke

(1) If shutdown command is sent, shut down operating units immediately.

(2) If cycle selection command is sent, change to next cycle record.

(3) If time setting is sent, adjust time in current cycle record.

(4) If temperature setting is sent, adjust temperature in current cycle.

2.3.3— Flowchart

This diagram is basically a representation of the relationships between major and minor tasks in the embedded software. The flowchart helps determine

• what functionality goes in which logical module and

• what functionality you expect (or hope) to be supplied by libraries.

You can also begin to give identifiers to important constructs.

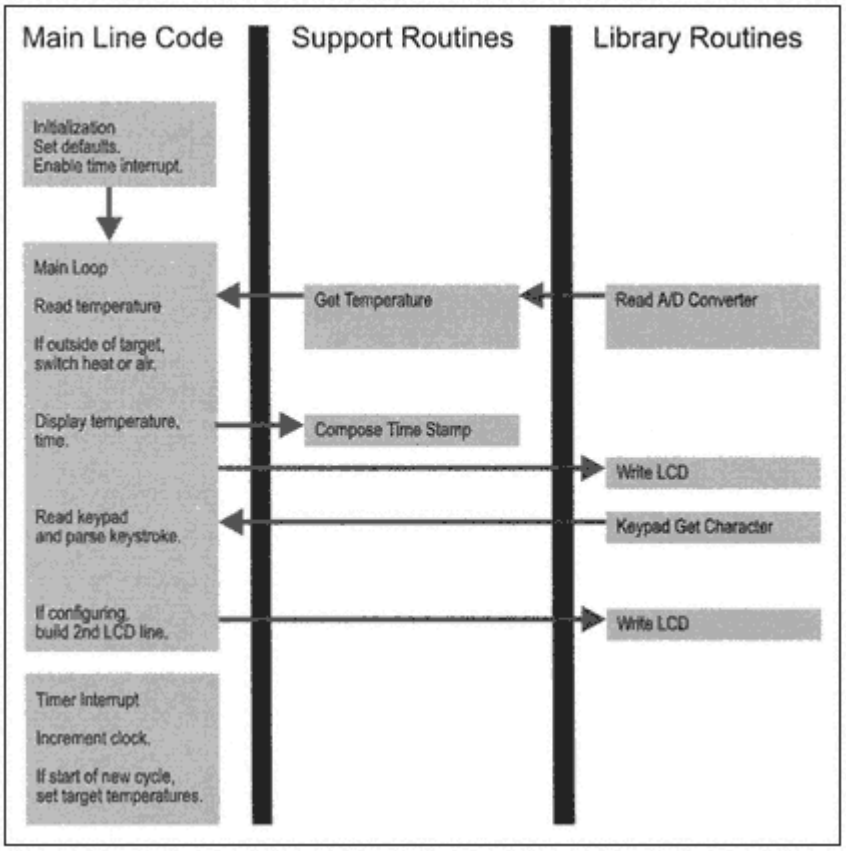

Figure 2.1
Data flow for the algorithm

2.3.4— State Diagram

The software will likely express different states, moving between them after processing external interaction or internal events. This diagram illustrates these states and the stimuli that make it progress through them.

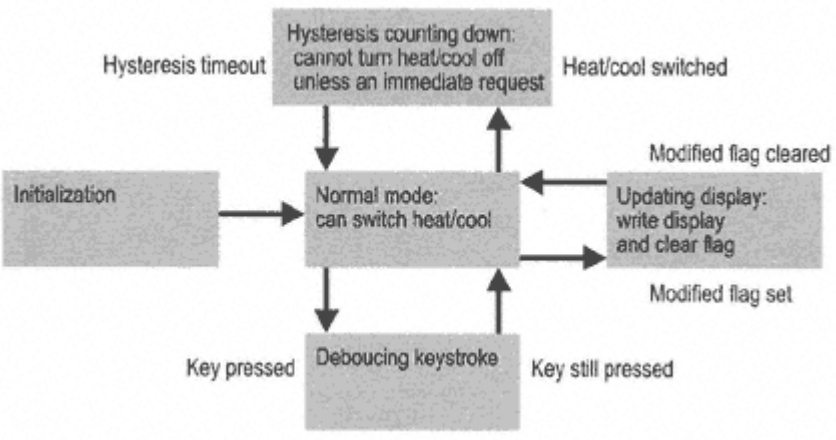

Figure 2.2
State diagram for the algorithm

2.3.5— Resource Management

In the restricted environment of a microcontroller, one too many variables or constants can change the memory requirements, and therefore the price, of the selected part. Features like multiple language support can quickly boost the resource requirements to a new level.

It makes sense to explicitly plan out the resources needed. This is not terribly premature — we are still talking about generic variables here, not specifics like page 0 access, serial ROM, or other technical choices.

If you have written assembly language programs before, estimating memory demands is easier. Without that experience, writing sample code and compiling it is the only way to forecast precisely. Fortunately, using C helps conserve all that development effort.

A rough outline follows.

Table 2.2 Estimating memory requirements

Variable/Module	Resources
Real time clock	~10 bytes RAM, both a counter and a text representation.
Daily cycle records	~20 bytes RAM.
User settings	~10 bytes RAM.
Stack	~10 bytes RAM: two or three function calls, and an interrupt.
Local variables	~10 bytes RAM.
Total RAM estimate	~60 bytes RAM.
Constants	~100 bytes ROM.
Interrupt service routine	~100 bytes ROM.
Initialization	~50 bytes ROM.
Main line	~300 bytes ROM.
A/D conversion (temperature sensor)	~50 bytes ROM.
LCD	~300 bytes ROM, with wide variation depending upon type of interface.
Keypad decode	~100 bytes ROM.
Total ROM estimate	~1,000 bytes ROM.

2.4— Testing Regime

Suggested steps for debugging embedded software include the following.

• Design for debugging.

• Code inspection.

• Execution within a simulator environment.

• Execution within an emulator environment.

• Candidate target system in a test harness.

Both hardware and software can benefit from early consideration of debugging needs. Especially in systems with alphanumeric displays, software can communicate faults or other out-of-spec information. This infor-

mation is useful both to the tester and the end user, but it may prove a liability if the market will not tolerate equipment that appears to fail.

In the absence of the panel, LEDs can signal meaningful states or events. Provision for run-time diagnostic feedback should appear in the pseudocode and resource projections.

The first step in debugging requires you to inspect the assembly code generated by the compiler. Embedded control applications on 8-bit CPUs are small enough, and the architecture simple enough, that a developer can review the entire generated assembly language easily. A listing file, which lines up C source fragments with the assembly they generate, provides the easiest navigation.

Beyond this first step, however, testing becomes a challenge: when the code in question implements the most basic behaviour of the machine, in-system debugging becomes more difficult. A bug may prevent any meaningful response from the embedded system at all, whereas desktop operating systems can provide core dumps or other diagnostic aids.

To make in-system debugging possible, simulators and emulators peer into the embedded system. Each tries to approximate different areas of the target environment while allowing you to inspect your software's performance thoroughly and easily. Software-only simulators are best used to examine algorithm performance and accuracy, in a situation in which you don't need or care about the hardware. Emulators focus more on I/O and internal peripherals operating in the real world. You will need access to at least an emulator. We bring it up now because tool selection is tied to the hardware design process and processor selection.

Finally, placing a prototype device within a testing harness provides the most accurate proof of working software.

Results

Our design will have an LCD panel. With this capability, the system can write debug messages to the display. These can include a "splash screen" on power-up, echoed keystrokes, or displayed status messages.

The compiler must help in debugging. The generated assembly code needs to be available for inspection.

Product choices should favour emulators that can perform source-level debugging, matching the currently-executing machine code with the original C. For a thermostat, speed of emulation is not a critical factor; the only time-dependent function is the real-time clock.

A test harness made up of a lightbulb and fan, switched by the controller and pointed at the thermistor, is the simplest effective solution.

Chapter 3—
Microcontrollers In-depth

This section reviews microcontroller features and outlines the options available in the 8-bit microcontroller market. Some of the features you are used to seeing in central processors, such as graphics enhancements or floating point support, are nonexistent here.

The most engrossing and charismatic part of computer hardware design is the choice of the central processing unit. In the desktop world, processor choices revolve around compatibility with the Intel x86 product line: those compatible with Intel, those nearly compatible, and those completely divergent from it.

There is little such consistency in the embedded world, especially when talking about a new design. The 8-bit controller market is very competitive, largely because of the focus on volume. There is usually no brand name recognition; consumer product manufacturers want to protect users from technical details. If users do care about the chip that drives their product, they are probably seeking to surpass its intended use.

The 8-bit microcontrollers are not as programmer-friendly as 32-bit processors. Latter-day enhancements to a highly-optimized architecture, like extra ROM address space, can quickly outstrip an 8-bit's architectural limitations. This in turn forces processor designers to add in kludges such as bank switching or restrictions on addressing to compensate.

Finally, factors such as the life expectancy of the architecture should be considered. Using a C compiler for generating device programming reduces the cost of changing controllers when the preferred choice reaches the end of its product life cycle.

An 8-bit microcontroller has all of the traditional functional parts of a computer.

Central Processing Unit (CPU) The arithmetic and logic units of microcontrollers are restricted and optimized for the limited resources present in such small architectures. Multiply and divide operations are rare, and floating-point is nonexistent. Addressing modes are restricted in sometimes infuriating ways.

ROM and RAM The 8-bit microcontrollers rarely address more than 16 lines (64Kb) of ROM and RAM. If a chip's package exposes address or data buses at all, they provide only several kilobytes of addressing space. Most often, MCUs (Microcontroller Units) contain small internal RAM and ROM arrays. Because of the requirement to program the individual chips, ROM is often available as electrically-programmable (or electrically-erasable) memory.

Timer Two kinds are common: counters and watchdog timers. Simple counters can respond to a clock cycle or an input signal. Upon reaching a zero-point or a preset threshold, they can trigger an interrupt.

Interrupt Circuitry Where a general-purpose microprocessor would have multiple generalized interrupt inputs or levels, a microcontroller has interrupt signals dedicated to specific tasks: a counter time-out, or a signal change on an input pin.

That is, if the controller has interrupts at all. There is no guarantee that designers will include them if the intended applications are simple enough not to need them.

Input and Output Most chips supply some I/O lines that can switch external equipment; occasionally these pins can sink heavy current to reduce external components. Some varieties provide A/D and D/A converters or specialized logic for driving certain devices (like infrared LEDs).

Peripheral Buses Parallel peripheral buses reduce the "single-chip" advantage, so they are discouraged. Because speed is not at the top of the

list in embedded systems design, several competing standards for serial peripheral buses have evolved. Using only one to three wires, these buses permit external peripheral chips, such as ROMs, to interface with the microcontroller without monopolizing its existing interface lines.

The main consequence of the microcontroller's small size is that its resources are proportionally limited compared to those of a desktop personal computer. Though all the qualities of a computer are there — RAM, ROM, I/O and a microprocessor — the developer cannot count on having 8 bits in an I/O port, for example.

Before settling on the perfect processor, you must consider the external development tools available for your target. An embedded system is not self-hosting, like a personal computer. To develop embedded software, your development tools must run on a desktop computer, and use at least some very specialized hardware.

3.1— The Central Processing Unit (CPU)

The number and names of registers vary among microcontrollers. Sometimes they appear within a memory address space, and sometimes they are completely separate. Certain registers are common to most microcontrollers, although the names may vary.

- The **accumulator**
- The **index register**
- The **stack pointer**
- The **program counter**
- The **processor status register**

Direct access to the accumulator and index register in C is only occasionally desirable. The C `register` data type modifier amounts to a "request" for direct access to a register: the compiler may not actually use a register if it cannot do so optimally.

When it is desirable or necessary, however, another type of declaration can link a variable name with a register itself. The Byte Craft compiler provides the `registera` type (and equivalents for other registers). Assignment to a `registera` variable generates a load into the accumulator register, but does not generate a store into memory. Evaluation of the identifier returns the value in the register, not a value from memory.

```
registera important_variable = 0x55;
```

Direct access to the stack pointer or program counter is even less desirable. The whole point of using C is to abstract the program logic from direct machine language references. Function calls and looping, which will even out device-dependent stack manipulation and branching, are the best ways to structure your code. If necessary, use the C `goto` keyword with a labelled target: the compiler will insert the appropriate jump instruction and, most importantly, take care of any paging or setup automatically.

3.1.1— Instruction Sets

Where machine instructions for multiply, divide, table lookup, or multiply-and-accumulate are expected on general purpose MPUs (Microprocessor Units), their 8-bit equivalents do not always appear on each variant of a controller family.

A `#pragma` statement can inform the compiler that the target chip does have a certain optional instruction feature, and that it can therefore optimize code that will benefit from the instruction. These examples are present in the header file of the MC68HC05C8.

Listing 3.1 Instruction set configuration

```
#pragma has MUL;
#pragma has WAIT;
#pragma has STOP;
```

3.1.2— The Stack

If your processor supports a **stack** in general memory, the space required to record the stack is allocated from RAM that would otherwise be used for global variables. Not all stacks are recorded in main (or data) memory: the Microchip PIC and Scenix SX architectures use a stack space outside of user RAM.

It is important to check the depth of return information stored by function calls and interrupts. The compiler may report stack overflow (meaning that your stack is too small), but your stack declaration may be larger than necessary as well.

Beyond declaring an area as reserved for the stack, there is little else to worry about. Consider the following stack from the Motorola MC68HC705C8. The stack is 64 bytes from address `00C0` to `00FF`.

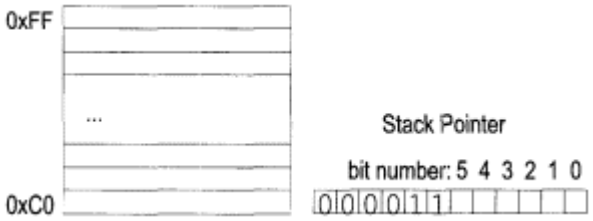

Figure 3.1
MC68HC705C8 stack

This is the required declaration in C.

```
#pragma memory stack [0x40] @ 0xFF;
```

Because stack sizes and configuration will change between processor families (or even between variants within the same family), the declaration makes the compiler aware of exactly how much space is available. Should you not need 64 bytes, you can reduce the size from 0x40 to a smaller number.

The compiler can provide information on the depth of function calling. See the `CALLMAP` option in Section 9.6, Debugging by Inspection.

3.2— Memory Addressing and Types

Most small microcontrollers provide very little RAM. The feeling of claustrophobia caused by absolutely running out of RAM or ROM is novel for desktop application programmers. Beyond the cursory check for failed memory allocations, programmers can rely on megabytes of RAM and swap files to almost always avoid out-of-memory errors.

The C compiler assists by reusing memory, wherever possible. The compiler has the patience to determine which locations are free at any one time, for reuse within multiple local **scopes.** "Free", of course, means not intended to be read by a subroutine until reinitialized by the next function call.

You will find that some typical programming techniques overwhelm the capacity of 8-bit microcontrollers because of memory concerns. Reentrant or recursive functions, gems of programming in desktop systems, assume abundant stack space and are practically impossible.

3.2.1— RAM and ROM

RAM and ROM are very permanently divided on a microcontroller. They may be part of different address spaces.

Controllers with anything less than the full complement of RAM or ROM (most of them) leave parts of the address space unimplemented. Instruction fetches or reads or writes to those areas can have unintended or erroneous results.

Declaring available RAM and ROM instructs the compiler where it is safe to place programming or data. The Byte Craft compiler requires all memory resources to be declared. The declarations can simply declare the type, size, and location of available memory, or they may optionally assign the area a symbolic name.

Named address spaces give you some control over the optimization process. If your processor has faster access to a portion of memory (page 0 on the 680x, for instance), and you have a particular scheme in mind, you can declare your variables as being in that memory area.

Listing 3.2 Declaring in named address space

```
#pragma memory ROM [0x4000] @ 0xA000;
#pragma memory RAM page0 [0xFF] @ 0x00;
#pragma memory RAM page1 [0xFF] @ 0x100;


/* ... */


/* my_ariable will appear in page0. If the processor has special
instructions to access page0, the compiler should generate them for
the assignment and later references */


int page0 my_variable = 0x55;
```

3.2.2— ROM and Programming

Programmable ROM, or PROM, started as an expensive means to prototype and test application code before making a masked ROM. In recent years, PROM has gained popularity to the point at which many developers consider it a superior alternative to a masked ROM in a mass production part.

As microcontroller applications become more specialised and complex, needs for maintenance and support rise. Many developers use PROM devices to provide software updates to customers without the cost of sending out new hardware.

The categories of programmable ROM are described in the following text.

Fused ROM is the traditional PROM, with ROM cells that are programmed by selectively blowing fuses in a memory matrix, according to bit patterns. Programmable only by external equipment.

EPROM (Erasable Programmable ROM) is nonvolatile and is read only. It must be erased by exposure to ultraviolet radiation.

EEPROM (Electrically Erasable Programmable ROM) devices have a significant advantage over EPROM devices, as they allow selective erasing of memory sections. The most common use for EEPROM is recording and maintaining configuration data vital to the application. For example, modems use EEPROM storage to record the current configuration settings.

Flash Memory is an economical compromise between EEPROM and EPROM technology. Your product can have a ROM-based configuration kernel, and application code written into flash memory. When you want to provide the customer with added functionality or a maintenance update, the hardware can be reprogrammed on site without installing new physical parts. The hardware is placed into configuration mode, which hands control to the kernel written in ROM. This kernel then handles the software steps needed to erase and rewrite the contents of the flash memory.

Depending upon the target part, EEPROM and Flash are programmable under program control. The programming process takes some time, as the electronics must wait for charge transfer and work slowly to avoid overheating the device.

3.2.3— von Neumann Versus Harvard Architectures

von Neumann architecture has a single, common memory space in which both program instructions and data are stored. There is a single internal data bus that fetches both instructions and data.

Harvard architecture computers have separate memory areas for program instructions and data. There are two or more internal data buses, which allow simultaneous access to both instructions and data. The CPU fetches program instructions on the program memory bus.

Programmers need not dwell upon which architecture they write for. C compilers should compensate for most of their respective drawbacks and quirks. Some of the more common characteristics are explained here as an insight into the code generated by compilers.

• Code generation for von Neumann-archtecture machines often takes advantage of the fact that the processor can execute programs out of RAM. Operations on certain data types may actually prime RAM locations with opcodes, and then branch to them!

• Since Harvard machines have an explicit memory space for data, using program memory for data storage is trickier. For example, a data value declared as a C constant must be stored in ROM as a constant value. Some chips have special instructions allowing the retrieval of information from program memory space. These instructions are always more complex or expensive than the equivalent instructions for fetching data from data memory. Others simply do not have them; data must be loaded by the side effect of a return instruction, for instance.

3.3— Timers

A **timer** is a counter that is incremented or decremented at the fixed rate of a clock pulse. Usually, an interrupt signals the completion of a fixed interval: the timer has counted to 0, has overflowed to 0, or has reached a target count.

Timers are a very competitive feature in microcontrollers. Timers or timing units of increasing sophistication and intelligence are readily available. The different types of timers available give the engineer lots of room to manoeuvre.

Programming the prescalar and starting the clock are tasks of the software developer. Knowing the processor clock frequency, and choosing correct prescalar values, you can achieve accurate timer clock periods.

The programmer's interface to a timer is several named control registers, declared with `#pragma port` statements and read or written as variables.

If a timer interrupt is available, it can be declared with a `#pragma vector` statement, and serviced by an associated interrupt service routine, written as a function.

Listing 3.3 Timer registers and interrupt handler

```
#pragma portr TIMER_LSB @ 0x24;
#pragma portr TIMER_MSB @ 0x25;

#pragma vector TIMER_IRQ @ 0xFFE0;

void TIMER_IRQ(void) {
       /* IRQ handler code */
}
```

3.3.1— Watchdog Timer

A **COP** (computer operating properly) or **watchdog** timer checks for runaway code execution. In general, watchdog timers must be turned on once within the first few cycles after reset. Software must then periodically reset the watchdog during execution.

If processor execution has gone off the track, it is unlikely that the watchdog will be reset reliably. It is this exact state that needs to be fixed: an indirect jump to an unexpected address could be the cause. A loop polling for external signals that are never received is also a possible cause.

The watchdog timeout can cause the processor to go to a known state, usually the RESET state, or to execute an interrupt. The hardware implementation of watchdog timers varies considerably between different processors. Some watchdog timers can be programmed for different time-out delays.

In C, the sequence to reset the watchdog can be as simple as assigning to a port.

Listing 3.4 Resetting the watchdog

```
#pragma portw WATCHDOG @ 0x26;
#define RESET_WATCHDOG() WATCHDOG = 0xFF

void main(void) {
  while(1) {
    /* ... */
    RESET_WATCHDOG();
  }
}
```

3.3.2— Examples

The following are some sample configurations.

• National Semiconductor's COP8SAA7 has a 16 bit timer called T1, a 16 bit idle timer called T0, and a watchdog timer. The idle timer T0 helps to maintain real time and low power during the IDLE mode. The timer T1 is used for real time controls tasks with three user-selectable modes.

• The Motorola MC68HC705C8 has a 16-bit counter and a COP watchdog timer. The COP watchdog timer is user-enabled, has selectable time-out periods, and is reset with two write instructions to the COPCR register. Interestingly, the COP watchdog is dependent upon the system clock; a clock monitor circuit resets the MCU if the clock stops, and thereby renders the COP watchdog useless.

• The Microchip PIC17C42a has four timer modules called TMR0, TMR1, TMR2, and TMR3, and a watchdog timer. TMR0 is a 16-bit timer with programmable prescalar, TMR1 and TMR2 are 8-bit timers, and TMR3 is a 16-bit timer.

3.4— Interrupt Circuitry

Microcontrollers usually provide hardware (signal) interrupt sources, and sometimes offer software (instruction) sources. In packages with restricted pin counts, IRQ signals may not be exposed or may be multiplexed with other I/O signals.

Interrupts that can be disabled are **maskable;** those which you cannot disable are **nonmaskable** interrupts. For example, RESET is nonmaskable; regardless of the code currently executing, the CPU must service a RESET interrupt.

Interrupt signals are asynchronous: they are events that can occur during, after, or before an instruction cycle. The processor can acknowledge interrupts using one of two methods: **synchronous** or **asynchronous** acknowledgement.

Most processors acknowledge interrupts synchronously: they complete the current instruction before dealing with the interrupt. In contrast, with asynchronous acknowledgement, the processor *halts* execution of the current instruction to service the interrupt. While asynchronous acknowledgement is more prompt than synchronous, it leaves open the possibility that the interrupt code will interfere with the instruction already in progress.

For instance, an interrupt routine updates a multi-byte value, which the main-line code reads regularly. Should the main-line code read that value in

a multi-byte fetch, and be interrupted part-way through, the loaded value becomes meaningless without any notice.

The code obeys our suggestion (Section 4.4.2, Interrupt Planning) about reading and writing variables one way, between interrupt and main-line code. To provide complete protection, the compiler needs to use **indivisible instructions,** or to disable interrupts temporarily, to protect the main-line code.

Synchronous acknowledgement is not a magic solution. This same problem affects processors with synchronous acknowledgement, when a multi-byte operation requires several instructions!

3.4.1— Vectored and Nonvectored Arbitration

There are two competing ways in which microcontrollers service interrupts. **Vectored arbitration** requires a table of pointers to the interrupt service routines. **Nonvectored arbitration** expects the first instructions of the ISR at a predetermined entry point. Most 8-bit microcontrollers use vectored arbitration interrupts.

When the compiler generates code for the interrupt service routine (ISR), it places the starting address in the appropriate interrupt vector within the ROM map, or relocates the code at the entry-point location in ROM. The compiler may also automatically generate arbitration code: remember to check for this when estimating ROM usage.

When an interrupt occurs, the processor will disable interrupts to prevent the service routine from being itself interrupted. A vectored machine then reads the address contained at the appropriate interrupt vector. It jumps to the address and begins executing the ISR code.

In contrast, a nonvectored system simply jumps to the known start location and executes what's there. The ISR may have to test each interrupt source in turn to implement priority, or to simply jump to a different location where the main body of the ISR resides.

Because of the extra handling in nonvectored systems, vectored interrupts are faster. In general, nonvectored ISRs are feasible for microcontrollers with less than five interrupts.

Table 3.1 shows the arbitration schemes of the major families of 8-bit microcontrollers.

Table 3.1 Interrupt arbitration schemes

Architecture	Arbitration	Notes
Motorola 6805/08	Vectored	Vectors at top of implemented memory.
National COP8	Mixed	See the text following this table.
Microchip PIC	Nonvectored	Some models do not have interrupts, and some provide vector dispatch for groups of interrupts.
Zilog Z8	Vectored	Priority setting required.
Scenix SX	Nonvectored	No priority levels.
Intel 8051	Nonvectored	Each interrupt jumps to a different, fixed, ISR entry point.
Cypress M8	Nonvectored	The processor **jumps to** a different, fixed, ISR entry point for each interrupt. These are called "vectors" and are two bytes long. A JMP instruction is required in these locations to jump to the ISR proper.

The National Semiconductor COP8 uses a mixed scheme. All interrupts branch to a common location in a nonvectored manner. At that location, the code must either execute the `VIS` instruction, which arbitrates among active interrupt sources and jumps to an address from a vector table, or poll the system for the interrupt condition explicitly and handle it in a user-defined manner. The latter method may be useful, but has many disadvantages.

Table 3.2 shows the COP8 vector table, as required for the COP8SAA7 device. The rank is as enforced by the VIS instruction.

Table 3.2 COP8 vectored interrupts

Rank	Source	Description	Vector Address *
1	Software	INTR Instruction	`0bFE - 0bFF`
2	Reserved	Future	`0bFC - 0bFD`
3	External	G0	`0bFA - 0bFB`

(table continued on next page)

(table continued from previous page)

Rank	Source	Description	Vector Address *
4	Timer T0	Underflow	0bF8 - 0bF9
5	Timer T1	T1A/Underflow	0bF6 - 0bF7
6	Timer T1	T1B	0bF4 - 0bF5
7	MICROWIRE/PLUS	BUSY Low	0bF2 - 0bF3
8	Reserved	Future	0bF0 - 0bF1
9	Reserved	Future	0bEE - 0bEF
10	Reserved	Future	0bEC - 0bED
11	Reserved	Future	0bEA - 0bEB
12	Reserved	Future	0bE8 - 0bE9
13	Reserved	Future	0bE6 - 0bE7
14	Reserved	Future	0bE4 - 0bE5
15	Port L/Wakeup	Port L Edge	0bE2 - 0bE3
16	Default	VIS Instruction Execution without any interrupts	0bE0 - 0bE1

* b represents the Vector to Interrupt Service routine (VIS) block. VIS and the vector table must be within the same 256-byte block. If VIS is the last address of a block, the table must be in the next block.

3.4.2— Saving State during Interrupts

On all chips, the interrupt process saves a minimal **processor state** of the machine, usually the current program counter. This is done to ensure that after an interrupt is serviced, execution will resume at the appropriate point in the main program.

Beyond this, machine state preservation varies widely. In any case, it is up to the programmer to provide code that saves as much extra state as is necessary. Usually, each interrupt handler will do this before attempting anything else. The location and accessibility of the saved state information varies from machine to machine.

Table 3.3 Processor state preservation during interrupts

Architecture	Interrupt Stacking Behaviour
Motorola 6808	All registers, except high byte of stack pointer, are automatically saved and restored.
Motorola 6805	All registers are automatically saved and restored.
National' COP8	Program counter is pushed.
Microchip PIC	Program counter is pushed.
Zilog Z8	PC and flags are pushed.
Scenix SX	PC is pushed, other registers are shadowed.
Cypress M8	PC and flags are pushed on the program stack.

Many C compilers reserve some locations in data memory for internal uses, such as pseudo-registers. Your compiler documentation should outline what code you must write to preserve the information located in these memory blocks. If your compiler creates a pseudo-register for 16-bit math operations, and your interrupt handler does not perform 16-bit operations that alter this pseudo-register, then you probably won't need to preserve its state.

3.4.3— Executing Interrupt Handlers

To minimize the possibility of an interrupt routine being itself interrupted, the microcontroller will disable interrupts while executing an interrupt handler.

Masking interrupts manually is useful during timing-critical sections of main-line code. The possibility of doing this is determined by your design; implementing it in C is easy. It doesn't take much more effort to generalize the procedure, either.

For the Byte Craft compilers, some simple macros in a header file can create the appropriate instructions. This code uses symbols defined by the compiler itself to choose the appropriate instructions.

Listing 3.5 Cross-platform interrupt control instructions

```
#ifdef CYC
#define IRQ_OFF() #asm < DI>
#define IRQ_ON()  #asm < EI>
#endif
```

```
#ifdef COP8C
#define IRQ_OFF() PSW.GIE = 0
#define IRQ_ON()  PSW.GIE = 1
#endif

#ifdef C6805
#define IRQ_OFF() CC.I = 0
#define IRQ_ON()  CC.I = 1
#endif
```

3.4.4— Multiple Interrupts

One some machines, the CPU first fetches and executes a program instruction, and then checks for pending interrupts. This guarantees that no matter how many interrupts queue up, the machine will always step through program code: no more than one interrupt handler will execute between each main program instruction.

On most machines, the CPU will check for interrupts before performing the next instruction fetch. As long as the controller detects a pending interrupt, it will service the interrupt before fetching the next instruction. This means it is possible to halt the main-line program by continuously sending interrupts. On the other hand, it guarantees that an interrupt is serviced before any more main program code is executed. This information is important for debugging: it can help explain why main-line software will not respond.

How does the CPU decide which interrupt to service first? A hardware priority level should determine this if two interrupts are signalled at the same time.

3.4.5— RESET

Some simple chips support no interrupts except a RESET sequence. If its intended applications require only a simple polling loop, or accept no input at all, there is no need for the extra hardware.

The only universal interrupting signal is RESET. A RESET can occur because of:

• initial power-on;

• a manual reset (signal on an external RESET pin);

• a watchdog time-out;

• low voltage, if your part supports power supply monitoring; or

• an instruction fetch from an illegal or unimplemented address, if your part implements protection against this.

The RESET interrupt prompts the chip to behave as if the power has been cycled. Since it does not *actually* cycle the power to the chip, the contents of volatile memory, I/O ports, or processor registers remain intact.

Taking advantage of this is tricky, but possible. If the compiler supports a user-written **initialization function,** you can check for particular values in memory, and decide to load default values or not. This can be used to check if the RESET was cold (power was cycled — use defaults) or warm (power was not cycled: preserve unaffected data).

There are conditions that upset this strategy. In the case of watchdog time-out, the data is electrically valid (the same as before watchdog RESET) but logically questionable.

3.5— I/O Ports

Input/output signals allow the microcontroller to control and read relays, lamps, switches, or any other discrete device. More complex components, such as keypads, LCD displays, or sensors, can also be accessed through ports. In this section, we talk about programming standard I/O lines. More specialized peripheral devices like A/D converters and communication buses are dealt with in subsequent sections.

Ports usually consist of eight switchable circuits, arranged in byte-sized I/O data registers. If a port is capable of both input and output, it will also have an associated register that specifies which way the port (or each individual bit of the port) is to operate. On many devices, this register is called the DDR (Data Direction Register).

Ports often support **tristate** logic. Tristate adds a third useful configuration besides input and output: **high impedance.** High impedance mode is the state of being undefined or floating. It's as if the port isn't actually part of the circuit at that time.

Since microcontrollers are intended to replace as many devices as possible, ports often include extras, such as internal **pull-ups** or **pull-downs.** These electrical features provide some noise immunity.

Data direction, tristate control, and optional pull-ups or pull-downs are all at the control of the programmer. As with desktop computer systems,

ports and their control registers appear as memory locations or as special I/O registers.

The following are some sample port configurations.

• The COP8SAA7 has four bidirectional 8-bit I/O ports called C, G, L, and F, in which each bit can be either input, output, or tristate. The programming interface for each has an associated configuration register (determines how the port behaves) and data register (accepts data for or presents data from the port).

• The Motorola MC68HC705C8 has three 8-bit ports called A, B, and C that can be either inputs or outputs depending on the value of the DDR. There is also a 7-bit fixed input port called port D, which is used for serial port programming.

• The Microchip PIC16C74 has five ports: PORTA through PORTE. Each port has an associated TRIS register that controls the data direction. PORTA uses the register ADCON1 to select analog or digital configuration. PORTD and PORTE can be configured as an 8-bit parallel slave port.

Ports and their associated configuration registers are not RAM locations, and as such are not electrically the same. Either reading or writing to a port may be illegal or dangerous if not explicitly permitted by the manufacturer. The compiler can watch for improper reads or writes by specifying acceptable modes in the port declaration.

With the Byte Craft compilers, ports are declared to the compiler using `#pragma` statements.

```
#pragma portrw PORTA @ 0x00;
#pragma portw  PORTA_DDR @ 0x04;
```

The acceptable modes of use are specified with `portr` for reading, `portw` for writing, or `portrw` for both.

3.5.1— Analog-to-Digital Conversion

It is often necessary to convert an external analog signal to a digital representation, or to convert a digital value to an analog level. A/D or D/A converters perform this function.

The science behind conversion, and the competitive environment of some analog disciplines like automotive instrumentation or audio processing, ensures that there is a variety of approaches to conversion, with tradeoffs in accuracy, precision, and time.

Typically, the support routines for an A/D or D/A converter are prime candidates for packaging as a library of C functions. It is important to note that the conversion process may take some time.

The Byte Craft compiler will support this type of peripheral in two ways.

• You can declare the control ports with `#pragma port` in the device header file.

• You can declare an interrupt raised by the conversion peripheral with `#pragma vector` and service it with an ISR function. This is an intuitive way to handle conversions that take a long time.

Most microcontrollers use a **successive approximation converter** for A/D conversion. The converter works with one bit at a time from the MSB (Most-Significant Bit) and determines if the next step is higher or lower. This technique is slow and consumes a great deal of power. It is also cheap and has consistent conversion times.

The Microchip PIC16C74 has an A/D converter module that features eight analog inputs. These eight inputs are multiplexed into one sample-and-hold, which is the input into the converter.

A **single slope converter** appears in National Semiconductor's COP888EK. It includes an analog MUX/comparator/timer with input capture and constant current source. The conversion time varies greatly and is quite slow. It also has 14- to 16-bit accuracy.

A **flash converter** examines each level and decides the voltage level. It is very fast, but draws a great deal of current and is not feasible beyond 10 bits.

3.6— Serial Peripheral Buses

Single-chip microcontrollers of sufficient pin count can expose address, data, and control signals externally, but this negates the benefit of single-chip design.

There are several standards for *serial* peripheral communication, using one to three external wires to communicate with one or more peripheral devices.

Of course, serializing frequent ROM or RAM accesses impacts on execution speed. Serial peripherals are not accommodated within the addressing range of a processor, so serial program ROM is not possible.

The compiler can assist by making data access to serial peripherals more intuitive. The Byte Craft compilers provide the `SPECIAL` memory declaration. Using it, you can declare the registers or memory of the remote device

within the memory map **as the compiler understands it.** You then write device driver routines to read and write each `SPECIAL` memory area.

Accesses to variables or ports declared within the `SPECIAL` memory area receive special treatment. Reading the value of a `SPECIAL` variable executes the associated read routine, and the value returned is the result of the read. Assigning a new value to a `SPECIAL` variable passes the value to the associated write routine. The read and write routines can conduct peripheral bus transactions to get or set the variable value.

Bus standards and driver routines are prime targets for library implementation.

Table 3.4 Serial peripheral bus options

Standard	Manufacturer	Notes
I^2C	Philips	Synchronous serial peripheral interface that operates across two wires. The two lines consist of the serial data line and the serial clock line, which are both bidirectional. No programming interface is specified.
SCI	various	Enhanced UART for board-level serial communication. Asynchronous over two wires.
SPI	various	Synchronous serial peripheral interface that operates across 4 wires: SPI Clock (SCK), master-out-slave-in (MOSI), master-in-slave-out (MISO), and a slave select (SS). Manufacturers rebrand, or enhance, this standard. For instance, National Semiconductor offers MICROWIRE/PLUS devices that are similar (and possibly compatible).

3.7—
Development Tools for a Microcontroller

Developing software in C requires the use of a desktop computer to run the cross-compiler. From there, you can program and evaluate the target system in one of the following ways.

Manual Programming The developer programs an EEPROM microcontroller, and replaces it in the target for each testing iteration. This is time- and labour-intensive, but provides the most realistic testing environment. The results are not tainted by the presence of test instruments.

Simulators The developer loads object code into a software program that simulates the eventual environment. This arrangement is best suited for examining complex programming on the fly.

Emulators The developer substitutes the microcontroller (or an external chip like a program ROM) in the design with a special piece of hardware that emulates the device while providing a link to the development platform. A well-designed emulator does not appear any differently to the target system than a normal controller, but allows the user to spy into the controller's behaviour and to examine the target platform's hardware at the same time.

Development tools are a factor in processor choice. A compiler can generate information to link the original source with the object code that the simulator or emulator uses. Watch for products that are compatible with your compiler.

Chapter 4—
Design Process

The design process mirrors the problem specification, making concrete decisions about each general point raised previously.

4.1—
Product Functionality

We can mirror the product requirements, the user-oriented checklist of tasks that the product should perform, with some details about the device to be designed.

Results

• Program will measure current temperature. We will have to service and read an A/D converter connected to a thermistor. To minimize part count, the A/D converter will be quite rudimentary.

• Program will count real time on a 24-hour clock. With a one-second timer interrupt, we should be able to count minutes and hours. We won't bother with day/date calculations — no automatic daylight savings time adjustment, but no year calculation problems either!

• Program will accept current time settings and reset clock count. Library routines should help in translating internal clock representation with a displayable format.

• Program will accept and store user-selected heating and cooling temperature settings, and time settings for three daily usage periods. We will build in reasonable defaults, and then keep the current settings in RAM. If the power goes out, the device won't put anyone in danger.

• Program will compare current temperature with settings for current time period, and turn on or turn off external heat or cooling units as needed. This will require asserting an output line to actuate a relay, one for both heating and cooling.

• Program will refrain from changing state of external units twice within a short period of time to avoid thrashing. This means keeping a separate count of a five-second waiting period between switching operations. Immediate shut-off should override this count, however.

• Program will accept manual override at any time, and immediately turn off all active external units. Whether the keypad is polled or interrupt-driven, one or two keys for shutdown should be responded to immediately.

4.2—
Hardware Design

As mentioned previously, hardware is outside the scope of this book. We include this hardware information to justify the choices we make in the design of the thermostat.

The part of choice is the MC68705J1A, for its simplicity and small pin count. It has just enough pins to control all devices.

• 14 I/O pins, plus a disabled IRQ input.

• 8 pins (port a) for keypad.

• 2 pins (1 from port b, 1 from disabled IRQ input) for the thermistor.

• 7 pins (3 from port b, 4 from port a) for serial LCD panel.

• 2 pins (port b) for heating and cooling
switching.

The `j1a` is the only chip needed; the rest are discrete parts.

Once the hardware is settled, the task moves to designing your program.

4.3— Software Design

4.3.1— Software Architecture

As before, we will be using C.

Prepackaged libraries of functions for microcontrollers are available with C compilers for embedded targets, but they are nowhere near as common as those for the general-purpose computer programmer.

Libraries for microcontrollers should always be accompanied by their source code! Since safety of the final product becomes a real factor in applications like machine control, libraries must be as carefully inspected as the rest of the program.

To remain productive, your compiler and emulation environment should agree on a format for extended debugging information. This allows the emulator to perform source level debugging with your listing file.

While traditional, a linker is not strictly necessary.

The development environment is not discussed here in detail. A text on configuration management can best provide assistance on how to implement revision control and build automation, if either are necessary.

Results

The compiler will be the C6805 Code Development System from Byte Craft Limited. It generates Motorola, Intel, and part-proprietary binaries, and a listing file that places the generated assembly code beside the original source.

With the Byte Craft CDS, device-specific details are captured in a header file that uses common identifiers to represent them. Ensure that the device header file `05j1a.h` is present. When using an EEPROM part, use the file `705j1a.h`. To change the target part, simply change the header file.

Libraries to be used in the thermostat include the following.

stdio includes routines to get and put strings from displays and keyboards. This library relies on others to do the actual input and output.

lcd includes routines to clear the display, move the hardware cursor, and write characters and strings.

keypad includes routines to check for keypresses and decode keys.

port provides transparent access to the two parallel ports of the `j1a` part.

delay times communications with the LCD display, and debounces the keyboard.

We will also write one completely new library.

timestmp converts a seconds count into a human-readable time, and back.

A clock/counter interrupt calculates time, requests display update, and sets target temperatures. The main line implements a loop that updates the LCD, polls the keyboard, samples environment temperature, and switches the HVAC machinery.

4.3.2— Flowchart

Now we can add some concrete information to the flowchart for the algorithm. This in turn will help us lay out our source files.

Results

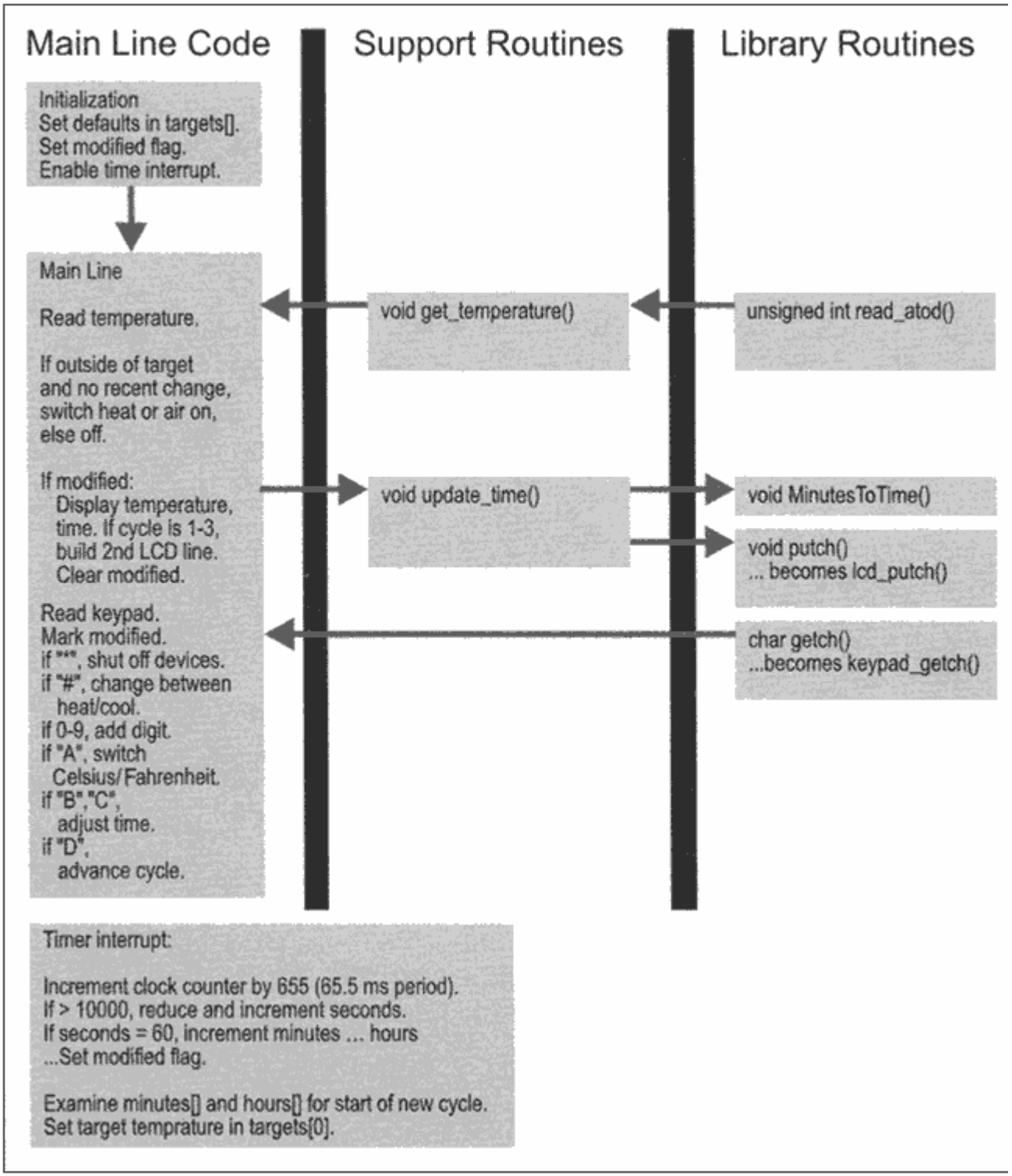

Figure 4.1
Data flow for the algorithm (revised)

4.4—
Resource Management

Now that we have some concrete information about the target platform, the development software, and the way data will flow between parts of the software, we can begin to nail down resource usage.

4.4.1—
Scratch Pad

Many C compilers use some available RAM for internal purposes such as pseudo-registers. An efficient C compiler will support *scratch pads* in data memory. A scratch pad is a block of memory that can be used for more than one purpose. A pseudo-register is a variable used as the destination for basic operations performed with larger data types. Your compiler documentation will detail the size and purpose of scratch pad allocations.

For example, if you attempt a 16-bit math operation on a chip with no natural 16-bit register, the compiler will dedicate a portion of RAM for 16-bit pseudo-registers that store values during math operations.

If the scratch pad allocation strains your memory budgeting, you can consider reusing the memory yourself. The only condition is that you must manage variable scope yourself.

For example, the Byte Craft compiler creates the 16-bit pseudo-index register `__longIX`. You can reuse this 16-bit location with the following statement.

```
long int myTemp @ __longIX;
```

Should you store a value in `myTemp`, and then make a library call, the library software must not perform any long operations or your data will be overwritten.

4.4.2—
Interrupt Planning

Unless you have delved into drivers or other low-level software development, you have probably been shielded from interrupts. Embedded C helps by providing an intuitive way to structure and code interrupt handlers, but there are some caveats.

• How will the main-line processor state be preserved? The processor registers might be saved automatically on a stack, or simply shadowed in hidden registers, by the processor. You might easily swap the main-line register values out if multiple banks of registers are available. As a last resort, you could save the register values manually, and restore them before returning from the interrupt.

The temporary registers used by compiler math functions also need to be preserved if calculations within the interrupt might obliterate them. Preserving these registers will require multi-byte transfer routines. The cost of these repetitive sequences within a frequently-called interrupt can add up.

• Will the tasks envisioned for the interrupt, including the previous save and restore operations, be completed in time? The frequency of the interrupt calls, and the amount of work to be done within them, need to be estimated.

If there is more than enough time to complete all operations, the speed of the processor could be reduced to gain electrical benefits.

• How will the interrupt routine and main-line code interact? Beyond protecting critical sections of the main line by disabling interrupts, there are broader synchronization conflicts to worry about, especially in global data.

One general rule is to write global variables in one place only — main line or interrupt code — and read them in the other. Make communication between the interrupt routine and main-line code travel one way if possible.

Results

The C6805 CDS creates a 4-byte scratch pad called `__SPAD`. It also creates two pseudo-registers for 16-bit operations. They are `__longAC` (2 bytes) and `__longIX` (4 bytes).

C6805 has support for local memory, so we can watch for economies in counter or temporary variable allocation.

The `j1a` a part has a software interrupt, which may be used by the compiler as a fast subroutine call. We won't use it explicitly. We will disable the IRQ input to use as a spare input pin.

The `j1a` also has a timer interrupt, which we will use to execute the time-keeping functions. The interrupt will run about every 65 milliseconds, so we will need to keep the following items.

A Millisecond Counter Actually, the millisecond counter needs an extra digit of accuracy to agree with the published specification, so we will keep tenths of a millisecond.

A Second Counter We will display time in minutes, so this is just for internal use.

A Counter for Hours and Minutes We will explain more on this later.

Since we will need the external IRQ pin as an extra input, we cannot use the keypad interrupt function associated with port A pins 0–3.

6805 interrupts cause the entire processor state to be preserved: accumulator, X register, PC, stack pointer, and condition codes. Therefore, we don't need to write code for this. We may need to preserve the pseudo-registers.

4.5— Testing Choices

4.5.1— Design for Debugging

With the processor selected, you can start to formulate a testing strategy. The processor may supply some help, in the form of a hardware debugging interface.

Designing the software by grouping it in libraries is a good organizational technique. You can then test each subsystem by writing small test programs that use one library apiece.

Modular testing solves an interesting quandary: a system with an LCD display can display human-readable status codes or other debugging messages. But until the LCD display itself is operational and reliable, it is of no help.

Focus directly on the configuration of the LCD display with a test program: it is one of the more complex "black box" devices, with a 4- or 8-bit interface, and enable, register-select, and read/write lines that must be signalled according to timing tolerances. In our design, it is cost-effective to multiplex the LCD data bus with the keypad. In your design, the LCD bus may be attached in even more complex ways. You may need a test program just to drive the library as you customize it for your hardware.

4.5.2— Code Inspection

When writing libraries, ensure they contain the following lines.

Listing 4.1 Library skeleton

```
#pragma library;
#pragma option +l;
/* . . . */
#pragma endlibrary;
```

This causes the compiler to omit generating code for any function not referenced from the main module, and to reproduce the library code within the listing file.

4.5.3— Execution within a Simulator Environment

Software-based simulators enjoy great flexibility as a test environment. Although not physical, they can be written or configured to match the specified programmer's model and hardware characteristics exactly.

When running on a contemporary PC, speed of simulation is not an issue: a PC running at hundreds of MHz can easily simulate events at the common MCU speeds of between 1 and 10 MHz.

4.5.4— Execution within an Emulator Environment

There is a tradeoff that appears with emulators: they provide a physical base for testing, but may not reproduce your specific physical configuration. They only present the success of the design to the extent that they implement it.

Emulator host software should accept a debugging file format. Byte Craft's `.COD` file is such a format. It includes extra information that would not normally be represented within the executable data, such as source code line numbers for each section of generated code.

With this extra information, emulators can coordinate breakpoints within the source or listing file. You can determine the context of the register values that the emulator host software reports.

4.5.5— Target System in a Test Harness

After prototype hardware has arrived, it makes sense to move candidate software to it as quickly as possible. The test harness can consist of simple components: switches, lights, small motors, or other simple indicators. It should replicate the connections (and any feedback conditions) of the working environment for which the unit is destined.

For the programmer, the challenge lies in understanding the difference between the test harness and the real world. Hopefully, you will not have to change important constants like prescalar values.

Results

For initial code inspection, we will use the C6805 listing file. The listing file includes numerous reports that are useful both during code-and-compile cycles, and when doing code review on others' work.

For an emulator, we will use the MC68HC705JICS product from Motorola. The emulator connects to a PC using a serial cable, and uses a 6805C8 to recreate I/O ports and communicate with the host system. The host system actually evaluates the `j1a` software. The emulator is non-real-time: commands to change port bits, for instance, must be transmitted by the PC to the JICS board.

For the thermostat, our test harness consists of the following.

• 30V lamps to represent heat and cool units when activated.

• Unit power and ground from a wall unit.

Chapter 5—
C for Embedded Systems

With a refined design in hand that takes into account the prospective hardware environment, you can begin coding. Starting to code an embedded project is not much different from coding a desktop application project.

Most significantly, the only software environment present is that which you establish, through device defaults, global declarations, and setup routines. The `main()` function is indeed the main function.

There are other practices that characterize embedded C development:

• in-line assembly language,

• device knowledge, and

• mechanical knowledge.

5.1—
In-line Assembly Language

While not required by ANSI C, most embedded development compilers provide a means of incorporating assembly language in C programs. One common way of accomplishing this is using preprocessor directives.

The Byte Craft compiler uses `#asm` and `#endasm` directives to signal assembly language code boundaries. Everything lying between the directives is processed by the macro assembler, which is built into the compiler.

The labels and variables used in C are available within included assembly, as well. However, the compiler will not attempt to optimize such code. The compiler assumes that the user has a good reason to avoid the compiler's code generation and optimization.

The microcontroller's manufacturer should provide assistance in hand-crafting assembly language programming. You may be required to flip opcodes out of order to accommodate a pipeline, something the compiler will do transparently.

The following two definitions of the `wait()` function show the function written in C and the equivalent function in Motorola 68HC705C8 assembly language.

Listing 5.1 C functions containing in-line assembly language

```
/* C function */


                                  void wait(int delay)
00EA                              {
0300 B7 EA    STA    $EA
0302 3A EA    DEC    $EA           while(--delay);
0304 26 FC    BNE    $0302
0306 81       RTS                }

/* Hand-written assembly version. Note: the code  to store parameters
and the return from the function are still generated. There's little
reason to change this: if you want to avoid using  a local variable,
consider declaring the parameter as (BCL) registera or registerx, or
another equivalent name */


                                  void wait2(int delay)
00EA                              {
0307 B7 EA     STA    $EA
                                  #asm
                                  LOOP:
0309 3A EA                          DEC delay;
030B 26 FC                          BNE LOOP;
                                  #endasm
030D 81        RTS                }
```

5.2— Device Knowledge

In the embedded world, one compiler, generating code for one controller architecture, must still support a potentially endless array of slightly different processors: parts with varying amounts of RAM and ROM, fewer or more ports, special features, and so on. Add to this the possibility of customized parts (with mask-programmed ROM routines, for instance).

The standard C environment allows the definition of compiler-specific extensions with the `#pragma` preprocessor directive. The preprocessor may deal with `#pragma` directives in your source code, or it may be the compiler that acts upon these directives.

The `#pragma` directive is used most commonly in embedded development to describe specific resources of your target hardware, such as available memory, ports, and specialized instruction sets. Even processor clock speed can be specified, if it matters to the compiler. The following sections describe `#pragma` directives needed by the Byte Craft compiler.

5.2.1— `#pragma has`

`#pragma has` describes specific architectural qualities of the processor. The qualifiers of the `#pragma has` instruction are dependent upon the processor family and the compiler.

Most `#pragma has` statements will appear in the device header file. The following examples show the difference between code compiled with `has MUL` enabled and disabled.

Listing 5.2 6805 multiplication without `#pragma has MUL`

```
                                void main(void)
                                {
00EB                            unsigned int result;
00EA                            unsigned int one;
00E9                            unsigned int two;

030E A6 17   LDA   #$17         one = 23;
0310 B7 EA   STA   $EA
0312 A6 04   LDA   #$04         two = 4;
0314 B7 E9   STA   $E9
```

```
0316 B6 EA     LDA     $EA          result = one * two;
0318 BE E9     LDX     $E9
031A CD 03 20  JSR     $0320
031D B7 EB     STA     $EB

031F 81        RTS                  }
0320 B7 ED     STA     $ED          /* multiplication subroutine */
0322 A6 08     LDA     #$08
0324 B7 EC     STA     $EC
0326 4F        CLRA
0327 48        LSLA
0328 59        ROLX
0329 24 05     BCC     $0330
032B BB ED     ADD     $ED
032D 24 01     BCC     $0330
032F 5C        INCX
0330 3A EC     DEC     $EC
0332 26 F3     BNE     $0327
0334 81        RTS
```

Listing 5.3?6805 multiplication with `#pragma has MUL` enabled

```
                                    void main (void)
                                    {
00EB                                unsigned int result;
00EA                                unsigned int one;
00E9                                unsigned int two;

030E A6 17     LDA     #$17         one = 23;
0310 B7 EA     STA     $EA
0312 A6 04     LDA     #$04         two = 4;
0314 B7 E9     STA     $E9

0316 B6 EA     LDA     $EA          result = one * two;
0318 BE E9     LDX     $E9
```

?

```
031A 42        MUL
031B B7 EB     STA     $EB

031D 81        RTS                 }
```

5.2.2— `#pragma port`

`#pragma port` directives describe the ports available on the target computer. This declaration reserves memory-mapped port locations, so the compiler does not use them for data memory allocation.

`#pragma port` directives indicate read or write access, or both. The electronics of I/O ports may sometimes forbid writing to them or even reading from them. The compiler can report undesirable accesses to a port if it finds a restriction in the declaration. Besides protecting the port register, the declaration allows you to provide a useful mnemonic name for the port. You can then use the name associated with the port to read or write its input or output state.

The following defines two ports and their associated data direction registers on the Motorola 68HC705C8.

Listing 5.4 Defining ports with `#pragma` directives

```
#pragma portrw PORTA   @ 0x0000
#pragna portrw PORTB   @ 0x0001;
#pragma portw  DDRA    @ 0x0004;
#pragma portw  DDRB    @ 0x0005;
```

The compiler is informed that two ports are available. The name `PORTA` refers to physical port A's data register, which is available for reading and writing and is located at address `0x0000.` The name `DDRA` refers to physical port A's data direction register, which is available for writing only and is located at address `0x0004.`

It is then possible to write the value `0xAA` (alternate bits high) to the port using the C assignment syntax.

Listing 5.5 Setting ports using assignment

```
DDRA=0xFF; /* set the direction to output */
PORTA=0xAA; /* set the output pins to 10101010 */
```

The resources for a specific part are best described through a header file that is brought in using `#include`. ANSI C has one prescribed rule about `#pragma` directives: if a `#pragma` directive is not recognised, the compiler ignores it. This ensures that unknown `#pragma` directives will not affect your code.

5.2.3— Endianness

One piece of device knowledge that the programmer must keep in mind is the **endianness** of the processor. C does not deal directly with endianness, even in multi-byte shift operations.

In cases in which you will directly manipulate part of a multi-byte value, you must determine from manufacturer's information whether the high byte (big end) or low byte (little end) is stored first in memory.

With the restricted resources of microcontrollers, some quirks appear. The COP8 architecture stores addresses in memory (for indirect operations) as big-endian, and data as little-endian. Addresses pushed on to the stack do not appear in the same endianness as they do in registers or in RAM.

Compilers, when building their symbol tables, normally use the lowest (first) memory location to record the location of an identifier, regardless of the endianness of the processor.

5.3— Mechanical Knowledge

Techniques used in an embedded system program are often based upon knowledge of specific device or peripheral operation. Modern operating system APIs are designed to hide this from the application developer. Embedded C systems need first-hand control of peripheral devices, but can still provide a healthy level of generalization.

One useful technique employed by the port library is to define the letters I and O to the appropriate settings for port control registers that govern **data direction**. The letters cannot be defined individually. They are defined in eight-letter sequences that are unlikely to appear elsewhere.

Applications may need to use a port both as input and output (for instance, driving a bidirectional parallel port through software), and setting a port's data direction using these macros provides device independence.

Listing 5.6 Device-independent data direction settings

```
#pragma portw DDR @ 0x05;

#include <port.h>
/* port.h contains numerous definitions such as the following:

   #define IIIIIIII 0b00000000
   #define IIII0000 0b00001111
   #define 00000000 0b11111111

   where 'O'utput sets DDR bits to one ('1')
   and 'I'nput sets DDR bits to zero ('0').
   They can be regenerated for the opposite settings.
*/

/* ... later ... */

DDR = 00000000; /* all bits set for output */
DDR_WAIT();
/* ... perform write to port ... */
DDR = IIIIIIII; /* all bits set for input */
DDR_WAIT();
/* ... perform read of port ... */
```

Low power operation can be achieved by repeatedly putting the processor in an inactive mode until an interrupt signals some event. Processor families provide variations on the STOP or WAIT operation, with different provisions for protecting the contents of processor registers and recovery times. C duly expresses these as `STOP()` or `WAIT()` macros. If a hardware stop was not available, the macro could be redefined to cause an infinite loop, jump to the reset vector, or perform another substitute operation.

When a button is pressed, it "bounces", which means that it is read as several quick contact closures instead of just one. It is necessary to include **debouncing** support to ensure that one keypress is interpreted out of several bounces. When a first keypad switch is registered on a port, software can call the `keypad_wait()` function to create a delay, and then check the button again. If the button is no longer in a pushed state, then the push is interpreted as a bounce (or an error), and the cycle begins again. When the signal

is present both before and after the delay, it is likely that the mechanism has stopped bouncing and the keypress can be registered.

5.4— Libraries

Libraries are the traditional mechanism for modular compile-time code reuse. C for embedded systems can make use of libraries as an organizational tool.

• As usual, a library is a code module that has no `main()` routine.

• The associated header file should declare the variables and functions within the library as `extern`.

• The linking process is simpler than that for desktop software development. There is no need to archive object files, and there is no dynamic linking to worry about.

• It is unacceptable in embedded software for unreferenced functions to be left in the object file during linking. In the Byte Craft compiler, the `#pragma library` and `#pragma endlibrary` bounding statements identify that not all routines within a library need to be linked in. The ROM space saved is worth the extra effort on the part of the compiler to extract only referenced routines.

• Peering into the code generated for libraries is as important as seeing the code for the main module. The statement `#pragma option +l;` within a library causes the compiler to add the source and assembly code from the library into the listing file of the final program.

5.5— First Look at an Embedded C Program

Traditionally, the first program a developer writes in C is one that displays the message "Hello World!" on the computer screen.

In the world of 8-bit microcontrollers, there is no environment that provides standard input and output. Some C compilers provide a `stdio` library, but the interpretation of input and output differs from that of a desktop system with pipes and shell environments.

The following introductory program is a good "Hello World!" equivalent. The program tests to see if a button attached to a controller port has been pushed. If the button has been pushed, the program turns on an LED attached to the port, waits, and then turns it back off.

Listing 5.7 A "Hello World!" for microcontrollers

```
#include <hc705c8.h>
/* #pragma portrw PORTA @ 0x0A; is declared in header
   #pragma portw  DDRA  @ 0x8A; is declared in header */
#include <port.h>
#define ON 1
#define OFF 0
#define PUSHED 1

void wait(registera); /* wait function prototype, not displayed */

void main(void){
  DDRA = IIIIIII0; /* pin 0 to output, pin 1 to input,
                      rest don't matter */
  while (1){
    if (PORTA.1 == PUSHED){
      wait(1);                /* is it a valid push? */
      if (PORTA.1 == PUSHED){
       PORTA.0 = ON;              /* turn on light */
        wait(10);                 /* short delay */
        PORTA.0 = OFF;            /* turn off light */
      }
    }
  }
} /* end main */
```

Chapter 6—
Data Types and Variables

Due to the restricted environment of embedded controllers, standard C variables and data types take on new characteristics.

The most drastic change takes the **default integer** type to 8 or 16 bits. While quite acceptable from a C point of view, programmers used to inexpensive 32-bit values need to adjust to the new environment. By default, the Byte Craft compiler creates 8 bit `ints`, while a `long` or `long int` data type is two bytes in size.

Embedded compilers expose standard C types, and several additional data types that are appropriate for embedded development. The embedded world brings a new aspect to **type conversion,** too. Casting is one task that is made easier by the compiler, but casting can more readily lose information and interfere with values destined for use in a context such as peripheral control.

The other substantial change involves data types and variables with important **side effects.**

• Constants or initialized variables will consume a more significant proportion of ROM, as well as RAM. Global variable declarations that contain an initialization will automatically generate machine code to place a value at the allocated address shortly after reset. In the Byte Craft com-

piler, one or more global variable initializations will generate code to zero all variable RAM before assigning the initialization values to the variables.

• Variables of type `register` are available, but the scarcity of registers in the typical 8-bit architecture makes them more volatile than usual.

• In the Byte Craft compiler, a simple assignment to or evaluation of a variable declared to be within a `SPECIAL` memory area can generate a subroutine call. The driver subroutine that reads or writes the value can take significant time to execute if it is communicating with an external device.

Beyond the built-in types, programmers can define their own custom types, as usual.

When the compiler comes across a **variable declaration,** it checks that the variable has not previously been declared and then allocates an appropriately-sized block of RAM. For example, a `char` variable will by default require a single word (8 bits) of RAM or data memory. **Data type modifiers** influence the size and treatment of the memory allocated for variables.

Storage modifiers affect when memory is allocated and how it is considered free to be re-used.

• Some variables are meant to be allocated once only across several modules. Even previously-compiled modules may need to access a common variable. The compilation units — libraries or object files — must identify these as **external symbols** using the `extern` storage class modifier.

• Non-`static` variables that are of mutually-exclusive **scope** are likely to be overlaid. Embedded C regards scope in much the same way that standard C does, but there is an extra effort to use scope to help conserve memory resources.

• The compiler will reinitialize local variables, if appropriate, on each entry into the subroutine. These variables are deemed to be declared as `auto`. Local variables declared as `static` are left alone at the start of the function; if they have an initial value, the Byte Craft compiler assigns it *once*, in the manner of a global initialization.

Embedded-specific interpretations of each of the C data type and storage modifiers are shown in Table 6.1.

Table 6.1 Data type modifiers and notes

Modifier	Notes
`auto`	Unnecessary for local variables. Compare with `static`.
`const`	Allocates memory in ROM.
`extern`	Flags the reference for later resolution from within a library.
`far`	Depends upon addressing scheme of target.
`near`	Depends upon addressing scheme of target.
`signed`	Generates extra code compared with `unsigned`.
`static`	Preserves local variable across function calls.
`unsigned`	Creates significant savings in generated code.
`volatile`	(No specific notes; consult the ISO standard for more information)

6.1— Identifier Declaration

An embedded C compiler uses C declarations to allocate memory for variables or functions.

As the compiler reads a program, it records all identifier names in a symbol table. The compiler uses the symbol table internally as a reference to keep track of the identifiers: their name, type, and the location in memory that they represent. Most compilers support identifier names of at least 31 characters.

It is sometimes necessary or desirable to direct the placement of variables. The Byte Craft compiler interprets the `@` operator and a number following the identifier as the location at which the variable value should be stored. The `@` operator is also used to associate port registers with identifiers in `#pragma port` statements. These identifiers occupy the same name space as RAM and ROM memory variable identifiers.

6.1.1— Special Data Types and Data Access

Every bit of RAM is precious. Even if unused RAM on a peripheral device is not within the immediate address space of the processor, subtle techniques can make it appear to be. Declaring a memory space as `SPECIAL` requires you to write routines to read and write data to and from the peripheral. The tradeoff is with performance.

Listing 6.1 `SPECIAL`: memory, driver method, and variable declarations

```
#pragma memory SPECIAL eeprom [128] @ 0x80;
#define eeprom_r(LOC) I2C_read(LOC)
#define eeprom_w(LOC,VAL) I2C_write(LOC,VAL)


int eeprom i;
```

Accessing the variable declared to be within the special memory area will take some time, but the compiler will allow the process to be transparent.

6.2— Function Data Types

A **function data type** determines the value that a subroutine can return. For example, a function of type `int` returns a signed integer value.

Without a specific return type, any function returns an `int`. An embedded C compiler provides for this even in the case of `main()`, though returning is not anticipated. To avoid confusion, you should always declare `main()` with return type `void`.

Some other specially-named functions will have predetermined types; those that implement interrupt coding, for example, will be of type `void` unless there is some method for an interrupt to return a value. The Scenix SX returns a value to support virtual peripherals, and so its interrupt handler will have a function data type of `int`.

Parameter data types indicate the values to be passed in to the function, and the memory to be reserved for storing them. A function declared without any parameters (i.e., with empty parentheses) is deemed to have no parameters, properly noted as `(void)`.

The compiler allocates memory differently depending upon the target part. For instance, the Byte Craft compiler passes the first two (byte-sized) parameters through an accumulator and another register in the processor. If local memory is specifically declared, the compiler will allocate parameter passing locations out of that space.

6.3— The Character Data Type

The C character data type, `char`, stores character values and is allocated one byte of memory space. The most common use of alphabetic information is

output to an LCD panel or input from a keyswitch device, where each letter used is indicated by a character value.

6.4— Integer Data Types

Integer values can be stored as `int`, `short`, or `long` data types. The size of `int` values is usually 16 bits on 8-bit architectures. The Byte Craft compiler's default `int` size is switchable between 8 and 16 bits.

The `short` data type helps compensate for varying sizes of `int`. On many traditional C platforms, the size of an `int` is more than two bytes. On platforms in which an `int` is greater than two bytes, a `short` should be two bytes in size. On platforms in which an `int` is one or two bytes in size — most 8-bit microcontrollers — the `short` data type will typically occupy a single byte.

Should your program need to manipulate values larger than an `int`, you can use the `long` data type. On most platforms the `long` data type reserves twice as much memory as the `int` data type. On 8-bit microcontrollers, the `long` data type typically occupies 16 bits.

It is important to note that `long` integer values are almost always stored in a memory block larger than the natural size for the computer. This means that the compiler must typically generate more machine instructions when a program uses `long` values.

`long` and `short` are useful because they are less likely to change between a target with a natural 8-bit data type and one that delves into 16-bit values. In cases of a switchable `int`, you can maintain code portability by using `short` for those values that require 8 bits, and `long` for values which require 16 bits.

Like the `int`, the `short` and `long` data types uses a sign bit by default and can therefore contain negative numbers.

6.4.1— Byte Craft's Sized Integers

The Byte Craft compiler recognizes `int8`, `int16`, `int24`, and `int32` data types. They are integers with the appropriate number of bits. These remove the ambiguity of varying or switchable integer sizes.

6.5— Bit Data Types

Embedded systems need to deal efficiently with bit-sized values.

ISO/IEC 9899:1999 specifies the `_Bool` type. Variables of type `_Bool` can hold a 0 or 1. This is a new addition to the C standard.

The Byte Craft compilers supply two types for bit-sized quantities: `bit` and `bits`. A `bit` value is a single independent bit, which the compiler places and manages depending upon the capabilities of the processor.

A `bits` variable is a structure of 8 bits, managed together and individually addressable using structure member notation. You can assign a byte value directly to a `bits` variable, and then address individual bits.

Listing 6.2 is an example for the MC68705J1A.

Listing 6.2 Bit-sized variable types

```
                                    bits switch_fixup(void)
                                    {
00EB 0000                              bit heat_flag;
00EB 0001                              bit cool_flag;
00EA                                   bits switches;

0300 00 01 04  BRSET  0,$01,$0307      heat_flag = PORTB.0;
0303 11 EB     BCLR   0,$EB
0305 20 02     BRA    $0309
0307 10 EB     BSET   0,$EB
0309 02 01 04  BRSET  1,$01,$0310      cool_flag = PORTB.1;
030C 13 EB     BCLR   1,$EB
030E 20 02     BRA    $0312
0310 12 EB     BSET   1,$EB

0312 B6 01     LDA    $01             switches = PORTB;
0314 B7 EA     STA    $EA
0316 0B EA 05  BRCLR  5,$EA,$031E      if(switches.5 &&
                                         heat_flag) switches.1 = 0;
0319 01 EB 02  BRCLR  0,$EB,$031E
031C 13 EA     BCLR   1,$EA
                                       return(switches);
031E 81        RTS
                                    }
```

6.6— Real Numbers

While many desktop computer applications make extensive use of real or floating point numbers (numbers with digits on both sides of the decimal place), 8-bit microcontroller applications do not. The resources needed to store and manipulate floating point numbers can place overwhelming demands on an 8-bit computer. Usually, the value gained is not worth the resources expended.

The fundamental data type for representing real numbers in C is the `float` type. The maximum value for the target computer is defined in a C header file called `values.h` as a symbolic constant called `MAXFLOAT`.

C compilers generally allocate four bytes for a `float` variable, which provides approximately six digits of precision to the right of the decimal. You can have greater precision with the `double` and `long double` data types. Compilers typically allocate eight bytes for a `double` variable and more for a `long double`. There are approximately 15 digits of precision with `double` values and perhaps more from `long double` values.

Another format, IEEE 754, specifies a 4- or 3-byte format for floating-point numbers.

You can assign an integer value to a floating point data type, but you must include a decimal and a 0 to the right of the decimal.

```
myFloatVariable = 2.0;
```

6.7— Complex Data Types

Complex data types include pointers, arrays, enumerated types, unions, and structures. Even within the restricted resources of an 8-bit microcontroller, complex data types are useful in organizing an embedded program.

6.7.1— Pointers

The implementation of pointer variables is heavily dependent upon the instruction set of the target processor. The generated code will be simpler if the processor has an indirect or indexed addressing mode.

It is important to remember that Harvard architectures have two different address spaces, and so the interpretation of pointers can change. A dereference of a RAM location will use different instructions than a dereference into ROM.

It is also important to differentiate between `near` and `far` pointers. The differences in code generation can be significant. For more information, see Section 6.9.4, Pointer Size Modifiers: `near` and `far`.

6.7.2— Arrays

When you declare an array, you must declare both an array type and the number of elements it contains. For example, the following declares an array containing eight `int` elements.

```
int myIntArray[8];
```

When you declare an array, a single, contiguous block of memory is reserved to hold it. This is why you must specify the array size or assign the contents in the declaration.

Listing 6.3 Initialized and uninitialized arrays

```
00C0 0008                       int myarray[8];
       /* uninitialized */
00C8 01 08 02 07 03 06 04 05    int my2array[] =
                                  {1,2,4,8,16,32,64,128};
       /* initialized below */
0312 01 08 02 07 03 06 04 05    const int myconsts[] =
                                  {1,8,2,7,3,6,4,5};
       /* no code generated for const array */



/* ... main() code omitted for clarity ... */

/* Initialization code. The first passage clears all variable
   memory. The second initializes my2array. Finally, the jump
   to main(). */

07FE 03 32
0332 AE C0     LDX   #$C0
0334 7F        CLR   ,X
0335 5C        INCX
0336 A3 EB     CPX   #$EB
0338 26 FA     BNE   $0334
```

```
033A 5F        CLRX
033B D6 03 48  LDA   $0348,X
033E E7 C8     STA   $C8,X
0340 5C        INCX
0341 A3 08     CPX   #$08
0343 26 F6     BNE   $033B

0345 CC 03 1A  JMP   $031A

0348 01 02 04 08 10 20 40 80
```

There are some restrictions on or disadvantages to using arrays in embedded C programming. They arise because of the available methods of indexing into an array.

The Byte Craft compiler forbids arrays of `struct` and `union`. This restriction arises because of the difficulty in addressing members of the data structures, which are themselves being addressed as array members. To overcome this limitation, you can use several global arrays of basic data types, and organize them together by context.

6.7.3— Enumerated Types

Enumerated types are finite sets of named values.

For any list of enumerated elements, the compiler supplies a range of integer values beginning with 0 by default. While in many cases this is sufficient to identify elements in the set, in embedded C you may wish to associate the enumerated set to a device-dependent progression. Enumerated elements can be set to any integer values in two ways.

1. Specify values for each enumerated element. The following example is from the COP8SAA7 WATCHDOG service register WDSVR. Bits 6 and 7 of this register select an upper limit to the service window that selects WATCHDOG service time.

Listing 6.4 Specifying integer values for enumerated elements

```
enum WDWinSel { Bit7 = 7, Bit6 = 6 };
```

Since character constants are stored as integer values, they can be specified as values in an enumerated list.

```
enum DIGITS {one='1', two= '2', three='3'};
```

will store the appropriate integer values of machine character set (usually ASCII) for each digit specified in the element list.

2. Specify a starting value for one or more of the enumerated elements. By default, the compiler assigns the value 0 to the first element in the list. You can set the list to begin with another value.

Listing 6.5 Specifying a starting value for enumerated elements

```
enum ORDINALS {first = 1, second, third, fourth, fifth};
```

When the compiler encounters an element in an enumerated list without an assigned value, it counts from the last value that was specified. For example, the following enumerated list specifies the appropriate values for its elements.

Listing 6.6 The assignment of integer values to an enumerated list

```
enum ORDINALS {first =1, second, fifth=5, sixth, seventh};
```

6.7.4— Structures

Structures support the meaningful grouping of program data. Building understandable data structures is one key to the effectiveness of a new program.

The following declaration creates a structured type for an extended time counter and describes each element within the structure. The display is defined as having the components `hours`, `minutes`, `seconds`, and an AM/PM flag. Later, a variable `timetext` is declared to be of type `struct display`.

Listing 6.7 Declaring the template of a structure

```
struct display {
        unsigned int hours;
        unsigned int minutes;
        unsigned int seconds;
```

```
        char AorP;
};

struct display timetext;
```

The Byte Craft compiler permits structures of bit fields, with individual fields taking less than 8 bits. Using bit fields allows the declaration of a structure that takes up the minimum amount of space needed: several fields could occupy one single byte.

The following example for the Motorola MC68HC705C8 defines the Timer Control Register (TCR) bits as bit fields in the structure called `TCR`, and uses the structure to configure the timer output compare.

Listing 6.8 Bit fields in structures

```
struct reg_tag {
  int ICIE : 1; /* field ICIE, 1 bit long */
  int OCIE : 1; /* field OCIE, 1 bit long */
  int notUsed : 3 = 0; /* notUsed is 3 bits and set to 0 */
  int IEDG : 1; /* field IEDG 1 bit long */
  int OLVL : 1; /* field OLVL 1 bit long */
} TCR;

/* To configure the timer: */

TCR.OLVL = 1; /* TCMP pin goes high on output compare successful */
```

The Byte Craft compiler can span a bit field across two bytes. Not all compilers support this optimization, however. In the worst case, the following structure would place the second field entirely in a separate word of memory from the first.

Listing 6.9 Compiler dependant storage of bit fields

```
struct {
        unsigned int shortElement : 1; /* 1 bit in size */
        unsigned int longElement : 7; /* 7 bits in size */
} myBitField;  /* could be 1 byte, worst case 2 */
```

The order in which the compiler stores elements in a structure bit field also varies from compiler to compiler.

Bit field elements behave exactly as an unsigned `int` of the same size. Thus, an element occupying a single bit could have an integer value of either 0 or 1, while an element occupying two bits could have any integer value ranging from 0–3. You can use each field in calculations and expressions exactly as you would an `int`.

6.7.5— Unions

C programmers developing for traditional platforms do not often use the `union` data type, but it is a very useful resource for the embedded system developer. The `union` type interprets data stored in a single block of memory based on one of several associated data types.

One common use of the `union` type in embedded systems is to create a scratch pad variable that can hold different types of data. This saves memory by reusing one 16-bit block in every function that requires a temporary variable. The following example shows a declaration to create such a variable.

Listing 6.10 Using a union to create a scratch pad

```
struct lohi_tag{
        short lowByte;
        short hiByte;
};
union tagName {
       int asInt;
       char asChar;
       short asShort:
       long asLong;
       int near * asNPtr;
       int far * asFPtr;
       struct hilo_tag asWord;
} scratchPad;
```

Another common use for `union` is to facilitate access to data as different types. For example, the Microchip PIC16C74 has a 16-bit timer/counter register called `TMR1`. `TMR1` is made up of two 8-bit registers called `TMR1H` (high byte) and `TMR1L` (low byte).

It might be desirable to access either of the 8-bit halves, without resorting to pointer manipulation. A union will facilitate this type of data access.

Listing 6.11 Using a union to access data as different types

```
struct asByte {
        int TMR1H; /* high byte */
        int TMR1L; /* low byte */
}
union TIMER1_tag {
      long TMR1_word;  /* access as 16 bit register */
        struct asByte halves;
} TMR1;

/* ... */

seed = TMR1.halves.TMR1L;
```

Since the compiler uses a single block of memory for the entire union, it allocates a block large enough for the largest element in the union. The compiler will align the first bits of each element in the lowest address in the memory block. If you assign a 16-bit value to `scratchPad` and then read it as an 8-bit value, the compiler will return the first 8 bits of the data stored.

If you arbitrarily extract one byte of a 16-bit variable, the value returned will differ depending on the **endianness** of the processor architecture. As mentioned in Section 5.2.3, Endianness, C does not contemplate endianness.

6.8— `typedef`

The `typedef` keyword defines a new variable type in terms of existing types. The compiler cares most about the size of the new type, to determine the amount of RAM or ROM to reserve.

Listing 6.12 Defining new types with `typedef`

```
typedef int new_int;
new_int result;  /* represents same range of values
                    in a different context. */


typedef struct {
     char * name;
     int start;
     int min_temp;
     int max_temp;
} time_record;

time_record targets[] {
     { "Night",   0, 20, 25},
     { "Day", 5*3600, 20, 25},
     { "Evening", 18*3600, 20, 25},
}
```

6.9— Data Type Modifiers

The C language allows you to modify the default characteristics of simple data types. Mainly, these data type modifiers alter the range of allowable values.

Type modifiers apply to data only, not to functions. You can use them with variables, parameters, and returned data from functions.

Some type modifiers can be used with any variable, while others are used with a set of specific types.

6.9.1— Value Constancy Modifiers: `const` and `volatile`

The compiler's ability to optimize a program relies on several factors. One of these is the relative constancy of the data objects in your program. By default, variables used in a program change value when the instruction to do so is given by the developer.

Sometimes, you want to create variables with unchangeable values. For example, if your code makes use of π, the constant PI, then you should place an approximation of the value in a constant variable.

```
const float PI = 3.1415926;
```

When your program is compiled, the compiler allocates ROM space for your `PI` variable and will not allow the value to be changed in your code. For example, the following assignment would produce an error at compile time (thank goodness).

```
PI = 3.0;
```

In embedded C, storage for constant data values is allocated from computer program memory space, usually ROM or other nonvolatile storage.

For the Byte Craft compiler, a declaration such as

```
const int maximumTemperature = 30;
```

declares a byte constant with an initial value of 30 decimal. The compiler will reserve far more than just one or two bytes for a constant if any special technique is required to load the value into a register. Due to architectural limitations, some platforms require constants to be the parameter of a multi-byte load statement embedded in a ROM subroutine: to access the constant value, the processor *executes* the dedicated load statement.

Volatile variables are variables whose values may change outside of the immediately executing software. For example, a variable that is "stored" at the location of a port data register will change as the port value changes.

Using the `volatile` keyword informs the compiler that it can not depend upon the value of a variable and should not perform any optimizations based on assigned values.

6.9.2— Allowable Values Modifiers: `signed` and `unsigned`

By default, integer data types can contain negative values. You can restrict integer data types to positive values only. The sign value of an integer data type is assigned with the `signed` and `unsigned` keywords.

The `signed` keyword forces the compiler to use the high bit of an integer variable as a sign bit. If the sign bit is set with the value 1, then the rest of the variable is interpreted as a negative value. By default, `short`, `int`, and `long` data types are signed. The `char` data type is unsigned by default. To create a signed `char` variable, you must use a declaration such as

```
signed char mySignedChar;
```

If you use the `signed` or `unsigned` keywords by themselves, the compiler assumes that you are declaring an integer value. Since `int` values are `signed` by default, programmers rarely use the syntax `signed mySignedInt;`.

6.9.3— Size Modifiers: `short` and `long`

The `short` and `long` modifiers instruct the compiler how much space to allocate for an `int` variable.

The `short` keyword modifies an `int` to be of the same size as a `char` variable (usually 8 bits).

```
short int myShortInt;
```

If you use the `short` keyword alone, the compiler assumes the variable is a `short int` type.

```
short myShortInt;
```

The `long` keyword modifies an `int` to be twice as long as a normal `int` variable.

```
long int myLonglnt;
```

Omitting the `int` in a long declaration likewise assumes a `long int`.

6.9.4— Pointer Size Modifiers: `near` and `far`

The `near` and `far` keywords are influenced a great deal by the target computer architecture.

The `near` keyword creates a pointer that points to objects in the bottom section of addressable memory. These pointers occupy a single byte of memory, and the memory locations to which they can point is limited to a bank of 256 locations, often from `$0000–$00FF`.

```
int near * myNIntptr;
```

The `far` keyword creates a pointer that can point to any data in memory:

```
const char * myString = "Constant String";
char far * myIndex = &myString;
```

These pointers take two bytes of memory, which allows them to hold any legal address location from `$0000–$FFFF`. `far` pointers usually point to objects in user ROM, such as user-defined functions and constants.

6.10— Storage Class Modifiers

Storage class modifiers control memory allocation for declared identifiers. C supports four storage class modifiers that can be used in variable declarations: `extern`, `static`, `register`, and `auto`. Only `extern` is used in function declarations.

The ISO standard specifies `typedef` as a fifth modifier, though it explains that this is for convenience only. `typedef` is described in Section 6.8, `typedef`.

When the compiler reads a program, it must decide how to allocate storage for each identifier. The process used to accomplish this task is called **linkage**. C supports three classes of linkage: external, internal, and none. C uses identifier linkage to sort out multiple references to the same identifier.

6.10.1— External Linkage

References to an identifier with **external linkage** throughout a program all call the same object in memory. There must be a single definition for an identifier with external linkage or the compiler will give an error for duplicate symbol definition. By default, every function in a program has external linkage. Also by default, any variable with global scope has external linkage.

6.10.2— Internal Linkage

In each compilation unit, all references to an identifier with **internal linkage** refer to the same object in memory. This means that you can only provide a single definition for each identifier with internal linkage in each compilation unit of your program. A compilation unit can be more than one file because of `#include` directives.

No objects in C have internal linkage by default. Any identifier with global scope (defined outside any statement block) *and* with the `static` storage class modifier, has internal linkage. Also, any variable identifier with local scope (defined within a statement block) *and* with the `static` storage class modifier, has internal linkage.

Although you can create local variables with internal linkage, scoping rules restrict local variable visibility to their enclosing statement block. This means that you can create local variables whose values persist beyond the immediate life of the statement blocks in which they appear. Normally, the computer shares local variable space between several different statement

blocks. If a local variable is declared as `static`, space is allocated for the variable once only: the first time the variable is encountered.

> **Note**
> Unlike other internal linkage objects, static local variables need not be unique within the compilation unit. They must be unique within the statement block that contains their scope.

Objects with internal linkage typically occur less frequently than objects with external or no linkage.

6.10.3— No Linkage

References to an identifier with no linkage in a statement block refer to the same object in memory. If you define a variable within a statement block, you must provide only one such definition.

Any variable declared within a statement block has no linkage by default, unless the `static` or `extern` keywords are included in the declaration.

6.10.4— The `extern` Modifier

Suppose the library function

```
int Calculate_Sum()
```

is declared in a library source file. An identifier with external linkage like this can be used at any point within the same compilation unit, as long as it was previously declared.

If you want to use this function in any other compilation unit, you must tell the compiler that the definition of the function is or will be available. The concept is identical to prototyping a function, except that the actual definition will not appear in the same compilation unit. The function definition is external to the compilation unit.

To declare an external function, use the `extern` keyword.

```
extern int Calculate_Sum();
```

When the compiler encounters an external function declaration, it interprets it as a prototype for the function name, type, and parameters. The

`extern` keyword claims that the function definition is in another compilation unit. The compiler defers resolving this reference to the linker.

If you build a library of functions to use in many programs, create a header file that includes `extern` function declarations. Include this header in your compilation unit to make library functions available to your code.

Like functions, global variables have external linkage. A global variable is a good way to present general configuration settings for a library. This avoids an extra function call.

To create a global variable that can be read or set outside its compilation unit, you must declare it normally within its source file and declare it as `extern` within a header file.

```
extern int myGlobalInt;
```

The compiler interprets an external declaration as a notice that the actual RAM or ROM allocation happens in another compilation unit.

6.10.5— The `static` Modifier

By default, all functions and variables declared in global space have external linkage and are visible to the entire program. Sometimes you require global variables or functions that have internal linkage: they should be visible within a single compilation unit, but not outside. Use the `static` keyword to restrict the scope of variables.

Listing 6.13 Using the static data modifier to restrict the scope of variables

```
static int myGlobalInt;
static int staticFunc(void);
```

These declarations create global identifiers that are not accessible by any other compilation unit.

The `static` keyword works almost the opposite for local variables. It creates a permanent variable local to the block in which it was declared. For example, consider the unusual task of tracking the number of times a recursive function calls itself (the function's depth). You can accomplish this using a static variable.

Listing 6.14 Using static variables to track function depth

```
void myRecurseFunc(void) {
        static int depthCount=1;
        depthCount += 1;
        if ( (depthCount < 10) && (!DONE) ) {
               myRecurseFunc();
        }
}
```

`myRecurseFunc` contains an `if` statement that stops it from recursing too deeply. The `static` variable `depthCount` is used to keep track of the current depth.

Normally, when a function is called, the computer reinitializes its automatic local variables (or at least leaves them in a questionable state). Memory for `static` variables, however, is only initialized once. The `static` variable `depthCount` retains its value between function calls.

Because `depthCount` is defined inside the `myRecurseFunc()` statement block, it is not visible to any code *outside* the function.

6.10.6— The `register` Modifier

When you declare a variable with the `register` modifier, you inform the compiler to optimize access to the variable for speed. Traditionally, C programmers use this modifier when declaring loop counter variables.

Listing 6.15 Using the register data type modifier

```
{
  register int myCounter = 1;
      while (myCounter<10) {
              /* ... */
              myCounter += 1;
      } /* end while */
} /* enclosing block enforces reallocation of myCounter */
```

Unlike other storage class modifiers, `register` is simply a recommendation to the compiler. The compiler may use normal memory for the variable if it is out of registers to allocate.

Because of the scarcity of registers on 8-bit machines and the desire for size optimization rather than speed, the `register` keyword is not very useful for embedded system programmers.

Notice that the technique used in the example does two things: it places the `register` declaration and the `while` loop close together and inside a statement block. This minimizes the cost of potentially dedicating a register to a specific variable. It also forces the compiler to reallocate storage for `myCounter` as soon as the loop is finished: if the compiler uses a register to store `myCounter`, it will not tie up the register longer than necessary.

6.10.7— The `auto` Modifier

The `auto` keyword denotes a temporary variable (as opposed to `static`). You can only use `auto` with local variables, because C does not support functions within a block scope. Since all variables declared inside a statement block have no linkage by default, the only reason to use the `auto` keyword is for clarity.

Listing 6.16 Using the `auto` data modifier

```
int someFunc(NODEPTR myNodePtr) {
        extern NODEPTR TheStructureRoot;
        /* global pointer to data structure root */
        auto NODEPTR tempNodePtr;
        /* temporary pointer for structure manipulation */
        /* ... */
}
```

In this example, we declare `tempNodePtr` as an `auto` variable to make it clear that, unlike the global `TheStructRoot` pointer, `tempNodePtr` is only a temporary variable.

Chapter 7—
C Statements, Structures, and Operations

Part of the benefit of using C for programming is the availability of mathematical expression. Beyond simple constant calculations, assembly forces you into a rigorous, procedural structure. C provides assignment statements, logical and arithmetic expressions, and control structures that allow you to express yourself using common math notation and helpful metaphors.

7.1—
Combining Statements in a Block

You create **statement blocks** for your functions, and at other times for the bodies of control statements. For instance, the general format for the `while` statement looks like the following.

```
while (condition) statement;
```

Since you can substitute a statement block anywhere a single statement can occur, the `while` statement most commonly appears as follows.

```
while (condition){
    statements
}
```

7.2— Functions

When the compiler reaches the **function definition**, it generates machine instructions to implement the functionality, and reserves enough program memory to hold the statements in the function. The address of the function is available through the symbol table.

A function definition includes a statement block that contains all function statements. Even if a function has only a single executable statement, it must be enclosed in a statement block.

Embedded C supports **function prototypes.** Function prototype declarations ensure that the compiler knows about a function and its parameter types, even if its definition has yet to appear in the compiler's input. Prototypes assist in checking forward calls. The function name is recorded as an identifier, and is therefore known when invoked in code prior to its definition.

Header files of function prototypes provide the foundation for using libraries.

The syntax for a **function call** in C is the function name and a list of actual parameters surrounded by parentheses.

Function calling is one area in which embedded C differs substantially from traditional C. The way that parameters are passed differs significantly, as well as the permitted number of parameters.

Functions that produce extensive **side effects** are harder to maintain and debug, especially for members of a development team. To safely use abstract functions, you need to know only the data that goes in and comes out — the function interface. When a function produces side effects, you need to know about the interface *and* behaviour to use it safely.

Some C programmers insist that functions that just produce side effects should return a value to indicate success, failure, or error. Since ROM space is at a premium, the code needed to evaluate the return status is a luxury.

7.2.1— Function Parameters

C for embedded processors places some unique restrictions on function calls. Some compilers restrict the number of parameters that can be passed to a function. Two byte-sized parameters (or one 16-bit parameter) can be passed within the common processor registers (accumulator and index register).

To pass by reference, pass a pointer as usual. See information on pointers in Section 6.7.1, `Pointers`, for extra information about the relative cost of using pointers.

A function with no parameters can be declared with an empty parameter list.

```
int myFunc()
```

However, it is good practice to specify that the function has no parameters with the `void` parameter type.

```
int myFunc(void)
```

In embedded programs, `main()` does not accept any parameters.

7.3— Control Structures

While the flow of some embedded C programs will appear strange at first (the prominence of `while(1)`, for instance), they are not fundamentally different than those in C for personal computing.

7.3.1— The `main()` Function

It may seem incongruous that an embedded program, which has no operating system to invoke it, has a traditional `main()` function and an explicit return value specification. What invokes `main()`? Where will the function return?

Embedded C retains the `main()` function for compatibility with standard C. The return type of `main()` should always explicitly be declared as `void`; omitting it, as mentioned in Section 6.2, `Function Data Types`, causes it to be understood as an `int` return.

From there, the `main()` function can execute code from other functions and receive return values. Remember to make your called functions available to `main()` by prototyping them, if necessary.

7.3.2— Initialization Functions

Embedded C also permits specialized initialization routines. `__STARTUP()` is one such function understood by the Byte Craft compiler. If it is present, its statements are executed before control is passed to `main()`.

You can better organize initialization tasks with a separate initialization function. Device-dependent hardware initialization, which must be rewritten for each target device, can live in the `__STARTUP` routine or equivalent.

7.3.3— Control Statements

Embedded developers often use program control statements that are avoided by other programmers. For example, the `goto` statement is used in C in the same contexts as an explicit jump or unconditional branch instruction would be used in assembly.

7.4— Decision Structures

C provides three structures the programmer can use to support different types of decisions. Decision structures test an expression to determine which statement or statement block to execute.

`if..else` is available, as expected. The C conditional operator is also available.

```
if(expression) statement else statement
result = expr ? result_if_true : result_if_false
```

The `switch..case` structure chooses between several different possible paths of code to execute. The `switch..case` structure is compiled to a structure resembling a string of `if..elses`.

Listing 7.1 `switch` and `case`

```
00EB                        int choice;

                            switch(choice) {
                              case 1: return 5;
0304 A1 01    CMP   #$01
0306 26 03    BNE   $030B
0308 A6 05    LDA   #$05
030A 81       RTS
```

```
030B A1 02    CMP   #$02       case 2: return 11;
030D 26 03    BNE   $0312
030F A6 0B    LDA   #$0B
0311 81       RTS
0312 A1 03    CMP   #$03       case 3:  return 37;
0314 26 03    BNE   $0319
0316 A6 25    LDA   #$25
0318 81       RTS
                                default: return 9;
0319 A6 09    LDA   #$09
031B 81       RTS
```

The Byte Craft compiler can extend the `case` label to deal with common programming problems. These two examples would require a great deal more generated code if the compiler accepted only single integer values for each `case` label.

Listing 7.2 Byte Craft `case` extensions

```
case '0'..'9':     /* accepts a range of values from '0' to '9' */
case 0x02,0x04:    /* accepts alternative values */
```

The benefit of such structures is in avoiding recomparing the `switch` argument for each integer value within a range of cases. The compiler can generate simple comparisons to deal with ranges or lists of alternate values.

Listing 7.3 A `case` comprising a range of values

```
                                case '0'..'9':
                                  {
0473 A1 30    CMP #$30
0475 25 24    BCS $049B         /* branch if less */
0477 A1 3A    CMP #$3A
0479 24 20    BCC $049B         /* branch if greater */
047B AE DA    LDX #$DA
047D CD 05 4B JSR $054B           scanf(&temperature,ch);
```

7.5— Looping Structures

C control structures allow you to make a decision on the path of code execution. C also provides looping structures for control over program flow. Loop control structures allow you to repeat a set of statements.

`while` plays an interesting role in embedded C. You will often use `while` to intentionally create infinite loops. An embedded controller typically executes a single program "infinitely", so this structure is appropriate.

The alternative, using a `goto`, requires you to use a label; the compiler will implement the `while (1)` decision with an unconditional jump or branch instruction anyway.

Listing 7.4 A skeleton infinite loop

```
                                  void main(void)
                                  {

                                    while(1)
                                      {
0300 B6 01    LDA   $01               PORTB = PORTB << 1;
0302 48       LSLA
0303 B7 01    STA   $01
0305 20 F9    BRA   $0300             }
                                  }
```

7.5.4— Control Expression

The key component of any loop structure is the control expression. At some point in each iteration, the control expression is tested. If the control expression evaluates to 0, program execution passes to the first statement following the loop structure. If the expression evaluates to 1, execution continues within the loop structure statement block.

7.5.5— `break` and `continue`

C provides two ways to escape a looping structure: the `break` and `continue` statements. When either of these statements is encountered inside a loop, any remaining statements inside the loop are ignored.

Use a `break` statement to completely break out of a structure. When a `break` is encountered inside a looping structure, the loop terminates immediately and execution passes to the statement following the loop.

You may wish to jump to the next iteration of a loop without breaking out of the loop entirely. A `continue` statement will allow you to do this. When a `continue` statement is encountered inside a looping structure, execution passes immediately to the *end* of the loop statement block.

If `continue` is used with a `while` or `for` loop, execution jumps from the end of the statement block to the control expression at the top of the loop. If used with a `do` loop, execution passes from the end of the statement block to the control expression at the bottom of the loop. In all cases, the effect is the same — a `continue` statement does not circumvent the loop control expression, but it does skip any statements remaining in the loop iteration.

The most common place for a `break` statement is inside a `switch..case` structure. Since `switch..case` is not a looping structure, a `continue` statement within it refers to the enclosing loop structure (if any).

Listing 7.5 `break` and `continue` in loop and switch statements

```
00EB                                  char ch;

                                      while(1)
                                        {
030D AD F1    BSR   $0300               ch = getch();
030F B7 EB    STA   $EB
                                        switch(ch)
                                        {
                                        case '0'..'9':
                                          {
0311 A1 30    CMP   #$30                   putch(ch);
0313 25 08    BCS   $031D
0315 A1 3A    CMP   #$3A
0317 24 04    BCC   $031D
0319 AD E8    BSR   $0303
031B 20 10    BRA   $032D                  break; /* after switch */
                                           }
                                         case 'A'.. C':
                                            {
```

```
031D A1 41   CMP    #$41                    continue; /* A-C ignored */
031F 25 04   BCS    $0325
0321 A1 44   CMP    #$44
0323 25 E8   BCS    $030D                   /* top, before getch() */
                                          }
                                        case 'D':
                                          {
0325 A1 44   CMP    #$44                    LCD_send_control(LCDCLR);
0327 26 04   BNE    $032D
0329 A6 05   LDA    #$05
032B AD DB   BSR    $0308
          /* falls through! */            break;
                                          }
                                        }

/* other statements in the while(1) loop appear here */

03FF 20 DE   BRA   $030D                }
```

7.6— Operators and Expressions

Using C for embedded programming relieves the tedium of coding large arithmetic operations by hand. Where a 32-bit integer divide operation may be encompassed by one instruction on a general-purpose microprocessor, an 8-bit controller will need a series of loads and stores, in addition to the simplified math operations, to perform the equivalent work.

With embedded systems, there is an increased emphasis on bitwise operations. Both for peripheral operation and for memory efficiency, the compiler will try wherever possible to use bit-manipulation instructions to implement bitwise operators.

7.6.1— Standard Math Operators

Multiply instructions are sometimes available in hardware. If the instruction is an enhancement to an architecture, the compiler may need configuration to generate code that uses it. The Byte Craft compiler can take advantage of an optional multiply instruction with an appropriate `#pragma has` instruction in the device header. See Section 5.2.1 `#pragma has` for more information.

?

If no instruction is available, the compiler will provide multiply, as well as divide, and modulus as functions. The Byte Craft compilers do this automatically if the operations are used.

7.6.2— Bit Logical Operators

C supports one unary and three binary bitwise logical operators. Each of these operators act only upon values stored in the `char`, `short int`, `int`, and `long int` data types.

> **Note**
> Binary logical operators perform data promotion on operands to ensure both are of equivalent size. If you specify one `short` operand and one `long` operand, the compiler will *widen* the `short` to occupy the `long` 16 bits. The expression will return its value as a 16-bit integer.

The bitwise AND operator, &, produces a bit-level logical AND for each pair of bits in its operands. For example, if both operands have bit 0 set, then the result of the bitwise AND expression has bit 0 set.

Listing 7.6 Bitwise `AND` operation using &

```
int x=5, y=7, z; /* 5 is binary 101 and 7 is binary 111 */
z = x & y;  /* z gets the value 5 (binary 101) */
```

The `AND` operation is easier to imagine if your compiler has an extension that permits data values in binary.

Listing 7.7 Using the `AND` bitwise operator with binary values

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x & y;/* z gets the value 00000101, or 5 */
```

The bitwise OR operator, |, performs a bit-level logical OR for each pair of bits in its operands. If either operand has a bit in a specific position set, then the result of the bitwise OR expression has that bit set.

Listing 7.8 Using the bitwise OR operator |

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x | y;/* z gets the value 00000111, or 7 */
```

The bitwise XOR operator, ^, produces a bit-level logical exclusive OR for each pair of bits in the operand. XOR sets a bit when one of the operands has a bit set in that position, but not if both operands have the bit set. This produces a result with bits set that the operands do not share.

Listing 7.9 The bitwise XOR operator

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x ^ y;/* z gets the value 00000010, or 2 */
```

The bitwise NOT operator, ~, produces the complement of a binary value. Each bit that was set in the operand is cleared and each cleared bit is set.

Listing 7.10 The bitwise NOT operator

```
int x=0b00000101,
    z;
z = ~x;/* z gets the value 11111010, or 250 */
```

If you apply bitwise operators to individual bits, the compiler will use bit manipulation instructions, if they are available. They avoid unintended side effects from reads or writes to other bits.

Listing 7.11 Bitwise operations on individual bits

```
                                  void alternate( void )
                                  {

0300 0D 00 03 BRCLR 6,$00,$0306     PORTB.2 = ~PORTA.6;
0303 15 01    BCLR  2,$01
0305 81       RTS
0306 14 01    BSET  2,$01
0308 81       RTS                 }
```

7.6.3— Bit Shift Operators

Both operands of a bit shift operator must be integer values.

The right shift operator shifts the data right by the specified number of positions. Bits shifted out the right side disappear. With unsigned integer values, 0s are shifted in at the high end, as necessary. For signed types, the values shifted in is implementation-dependant. The binary number is shifted right by *number* bits.

```
x >> number;
```

Right shifting a binary number by *n* places is the same as an integer division by 2^n.

The left shift operator shifts the data right by the specified number of positions. Bits shifted out the left side disappear and new bits coming in are 0s. The binary number is shifted left by *number* bits.

```
x << number;
```

Left shifting a binary number is equivalent to multiplying it by 2^n.

Listing 7.12 Shifting bits left and right

```
porta = 0b10000000;
while (porta.7 != 1){
      porta >> 1;
```

```
}
while (porta.0 != 1){
      porta << 1;
}
```

Shifting by a variable number of bits can create a substantial loop structure in code. This presents an extra cost in ROM space that you must keep in mind.

Listing 7.13 Shifting by a variable number

```
00EB                                int setting;

/* set LED bit based on integer level from keypad */

0303 AD FB     BSR    $0300         setting = getch() - '0';
0305 A0 30     SUB    #$30
0307 B7 EB     STA    $EB

0309 A6 01     LDA    #$01          PORTB = 1 << setting;
030B BE EB     LDX    $EB
030D 27 04     BEQ    $0313
030F 48        LSLA
0310 5A        DECX
0311 26 FC     BNE    $030F
0313 B7 01     STA    $01
```

Chapter 8—
Libraries

Libraries contain functions that serve a common purpose and a wide range of development projects. Embedded and desktop systems share some library needs (e.g., enhanced mathematical functionality or data type conversion). Libraries are the typical generic structure for cataloguing and transporting this specialized knowledge.

Embedded systems can rely on libraries even more: a library can provide **device drivers** for a common LCD controller or a timer peripheral. Programmers can be overwhelmed by taking responsibility for everything within an embedded system. A programmer can relax and focus on the core of the project if they have libraries to help them with direct manipulation of hardware peripheral devices.

Since C is intended to be highly portable, libraries are a way to organize platform dependency. Main line C code written for one specific 8-bit microcontroller can therefore be compiled for and run on a different microcontroller with very minor changes to the code. Without the portability offered by libraries, your investment in a particular architecture grows, and it becomes less attractive to seek out a less-expensive processor option.

The Byte Craft Code Development System products ship with a range of useful portable libraries (and traditional API-style documentation). They provide routines for the most common features of 8-bit embedded systems.

• Standard I/O

With appropriate configuration, you can deal with a keypad and LCD display as standard input and output.

• SPI (Serial Peripheral Interface)

• MICROWIRE bus

• SCI (Serial Communications Interface)

• UART (Universal Asynchronous Receiver Transmitter)

A UART is a prime candidate for replacement by "bit banging" software, which could be encapsulated within a library.

• Analog to Digital conversion and Digital to Analog Conversion

• I/O ports

While manipulating I/O ports is usually a matter of a few assignment statements, there is some benefit in abstracting the port from the particular implementation.

• LCD displays

These routines can support the standard I/O model, and provide convenience routines for clearing the display and moving the cursor.

• PWM (Pulse Width Modulation)

• Timers

8.1—
Creating Libraries

This section discusses how to create a library from scratch.

For the thermostat, we need to display the current time and preset cycle start times, as a string. A time string is seven bytes long.

```
"12:00a" /* with a trailing null  */
"06:35p" /* leading 0 to simplify things */
"23:00h" /* for regions that use 24 hour time */
```

In the thermostat, we are really tracking four times: the current time and three cycle start times. There are several alternative ways to store these values, each with tradeoffs. Directly manipulating the string representations is unworkable: it requires consuming a full quarter of working RAM, and there would be lots of code to perform very odd carries and compares.

Unsigned long variables as minute counters (0–1439) proved expensive in terms of ROM, but used only 8 bytes of RAM (and scratchpad). Structures of time counter components (i.e., hours, minutes, and am/pm) served better, but an array of them was not possible.

Two arrays of integers, one for hours and one for minutes, seemed best. Array element 0 is a good choice for the current time, and 1–3 for the daily cycle start times.

For text representation of the time, we need to translate from a time counter value (two integers) into a timestamp string. Different projects will use this type of functionality, so we will package it as a library. We concluded that both 24-hour and 12-hour systems need to be supported, and the switch between 12-hour and 24-hour should be a run-time configuration.

The library will expose two functions

```
void MinutesToTime( int hours, int minutes );
void TimeToMinutes( int near *hours, int near *minutes );
```

and two variables

```
bit use_metric;  /* determines format for conversion */
char buffer[7];  /* buffer to perform conversion */
```

To create this library, perform the following steps.

1. Create a C source file named `timestmp.c`.

2. Write in the following lines.

Listing 8.1 Source file skeleton

```
#ifndef __TIMESTMP_C
#define __TIMESTMP_C

#pragma library;

#include <timestmp.h>

/* Declared above:
   bit use_metric = 0:
   char buffer[7];
*/
```

```
void MinutesToTime( int hours, int minutes )
{
     ;
}
void TimeToMinutes( int near *hours, int near *minutes )
{
     ;
}


#pragma endlibrary;

#endif /* __TIMESTMP_C */
```

3. Create a C header file named `timestmp.h.`

4. Write in the necessary declarations and prototypes.

Listing 8.2 Header file skeleton

```
#ifndef __TIMESTMP_H
#define __TIMESTMP_H

bit use_metric;
char buffer[7];

void MinutesToTime( int hours, int minutes );
void TimeToMinutes( int near *hours, int near *minutes );

#endif /* __TIMESTMP_H */
```

5. Compile the C file.

```
c6805.exe timestmp.c +O O=timestmp.lib
```

This is the skeleton of a library. When the library is completed, place the `.lib` file with the other libraries, and the `.h` file with the other include files.

8.2— Writing the Library

The library software is much like other embedded programming. We have, in previous sections, outlined what techniques are safe, what techniques are expensive, and what techniques are impossible in the embedded environment.

`MinutesToTime()` accepts an hour integer and a minute integer. It inspects the `use_metric` flag, and renders the time in `buffer[]`.

Listing 8.3 Converting hours and minutes to a timestamp

```
void MinutesToTime( int hours, int minutes )
{
    char i;

    /* Set up string */
    buffer[5] = 'h'; buffer[6] = 0; buffer[2] = ':';

    /* Deal with 12-hour time */
    if(!use_metric) {
      buffer[5] = 'a';
      if(hours > 11)
    {
      hours = hours - 12;
      buffer[5] = 'p';
    }
      if(hours == 0)
    {
      hours = 12;
    }
    }

    /* Fill in hours */
    buffer[0] = '0';
    for(i = '2'; hours >= 10; hours -= 10, i--);
    buffer[0] = i;
    buffer[1] = hours + '0';
```

```
    /* Fill in minutes */
    buffer[3] = '0';
    for(i = '5'; minutes >= 10; minutes -= 10, i--);
    buffer[3] = i;
    buffer[4] = minutes + '0';
}
```

Alternatively, you could unroll the bottom `for` loops to avoid the loop management code.

`TimeToMinutes()`, which isn't used in the thermostat project, is the reverse function. We include it because it is simple and useful. In the thermostat project, time adjustments are made with hour and minute increment buttons, much like an alarm clock. If ROM permitted, the configuration could be rewritten to allow the user to enter the time using digits: the extra code for checking the digits entered against valid times was substantial.

`TimeToMinutes()` accepts pointers to the hours and minutes integers that should receive the translated values. Note they are `near` pointers, which should prove to be 8-bit values.

Listing 8.4 Converting a timestamp buffer to hours and minutes

```
void TimeToMinutes( int near *hours, int near *minutes)
{
    if(buffer[0] <= '0') buffer[0] = '0';
    if(buffer[0] >= '2') buffer[0] = '2';

    *hours = (buffer[0] - '0') * 10;
    *hours += (buffer[1] - '0');

    if(buffer[3] <= '0') buffer[0] = '0';
    if(buffer[3] >= '5') buffer[0] = '5';

    *minutes = ((buffer[3] - '0') * 10);
    *minutes += ((buffer[4] - '0'));

    if(buffer[5] = 'p') *hours += 12;
}
```

8.3—
Libraries and Linking

With the Byte Craft compilers, there are two scenarios for library use: traditional linking with *BClink* and Absolute Code Mode.

As previously presented, the `timestmp` library source files are written for Absolute Code Mode. To use them, write your main module as follows.

Listing 8.5 Sample source using Absolute Code Mode

```
#include <705j1a.h> /* insert your device here */
#include <timestmp.h>


void main(void) {
/* ... */
}

#include <timestmp.c>
```

To make `timestmp` suitable for linking, you need to add some conditional defines to the library header. Ideally, the header file should allow both Absolute Code Mode and traditional linking. Use the `MAKEOBJECT` symbol to choose between the two as shown in Listing 8.6.

Change `timestmp.h` to the following.

Listing 8.6 Header file for both linking and Absolute Code Mode

```
#ifndef __TIMESTMP_H
#define __TIMESTMP_H

ifdef MAKEOBJECT

#include <dev_def.h> /* replace dev with your CDS name */
extern bit use metric;
extern char buffer[7];

extern void MinutesToTime( int hours, int minutes );
extern void TimeToMinutes( int near *hours, int near *minutes );

#else /* MAKEOBJECT */
```

```
bit use_metric;
char buffer[7];

void MinutesToTime( int hours, int minutes );
void TimeToMinutes( int near *hours, int near *minutes );

#endif /* MAKEOBJECT */

#endif /* __TIMESTMP_H */
```

No changes are needed for `timestmp.c` if it includes the header file itself.

You can define `MAKEOBJECT` on the command line when you create the library object file. Invoke

```
cds.exe -dMAKEOBJECT timestmp.c +O O=timestmp.lib
```

where *cds* is your compiler executable name. Copy the `.lib` file to the libraries directory and the `.h` file to the headers directory.

Defining the `MAKEOBJECT` symbol will cause the functions and variables to be `extern`, and will include a definitions file. The definitions file is a device header file with definitions for all the important device symbols (e.g., ports, timer registers, and so on). The most common values are present in it, but these are not important: the compiler uses the definitions file to compile the library to object without depending upon a particular device header file. During linking, the actual device values will be matched with the references in the object file.

Some Byte Craft compilers define the symbol `MAKEOBJECT` automatically when compiling to an object file (`+o` is present on the command line).

One other customization is helpful: `buffer[]` is a 7-byte string in RAM that you may wish to declare in other ways (for instance, as `SPECIAL` memory). You can conditionalize its declaration with an `#ifndef` if you are using Absolute Code Mode.

Chapter 9—
Optimizing and Testing Embedded C Programs

As in any other programming endeavour, getting the code to compile ensures only linguistic correctness. Without understanding the capabilities of the compiler, we have no real certainty about how to read the generated code.

Without understanding the compiler's limitations, we have no way of adding in human intuition. Compilers are best at relieving drudgery: they are no match for inspired programming.

Testing embedded software differs significantly from testing desktop software. One new central concern arises: embedded software often plays a much more visceral role. Where a protection fault on a desktop machine may cost the user hours of work, a software fault in an embedded system may threaten:

• the user's safety or physical comfort,

• a lifeline of communication, or

• the physical integrity of the hosting equipment.

The issue of life-supporting devices is outside the scope of this book. Devices meant for human implant, or for monitoring or regulating health-related factors, are life-supporting devices. It is debatable whether compiled code should be used in these devices. The motivation for compiled code is relief from having to write assembly code from scratch. The risks of life-supporting activities cannot permit such luxury.

Decisions about development testing software are first made when evaluating processor options. For more information about tools, see Section 3.7, Development Tools for a Microcontroller.

9.1— Optimization

Anyone interested in the art and science of compilers soon learns that **optimization** is the perpetual goal of the compiler writer. Any interesting fact about the code that the compiler can recognize becomes a candidate for optimization.

While some might feel that laborious hand-coding of assembly is the only way to really massage the code, a compiler that is detached and objective can find otherwise hidden patterns suitable for reduction.

The need for optimization is never greater than in embedded environments. For the 8-bit microcontroller, successful optimization primarily reduces the amount of ROM and RAM used. This is the acid test of code generation. Increasing execution speed comes a distant second.

There is a host of traditional strategies for optimizing generated code. You can trust that the compiler watches for these factors.

Algebraically Equivalent Variables If a reference to a variable causes it to be loaded into a register, and a reference to another variable that is known to have the same value immediately follows, the compiler can omit the extra load operation.

Register Data Flow The compiler can recognize if a variable will be loaded into a register twice, and remove the redundancy.

Code That Is Redundant or Dead Code governed by expressions that will never prove true can be ignored at compile time. Code following a `break` or `continue` statement that will never be executed, due to constants within the control structure, can be discarded.

Adjacent Instruction Reductions A pattern of simple instructions can be reduced into a more complex operation, such as an instruction with an auto-increment side-effect.

Constant Folding This evaluates constant values in the source and combines them if they are the same.

Lofting Instructions within a loop that do not directly pertain to it can be lofted to an enclosing syntax level.

Arithmetic Operations Involving Low Value Constants Operands of zero, one, and two can be changed into instructions like increment or decrement to reduce code size and improve execution time. No code is generated for adding 0, subtracting 0, or multiplying or dividing by 1.

Edge Effects Code that causes values to roll over within their variables can be a candidate for special treatment.

Long Operations In controllers that have only 8-bit registers, long operations cost far more than twice the instructions (some controllers can pair registers into a 16-bit variable and use it for longs). Any knowledge about the range of possible values can determine whether to ignore either the top or bottom bytes of a 16-bit variable.

Array Calculations Fixed references to an array element are dereferenced at compile time. This avoids overwriting an index register.

9.1.1— Instruction Set-Dependent Optimizations

Some optimizations are possible because of features of the instruction set.

• Adding 1 becomes an increment, and subtracting 1 becomes a decrement.

• ++ increments a memory location, and `--` decrements a memory location. If the variable is long, the carry must be preserved with subsequent instructions.

• Bit operations can be conducted using bit set and bit clear instructions instead of using a multibyte sequence that does a load, bitwise `AND` or `OR`, and store.

9.2— Hand Optimization

If a compiler is charged with taking a high-level program and generating optimized machine language, why should hand optimization be a concern? For all its capability, a compiler cannot see "the big picture". Sometimes it follows your high-level directions too well.

These are some strategies for conserving ROM and RAM.

Examining Register Use In small routines, a register that starts out holding a function parameter may be otherwise unused, especially if the routine manipulates memory directly (i.e., bit manipulation with specialized instructions). Our normal reflex is to declare function parameters as `int`, which will most likely cause local RAM to be reserved for the value. Declaring the function parameter as a register type (`registera` or equivalent on Byte Craft compilers) saves the byte.

Rolling and Unrolling `for` Loops It may seem unintuitive to unroll an easily-understood short loop, but the savings in ROM space may make it profitable. The opportunity to look for is expensive code generated for the condition and action parts of the loop.

Using Ports as Variables Do not underestimate the desperation with which embedded programmers pursue savings in RAM usage. If an output port can be read safely to determine the current state of the output pins, and the port needs a looping operation, there is no reason not to use the port itself as an index variable. Consider the following.

Listing 9.1 Using a port as a variable

```
#pragma portrw PORTA @ 0x00;

void walk_through_A(void)
{
  for(PORTA = 0x01; PORTA != 0; ASL(PORTA))
    delay_100us(10);
}
```

If, in this example, a separate `char` had been used to index the loop and assign to the port, there is no reason to think that the compiler could omit the otherwise unused variable. The compiler considers ports volatile, but we

can determine from the design whether the port in this case will act in a volatile manner.

9.2.1— Manual Variable Tweaking

In a traditional C environment, compilers can allocate variables without too much hand-wringing. For instance, it is common to allocate a new location for each counter variable name within a scope.

Listing 9.2 Local counter variables

```
void up_and_down(void)
{
        int up, down; /* probably separate locations */

        for(up = 0; up < 128; up++)
                porta = up;
        /* ... */
        for(down = 127; down > 0; down--)
                 porta = down;
}
```

To minimize RAM usage, embedded systems developers will often create global loop counter variables. Any function can then use this allocated block of data memory when a counter or temporary variable is needed. The programmer oversees conflicts between enclosing loops.

An alternative solution leaves the variables as strictly local: some C compilers support an extension which fixes the location of a symbol in memory. You can use this feature to manage how variables are placed in data memory space. Here is suitable notation for the Byte Craft compiler.

Listing 9.3 Local counter variables overlay on another

```
void up_and_down(void)
{
        int up;
        int down @ up; /* overlay */

        for(up = 0; up < 128; up++)
```

```
            porta = up;
        /*...*/
       for(down = 127; down > 0; down--)
            porta = down;
}
```

Because the declaration is so specific, the compiler will obey it as is. This is a useful technique for reusing allocated variable space without resorting to macros or other techniques. If memory opens up, only the unobtrusive `@ location` extension needs to be removed.

9.3— Debugging Embedded C

After learning how to interpret the results of the compiler's code generation, you can begin debugging.

There are some pitfalls in debugging C on an embedded system.

9.3.1— Register Type Modifier

Those compilers that implement the `register` keyword may not actually grant exclusive access to a register. 8-bit MCUs do not have many registers to spare. Instead, the compiler may allocate from the fastest available memory.

Other keywords, such as Byte Craft's `registera` and equivalents will associate an identifier with the appropriate register, but the resulting variable should be considered volatile. You have immediate access to all the assembly code used in your system; with it, you can determine by inspection whether the compiled code is meddling with register contents.

9.3.2— Local Memory

If your compiler supports variables with local scope, you should determine the manner in which the compiler allocates memory for variables in function calls.

There are three strategies for local memory allocation:

Within a Stack Frame This requires explicit stack-relative addressing, which is very much a luxury. It isn't always a preferred code option, and the compiler may not use it even if available.

From the Global Heap Variables are simply allocated from RAM as needed. Globals and locals intermingle.

"Dedicated" Local Memory This is used and reused from within multiple function calls.

9.3.3— Pointers

Because Harvard architecture MCUs have two address spaces that are chosen by context, pointers must target either program (ROM) space or data (RAM) space. The resulting code sequences can be confusing.

In some architectures, far pointer variables can only be accomplished by self-modifying code. For more information, see Section 9.6, Debugging by Inspection.

9.4— Mixed C and Assembly

Embedded systems code lives in a much more spartan environment than traditional application software. Resorting directly to assembly code is undesirable, unless you have to observe fixed timing, or you want to use pre-existing assembly code in your current project.

9.4.1— Calling Conventions

Embedded C cross-compilers generate less-standardized code for calling functions. When debugging your program, you should know the answers to the following questions.

• Does your compiler set up page bits, or perform bank switching, prior to calling a subroutine?

• Does the compiler or processor handle saving and restoring state during an interrupt?

• How are function arguments passed? How are results returned? It's almost guaranteed that an 8-bit result will be left the accumulator.

9.4.2— Access to C Variables from Assembly

Does your assembly code properly address C identifiers? While the compiler may allow you to use a C identifier as an argument in an assembly mnemonic, it may not check the size of the value against the prescribed size of

the instruction. As a result, the program may load one byte of a multiple byte value, without regard for its significance.

9.5— Exercising Hardware

If you have access to a prototype of the target hardware, a small program to test the hardware will confirm your beliefs about its configuration and performance.

If your main project does not behave as predicted in an emulator or development system, the same technique will determine whether a problem lies in hardware or software.

9.6— Debugging by Inspection

The compiler can help you inspect code by generating different reports. The Byte Craft compiler assembles all reports in the listing file that centres around the generated code and the source code from which it came. These reports can assist in the chores of hand optimization, as described in Section 9.2, Hand Optimization.

The compiler should generate a map of all symbols that it recognizes. The **symbol table** generated by the Byte Craft compiler follows the format shown in Listing 9.4.

Listing 9.4 Symbol table exerpt

```
SYMBOL TABLE

LABEL    VALUE LABEL         VALUE

CC         0000 | COPC          0000
COPR       07F0 | DDRA          0004
DDRB       0005 | IRQE          0007
IRQF       0003 | IRQR          0001
ISCR       000A | LOCAL_START   00EB
```

The symbols listed are declared variables and functions, and preprocessor symbols. Identifiers declared by other means, such as `#pragma` statements, also appear. This is an inventory of all identifiers understood by the compiler.

Desktop programmers don't usually deal with a pointer's actual value. Typically, they assign the address of an object to a pointer variable, and manipulate the pointer (increment or decrement). The actual number is best left unknown, because it will change.

Since code and variables will not be relocated on an 8-bit embedded system, and since RAM is precious, it is more useful to examine RAM allocation in the embedded environment.

Listing 9.5 RAM usage map excerpt

```
RAM USAGE MAP

0050 use_metric                    signed char
0051 buffer                        unsigned char[6]
0000 CC                            register cc
0000 PORTA                         portrw
0001 PORTB                         portrw
0057 temp                          unsigned long      0100 0114
0051 buffer                        unsigned char[6]
005D hours                         unsigned char      011A 01DE
005E minutes                       unsigned char      011A 01DE
```

This report presents all the symbols that have memory allocated for their values, and the location of each. This is the location returned by the & (address-of) operator. Local variables are listed with the program range where the variable is in scope.

The compiler should give you an overall ROM usage count. This is the acid test for programmers and compilers: can a different code passage, a different theoretical approach, or a different method of optimization save a few extra bytes of ROM?

The program listing itself can be customized. As a convenience, the compiler can list **execution times** for each opcode. You can count them to gauge how long an interrupt service routine runs, for example. This information can in turn help you calibrate timing-dependent functions.

In the Byte Craft compilers, one helpful listing file option outlines the nesting level of each block of C statements, as the compiler understands

them. A similar option reveals the hierarchy of function calls in a separate report.

```
#pragma option NESTINGLEVEL;
#pragma option CALLMAP;
```

The most useful aspect of `CALLMAP` is to determine how much of the stack is used. The compiler takes a static setting for the depth of the stack. Using `CALLMAP` and your knowledge of the system, you can tailor stack size to save unused space.

The compiler can also present the values that it knows are held in the processor registers. If you are working without the benefit of an emulator, this provides some of the information an emulator would track.

9.7— Dummy Loads

One way to test the software of a microcontroller is to cause the controller to operate within a **dummy load** environment. This is a hardware technique more than a software chore, but the gist of it is to replicate with simple buttons, relays, and lights each external component of the target system. Using your knowledge of how the target system should behave, you can recreate the signals expected by the controller and watch for the controller to react.

9.8— Working with Emulators and Simulators

After a program is compiled, it must be tested using a **simulator** or an **emulator.**

9.8.1— Simulators

A simulator is a host-based or desktop software application that evaluates a program designed for an embedded target machine. The simulator recreates the running conditions of the target machine and interprets the executable.

Using a simulator, you can step through your code while the program is running. The simulator will report on register and status values, peripheral register contents, and RAM usage.

Since simulators are not hardware-based, they lack the particular character of a physical electrical device. A simulator can be written according to the microprocessor documentation, and therefore will omit any hardware quirks introduced in fabrication.

9.8.2— Emulators

An **emulator** is a hardware device that behaves electrically and logically like a target processor. It may include a similar processor, but with extra programming to support development host control and communication. The emulator has a link to the development system, to provide a window into the device under test. Since microcontrollers usually contain the ROM and RAM the system needs, this too is under external control.

Emulators work best when the program being inspected is unaltered from its intended production version, though this is not always possible for reasons explained in the following text.

Common emulator features include the following.

• Capability to set **breakpoints**

Good emulators set breakpoints based on an "external" table of addresses. When emulated execution arrives at the location, the breakpoint stops execution and waits for user intervention.

The alternative is to rewrite the program: an emulator might save the value at the breakpoint location and write in a software interrupt instruction. The software interrupt will in turn invoke management code that returns control to the emulator host.

• Support to examine and change registers and memory locations

Once in a breakpoint, the emulator will report on the internal state of the target processor, nondestructively.

• Trace buffers to analyse bus traffic

While not directly software-related, an expensive emulator will give detailed information on the electrical and timing signals presented to the target processor.

One particular challenge in debugging and testing via emulator is a frequently-invoked interrupt. An interrupt that happens too often or is too short-lived will lap the emulator easily. Only high-end emulators with extensive trace buffers can properly record the execution of these events.

Another challenge grows from the advances in semiconductor packaging. In-circuit **emulators** need to attach to a target system in place of a microcontroller. MCU packaging has shrunk from DIP-sized (often socketed) to tiny surface-mount parts. The required stable physical connection is increasingly difficult to engineer.

The issue with external emulators is cost; the specialized hardware is low-volume, high-complexity, and therefore expensive. Emulators deal with the external signals of the MCU: they may sacrifice speed to adopt a simple

manipulation technique, or may provide real-time signal emulation and monitoring at a tremendous increase in complexity and cost.

There are two ways to resolve the cost issue.

1. Less complex than an emulator that replaces the microcontroller, a **ROM emulator** replaces an *external* program memory device in your target system. It responds to instruction fetches by returning the opcodes of your program, and can insert software interrupts at any point. Furthermore, it can also provide the monitor code needed by the target microprocessor to service the breakpoints.

2. Many new MCU designs are incorporating **on-chip emulation** facilities into each production device. The aim here is to build a complete prototype with a normal sample or production processor permanently in place. Rather than use a specialized emulation device, developers can use built-in emulation facilities to interrogate the processor.

The link to the controlling host is provided by a 2- to 4-pin serial interface. On the prototype, the emulation signals are routed to a header strip, and a small cable and jack can provide the link to the host, perhaps through a serial port. The final design will probably not feature the header, unless it is needed to provide access to field engineers; the traces can be left in with little worry.

9.9— The Packaging of Embedded Software

An embedded program is usually compiled into a proprietary hexadecimal or binary representation. This output is suitable for the following.

• Download to a programming device

For testing and short runs, individual parts with programmable ROM may have the binary image created by the compiler burnt into them.

• Submitting for masked part production

For long runs, a fabrication facility can write the binary information into the masks used for silicon production. Each part is created with ROM cells set according to the binary image.

Chapter 10—
Sample Project

This chapter covers technical topics about the thermostat project not previously discussed.

Source code for the thermostat is available on the CD. If you wish to build the thermostat, detailed information is available on the CD. This chapter comments on several technical topics in detail, but the discussion will be helpful in other projects as well.

Updates and revised information is available via the website at
http://www.bytecraft.com/embedded_C/

10.1—
Hardware Exercise Programs

These are the programs that were used to test the thermostat hardware. We wrote them to get to know the challenges the board would impose. They are good examples to enter and modify, to experiment with C and the JICS emulator.

10.1.1— "Hello World!"

Since we don't have any indicator LEDs on the thermostat board, we toggle one of the heating/cooling unit relays. The LCD library was not yet configured.

Listing 10.1 "Hello World!" through a relay

```
#pragma option s5;  /* map file for jics */
#pragma option f 0; /* no page breaks in listing file */

#include <705j1a.h>
#include <port.h>

unsigned long counter;

void pause(void)
{
  for(counter = 0; counter < 255; counter++)
    {
      NOP();
    }
}

void main(void)
{
  PORTB.0 = 0;
  DDR_MASKED(PORTB,_______C,00000000);
  DDR_WAIT();

  while(1)
    {

      pause();
      PORTB.0 = 1;
```

```
    pause();
    PORTB.0 = 0;
  }
}
```

10.1.2— Keypad Test

Next we configure the keypad. Depending upon your hardware setup, the keypad library may require customization. In our example, it required some modification.

Listing 10.2 Keypad test program

```
#pragma option s5;  /* map file for jics */
#pragma option f 0; /* no page breaks in listing file */

#include <705j1a.h>
#include <delay.h>
#include <port.h>

#define KEYPAD_PORT PORTA
#define KEYPAD_DDR_REGISTER DDRA
#include <keypad.h>

void main(void)
{

  int8 store;

  /* must keep LCD_E low */
  PORTB = 0;
  DDR(PORTB,00000000);
  DDR_WAIT();

  keypad_init();

  while(1)
    {
```

```
      switch(keypad_getch())  {
      case '0'      : PORTB.0 = 1; break;
      case '6'      : PORTB.0 = 0; break;
      case '#'      : PORTB.0 = ~PORTB.0;
      }
    }
}

#include <keypad.c>
#include <port.c>
#include <delay.c>
```

10.1.3— LCD Test

This is a simple program for testing the LCD display.

Note the configuration needed by the LCD library. The symbols and possible values are documented in the library reference materials and in the file `lcd.h`.

Listing 10.3 LCD test program

```
#pragma option s5;  /* map file for jics */
#pragma option f 0; /* no page breaks in listing file */

#include <705j1a.h>
#include <delay.h>
#include <port.h>

#define LCD_DL 0
#define LCD_UPPER4 1
#define LCD_DATA PORTA
#define LCD_RS PORTB.2
#define LCD_RW PORTB.3
#define LCD_E PORTB.4
#define LCD_CD DDRB
#define LCD_CDM ___CCC__
#include <lcd.h>
```

```
void main(void)
{
  lcd_init();

  while(1)
    {
      puts("Hello World");
      delay_100us(10);
      lcd_send_control(LCDCLR);
      delay_100us(10);
    }
}

#include <lcd.c>
#include <delay.c>
```

10.2— Talking to Ports

One of the most challenging aspects of working with libraries is ensuring that they work with each other when sharing ports. Should a library not assume complete control of the ports it needs, and, more importantly, **leave them in a stable state,** you run the risk of misdriving the external devices.

The typical character-based LCD interface uses

• eight or four wires for data transfer,

• one wire for command select or data select,

• one wire for read or write, and

• one wire for enable.

In the thermostat design, the data wires of the LCD display are multiplexed with four wires of the keypad matrix.

These are the guidelines we devised for keeping accesses of both the keypad and LCD organized.

• Ensure the LCD enable line is disabled after writing or reading data. This was accomplished by quick code inspection.

• Determine the routines that require port direction setup. The `lcd_read()` and `lcd_write()` functions required data direction setup, as they actually drive the LCD interface; other library routines such as

`lcd_set_address()` use these functions, and therefore don't need their own port direction setup.

Even though `keypad_getch()` uses `keypad_kbhit()`, they both need data direction setup. `keypad_kbhit()` is intended for the user's own polling loops; however, `keypad_getch()` does not return until a key is pressed.

10.3— A/D Converter Theory

This design features a simple A/D converter circuit, in place of a dedicated converter peripheral as described in Chapter 3. Removing the requirement for an integrated A/D peripheral opens up the number of part choices.

The main feature of this device is that it is inexpensive, an important consideration for a mass-produced device. The tradeoff is that it is software-intensive.

This is the circuit. Please note that Ri is a thermistor.

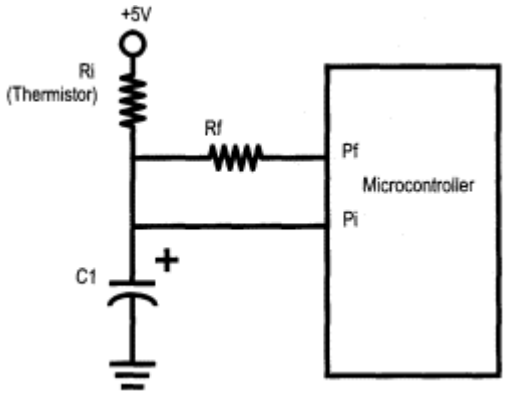

Figure 10.1
A/D converter circuit

The A/D converter assumes that the input impedance of an embedded microprocessor port is relatively high, and that the switch point remains constant with little hysteresis.

It also assumes that the junction between Ri and Rf is a current-summing junction, with capacitor C1 integrating the error current. The microprocessor has the ability to modulate the current through Rf by sending a pulse stream out of the Pf port bit. The ratio of the total number of ones to total bits emitted is a function of the average voltage on Pf. Consider the microprocessor as a high gain op-amp that attempts to keep voltage at the summing junction on the threshold of Pi low to high sense voltage.

Physically, Pf is PORTB bit 5 and Pi is the IRQ input, disabled as an interrupt source. Pi must be reset when it latches, but is in other ways like an input bit. To get an idea of the A/D) converter input range, run the following code on the thermostat.

Listing 10.4 Simple A/D driver code

```
#include <705j1a.h>
#pragma mor @0x7F1 = LEVEL;
#include <port.h>

#define Pf PORTB.5
#define Pi ISCR.IRQF

void main (void)
     {
        DDRB = 00000000;
        Pf = 0;
        ISCR.IRQE = 0; /* No interrupts please */
        ISCR.IRQR = 1; /* Reset IRQF/Pi to start */
        while(1)
           {
              Pf = Pi; /* If using a normal bit for input, invert */
              if(Pi) ISCR.IRQR = 1; /* reset the Pi latch */
           }
     }
```

Scope pin Pf, and warm or cool the thermistor.

This mode is actually using the microcomputer as a high-gain operational amplifier. The scope will show a pulse stream whose duty cycle will vary with input voltage from Ri. The ratio of zeros on the scope trace to the total time is a direct function of input voltage. It is this ratio we ultimately want to measure using software.

The range of the input voltage that can be measured is dependent on the sense voltage (Vs) of the input port, the output voltage of Pf in high and low states (Vh and Vl), and the value of the resistors Ri and Rf. The following

equations determine the minimum and maximum input voltage that can be read by the A/D converter.

```
Vmin = (Vh - Vs) * (Ri / Rf)
Vmax = (Vs - Vl) * (Ri / Rf)
```

The value of Vmin occurs when Pi is consistently just at the sense threshold, and the processor is always feeding back a 1 to the Pf pin. At an input of Vmax, a 0 is always being fed back from Pf. The A/D value is linear and scaled between Vmin and Vmax. It is determined from the ratio of 1s read on Pi (N1) to the total tests in a sample. The accuracy of the system is a linear function of test sample size (N). Vi can be calculated using the following relationship.

```
Vi = (N1 / N) * (Vmax - Vmin)
```

The value of C1 is not critical, it is used to control the slew rate and noise immunity of the system. For a typical system measuring an input from 0–5 volts, start with 47K resistors and a .01–.1 micro-farad capacitor.

Finally, ratiometric measuring systems like this one provide conversion accuracy that is a function of conversion time, and results can be easily scaled to the application. This eliminates conversion multiplies and divides created by changing the sample size.

Appendix A—
Table of Contents

Appendix A—
Embedded C Libraries

Introduction

Pressure to cut development costs leads naturally to the urge to standardize hardware and software products. Standardized computers led to standardized development languages and (quasi-) standardized operating systems. As well, developers created standard libraries of useful functions with widespread appeal.

In contrast, the popular notion of 8-bit embedded systems is that each new design is a one-of-a-kind programming task. The variety of applications doesn't lend itself to standard hardware. Only in latter years have compilers equalled and surpassed hand-coded assembly efficiency. Finally, the intimate level of programming forbids making any assumptions about third-party software.

Our experience is that programming 8-bit systems can take advantage of the development practices that evolved for mainstream computer systems. Even though the architectures vary, embedded hardware is standardized, functionally speaking. For instance: I/O facilities have port-pin features, such as selectable tristate, but in a limited number of permutations. As well,

controllers often use highly standardized buses like SPI or CAN: even though the interfaces differ, the expected results remain similar.

This relative similarity in hardware leads to standardized development languages. We have found that the vast majority of embedded applications can be implemented in C, and compiled for more than one of the leading microcontroller architectures on the market. Just as in desktop computing development, choosing a standard development language loosens your dependence on a specific architecture and supplier. This in turn can provide downward pressure on costs.

What remains largely unexplored is the feasibility of standardized C libraries for the 8-bit environment. Can they play the same role in embedded systems as they do in desktop computer software development? The ideals they represent are attractive.

Reduced Time to Market This is a simple savings in keystrokes per product. Libraries represent necessary steps already taken.

Reusable Code Libraries represent predigested knowledge, an investment in a well known, well structured, and well documented body of code. The return arrives with the reduced time and effort needed to customize or configure them. In C, configuration is a matter of answering a few questions using `#defines`.

Product Reliability Each development project that reuses a library can reinspect it for quality assurance. Since each user of the libraries should have access to the source code, local customizations and fixes can be integrated into the libraries for posterity. Reinventing the wheel each time disrupts a potentially valuable revision history or paper trail.

The downside, of course, is the challenge of reconciling a wide range of unforseen applications into an authoritative standard.

Working with libraries themselves is not a problem. Software that performs multiplication, division, or modulus is best supplied as an external set of library functions, which the compiler reads in as necessary. However, there is little debate about the design of the intended functionality: being operators, they have the most common calling interface of all.

The interface presents the largest stumbling block. Extended mathematics and peripheral functionality are the targets that need a standard functional interface and library implementation. Floating-point practices, 8-bit

implementation tradeoffs, and logical division of functionality are all likely points of contention.

The challenge is to find a robust general interface that accomodates some embedded-specific needs.

Efficient Function Calls Eight-bit architectures with little stack space are not candidates for frivolous function calling. The formal parameters of a library call will always include one too many values for some users.

If you make the reasonable assumption that there will not be more than one compiler at work on a project, the physical part of function invocation has no unknowns. The compiler can do anything to overcome the limits on resources of the target device.

Physical Differences Underlying Logically Similar Functions Input and output bits are likely to represent the actual voltage levels on I/O pins, but there is no consensus for data direction settings.

C can easily accommodate symbolic changes: see the port library for an excellent abstraction.

External Design Decisions This one is not so easily dismissed. If two peripherals are multiplexed on one port, as is the case with the thermostat, they can cause mutual interactions that a standard library might not contemplate. C can easily accommodate **multiple levels** of symbolic changes, but the design challenge moves from tricky to inscrutable.

The latter point is one of the reasons why it's important to ship the library source code with the compiler. Product reliability, discussed previously, is another. Fortunately, contemporary software industry practice, from a business point of view, permits, and even encourages, the distribution of source code. Byte Craft realized early on the importance of shipping library source with each compiler.

The subsequent sections outline a robust standard library interface. At this point, the libraries are useful and portable. We have obeyed the C (desktop) library interface as closely as possible, where needed.

Using the Libraries

You can easily use the libraries in your programs with the following steps.

• Add the `include` subdirectory to your environment's `INCLUDE` environment variable (the full path names will vary depending upon your instal-

lation). Alternatively, specify the `include` subdirectory on the command line with the `n=` command-line option.

• Add the `lib` subdirectory to your environment's `LIBRARY` environment variable (the full path name will vary depending upon your installation). Alternatively, specify the `lib` subdirectory on the command line with the `t=` command-line option.

• Use `#include <>` to add their header files at the top of your source code. For example:

```
#include <stdio.h>
/* your main function and other code */
```

This is referred to in the compiler manual as Absolute Code Mode. The compiler will search for a matching library file for every header file included at the top of your source.

Device Header Files and Definition Files

The Code Development System relies upon header files for definitions and constants. These often vary between part numbers. They are usually named for the part to which they apply, with a `.h` extension.

For more information, see "Library Definitions" on page 127.

Math Library

The math library for the Code Development System is contained in a file whose name matches the name of the product. It is usually supplied in source form, but with a `.lib` file extension. Thus, the compiler can read it in and compile it when necessary.

The math library supplies functions to implement the `*`, `/`, and `%` operators on 8- and 16-bit values. The relevant function names are as follows.

Operator	Functions
*	__MUL8x8(void) __MUL16x16(void)
/	__DIV8BY8(void) __LDIV(void)
%	__RMOD(void)

To adjust the math routines to your liking, back up the library file and make your changes to it directly. For instance: for a Code Development System product named ABC, the math library file itself would be `ABC.LIB`.

It is not necessary to `#include` this library, because the compiler will automatically include it if necessary. It searches for the library

• in the current directory and

• along the `LIBRARY` path.

Accordingly, it is important to have the Byte Craft library subdirectory in your `LIBRARY` path.

Library Definitions

DEF.H

Note

The name of the definitions header will change between CDS products. Look for a file named *`abc`*`_def.h`, where *`abc`* is the name of the CDS product.

Description

The definitions header is useful for compiling libraries.

When writing libraries of common code, you may not know for which target part to compile. Without including a device header file, you cannot write code using the standard identifiers that make your routines easier to read and maintain.

The solution to this dilemma is to include the library definitions header in place of any specific device header. The library definitions file defines all the standard identifiers present in each device header.

When compiling your library to an object file, Byte Craft compilers will ignore the values defined in the definitions file, preserving only the identifiers. During the linking process, the compiler will link the identifiers to the actual values specified in the particular device header file.

Example

This example assumes you will use Absolute Code Mode (i.e., not using *BCLink*). If you do link libraries with *BCLink*, remember to properly declare library functions as `extern`. The presence of the `MAKEOBJECT` definition can help you decide to do so conditionally.

When writing the library `my_library. lib`, include the `def.h` header file.

```
#pragma library
#pragma option +l  /* keep library code in the listing */

#include <abc_def.h>

void my_func1(void)
{
    PORTO.1 = 0; /* uses general definition in abc_def.h */
}

#pragma endlibrary
```

Compile the file to an object file, rename the object file with a . `lib` extension, and place it in a directory in the `LIBRARY` path.

Create a library header file.

```
void my_func1(void);
```

Save the file as `my_library. h`, in a directory in your `INCLUDE` path.

Create your program source file and include both the device header and the library header file.

```
#include <specific_device.h>
#include <my_library.h>

void main(void)
{
/* . . . */
my_func1();
/* . . . */
}
```

Compile the program source file as usual.

STDIO

STDIO.H and STDIO.C

Name

`stdio` is standard input and output functions.

Description

`stdio` is a good example of the way C can make embedded programming more palatable. Though an operating system with streams is not generally possible on an 8-bit microprocessor, programmers can call some of the familiar functions to perform input and output operations to the predictable devices.

`stdio` can also provide embedded interpretations of more complex functionality. One possibility that has been briefly investigated is a `scanf()` function that reads characters from the user-supplied `getch()`, and evaluates keycodes against template characters in a buffer ('`0`' for digits, '`a`' for letters, and so on). A trial implementation consumed about 200 bytes of ROM.

gets and puts

Name

`gets()` and `puts()` input and output strings.

Synopsis

```
#define BACKSPACE . . .
#include <stdio.h>
void puts(char far * str);
void gets(char near * str, int8 size);
```

Description

`puts()` outputs a null-terminated string to a device understood to be the standard output.

`gets()` retrieves a line from a device understood to be the standard input, and places it in the buffer `str`, which has size `size`. It retrieves characters up to a newline or carriage return, or to `size - 1`. It zeros the last position of the buffer.

Defining the symbol `BACKSPACE` to a character allows `gets()` to backtrack when it receives `BACKSPACE` from `getch()`. `gets()` actually uses `BACKSPACE` to perform the backtrack, so the `getch()` device must provide `BACKSPACE`, and the `putch()` device must understand `BACKSPACE` to be a character that moves the input point or cursor back one space.

These routines rely upon the library functions `getch()` and `putch()`, which must be declared elsewhere. Possible definitions for `getch()` and `putch()` are

- `keypad_getch()` in the `keypad` library,
- `lcd_getch()` and `lcd_putch()` in the `lcd` library, or
- `uart_getch()` and `uart_putch()` in the `uart` library.

STDLIB

STDLIB

Name

`stdlib` is a library of standard functions.

Description

`stdlib` holds a variety of useful utility functions.

rand and randmize

Name

`rand()` and `randmize()` generate pseudorandom numbers.

Synopsis

```
#include <stdlib.h>

#define SEED 0x3045 /* Seed must not be 0. */
#define srand(SEED) Rand_16=SEED
#define randmize() Rand_16=RTCC
int16 rand(void);
```

Description

`rand()` provides and manages a pseudorandom number sequence.

`randmize()` initializes the pseudorandom number sequence.

To initialize the pseudorandom number sequence, call `randmize()` in your initialization procedures. Then, call `rand()` for each new random number.

The current random number is stored in a static-duration data object, and is updated on each call to `rand()`.

Requirements

Requires a part header file or definitions file and the string library.

`abs` and `labs`

Name

`abs()` and `labs()` determine the absolute value.

Synopsis

```
#include <stdlib.h>
int8 abs(int8 i)
int16 labs(int16 l)
```

Description

`abs()` accepts a signed word value and returns the absolute value as a positive signed word value.

`labs()` accepts a signed `int16` value and returns the absolute value as a positive signed `int16` value.

ui16toa, ui8toa, i16toa, and i8toa

Name

`ui16toa()`, `ui8toa()`, `i16toa()`, and `i8toa()` convert unsigned or signed integers to ASCII representations.

Synopsis

```
#include <stdlib.h>
void ui16toa(unsigned int16 value,char near * str,
             unsigned int8 radix);
void ui8toa(unsigned int8 value,char near * str,unsigned int8 radix);
void i16toa(int16 value,char near * str,unsigned int8 radix);
void i8toa(int8 value,char near * str,unsigned int8 radix);
```

Description

`ui16toa()` converts an unsigned `int16` integer to a null-terminated ASCII string. It accepts a pointer to a string buffer, a value to be converted to a string representation, and the radix in which to represent the number.

`radix` may be one of the following values. The string buffer must be long enough to contain all characters created by the conversion. Therefore, the buffer must be sized accordingly.

Radix	Representation	Required Buffer Size
2	Binary	16 characters
8	Octal	6 characters
10	Decimal	5 characters
16	Hexadecimal	4 characters

`ui8toa()` is similar to the `ui16toa()`, except that it translates unsigned word values (8 bits). Therefore, the space requirements for the output buffer are as follows.

Representation	Required Buffer Size
Binary	8 characters
Octal	3 characters

(table continue on next page)

(table continued from previous page)

Representation	Required Buffer Size
Decimal	3 characters
Hexadecimal	2 characters

`i16toa()` converts a signed `int16` integer to a null-terminated ASCII string. It accepts a pointer to a string buffer, a value to be converted to a string representation, and the radix in which to represent the number.

`radix` may be one of the following values. The string buffer must be long enough to contain all characters created by the conversion. Furthermore, a negative value has a minus sign (–) prepended to it. Therefore, the buffer must be sized accordingly.

Radix	Representation	Required Buffer Size
2	Binary	16 characters
8	Octal	7 characters
10	Decimal	6 characters
16	Hexadecimal	5 characters

`i8toa()` is similar to the `i16toa()`, except that it translates signed word values (8 bits). Therefore, the space requirements for the output buffer are as follows.

Representation	Required Buffer Size
Binary	8 characters
Octal	4 characters
Decimal	4 characters
Hexadecimal	3 characters

`ahtoi16`, `ahtoi8`, `atoi16`, and `atoi8`

Name

`ahtoi16()`, `ahtoi8()`, `atoi16()`, and `atoi8()` convert an ASCII string value representing a decimal or hexadecimal number into an integer.

Synopsis

```
#include <stdlib.h>
unsigned int16 ahtoi16(char near * str);
unsigned int8 ahtoi8(char near * str);
int16 atoi16(char near  *str);
int8 atoi8(char near * str);
```

Description

`ahtoi16()` converts a null-terminated ASCII string representing an unsigned hexadecimal number into a `int16` integer value.

`ahtoi8()` converts a null-terminated ASCII string representing an unsigned hexadecimal number into a word integer value.

`atoi16()` converts a null-terminated ASCII string representing a signed number into a signed `int16` value.

The string should be in one of the following forms.

`-0b1000000000000000` to `0b1111111111111111`	Binary
`-0o100000` to `0o17777`	Octal
`-0100000` to `0177777`	Octal
`-32768` to `65535`	Decimal
`-0x8000` to `0xffff`	Hexadecimal

`atoi8()` converts a null-terminated ASCII string representing a signed number into a signed word value.

The string should be in one of the following forms.

`-0b10000000` to `0b11111111`	Binary
`-0o200` to `0o377`	Octal
`-0200` to `0377`	Octal
`-128` to `255`	Decimal
`-0x80` to `0xFF`	Hexadecimal

qsort

Name

`qsort()` quicksorts an array in place.

Synopsis

```
#include <stdlib.h>
void qsort(void near * base,size_t nelem, size_t size);
```

Description

`qsort()` sorts the elements of an array. The elements are left in place.

The function accepts a pointer to the array, a number of elements in the array (`nelem`) and a size of each element (`size`). `nelem` and `size` are of type `size_t`, which is defined in `string.c`.

`qsort()` compares the array elements using an external function that must have been defined as

```
#define QSORT_COMPARE(arg1, arg2)
```

If not defined, `QSORT_COMPARE` defaults to `strcmp()` in `string.c`. `QSORT_COMPARE` must accept two pointers and return an `int8` value. The return value must be:

• < 0 if the first argument is less than the second,

• = 0 if the first argument is equal to the second, or

• > 0 if the first argument is greater than the second.

pow

Name

`pow()` raises a number to an exponent.

Synopsis

```
#include <stdlib.h>
unsigned int16 pow(unsigned int8 base, unsigned int8 exponent);
```

Description

This function raises `base` to the power `exponent`.

STRING

STRING.H and STRING.C

Name

`string` performs operations on null-terminated and known-length strings.

Description

Routines in this library perform operations on both null-terminated and known-length string buffers.

size_t

Name

`size_t` is the type for "size of" variables.

Synopsis

```
#include <string.h>
typedef unsigned int8 size_t;
```

Description

Byte Craft libraries accept "size of" parameters as type `size_t`. A `size_t` parameter usually represents the size of another parameter or object.

memcpy, memchr, and memcmp

Name

`memcpy()`, `memchr()`, and `memcmp()` copy, search, and compare buffers.

Synopsis

```
#include  <string.h>
void memcpy(char near * dest,const char far * src,size_t n);
void * memchr(const void * s,int8 c,size_t n);
int8 memcmp(unsigned char far * str1,unsigned char far * str2,
            size_t n);
```

Description

`memcpy()` copies `n` bytes of memory from location `src` to location

`memchr()` searches an array for a character. It begins at address `s`, and searches for the first element of the array of size `n` that equals `(unsigned char)c`. It returns the address of the matching element, or a null pointer if no match was found.

`memcmp()` compares two arrays of `unsigned char`, `str1`, and `str2`, to find differences between them. If all elements are equal, `memcmp()` returns `0`.

Where a difference occurs, if the element of `str1` is greater than that of `str2`, `memcmp()` returns a positive value. If the element of `str1` is less than that of `str2`, `memcmp()` returns a negative value.

Both arrays must be of size `n`.

`strcat`, `strchr`, and `strcmp`

Name

`strcat()`, `strchr()`, and `strcmp()` copy, search, and compare null-terminated strings.

Synopsis

```
#include <string.h>
void strcat(char near * dest,char far * src);
void * strchr(const void * str,int8 c);
int8 strcmp(unsigned char far * str1,unsigned char far* str2);
void strcpy(char near * dest,char far * src);
```

Description

`strcat()` copies elements of the null-terminated string `src`, including its null termination character, to the array `dest`.

`strchr()` searches the null-terminated string `str` for the first occurrence of `(char)c`. `strchr()` examines the terminating null of `str` as part of the string. `strchr()` returns a pointer to the matching character of `str`, or a null pointer if no match was found.

`strcmp()` compares two null-terminated strings, `str1` and `str2`, to find differences between them. If all elements are equal, `strcmp()` returns `0`.

Where a difference occurs, if the element of `str1` is greater than that of `str2`, `strcmp()` returns a positive value. If the element of `str1` is less than that of `str2`, `strcmp()` returns a negative value.

If one string is shorter than the other, `strcmp()` does not finish the longer string.

`strcpy()` copies the null-terminated string `src`, including terminating null, to the array of char pointed to by `dest`.

strlen

Name

`strlen()` determines the length of a null-terminated string.

Synopsis

```
#include <string.h>
unsigned int8 strlen(char far * str);
```

Description

`strlen()` returns the number of characters in the null-terminated string `str`. The count does not include the terminating null character.

strset, strupr, and strlwr

Name

`strset()`, `strupr()`, and `strlwr()` reinitialize or convert a null-terminated string.

Synopsis

```
#include <string.h>
void strset(char near * str,char ch);
void strupr(char near * str);
void strlwr(char near * str);
```

Description

`strset()` stores `(unsigned char)ch` in each of the elements of the array pointed to by `str`.

`strupr()` converts all lowercase characters in the null-terminated string `str` to uppercase. It converts the string in place.

`strlwr()` converts all uppercase characters in the null-terminated string `str` to lowercase. It converts the string in place.

CTYPE

CTYPE.H

Name

`ctype` routines operate on characters.

Description

Routines in this library perform type recognitions and conversions on characters.

isxyz, toascii, tolower, and toupper

Name

`isalnum()`, `isalpha()`, `isascii()`, `iscntrl ()`, `isdigit()`, `islower()`, `isupper ()`, `isxdigit()`, `toascii()`, `tolower()`, and `toupper()` evaluate and convert characters.

Synopsis

```
#include <ctype.h>
int8 isalnum(int8 ch);
int8 isalpha(int8 ch);
int8 isascii(int8 ch);
int8 iscntrl(int8 ch);
int8 isdigit(int8 ch);
int8 islower(int8 ch);
int8 isupper(int8 ch);
int8 isxdigit(int8 ch);
#define toascii(CH) CH&0x7f
int8 tolower(int8 ch);
int8 toupper(int8 ch);
```

Description

`isalnum()` evaluates the character `ch` and returns a nonzero value if it is a lowercase character (a–z), uppercase character (A–Z), or decimal digit (0–9). If not, it returns zero.

`isalpha()` evaluates the character `ch` and returns a nonzero value if it is a lowercase character (a–z) or uppercase character (A–Z). If not, it returns zero.

`isascii()` evaluates the character `ch` and returns a nonzero value if it is an ASCII character (high bit is 0).

`iscntrl()` evaluates the character `ch` and returns a nonzero value if it is an ASCII control character. (ASCII control characters include characters 0–31 and 127.) If not, it returns zero.

`isdigit()` evaluates the character `ch` and returns a nonzero value if it is a numeric digit (0–9). If not, it returns a zero.

`islower()` evaluates the character `ch` and returns a nonzero value if it is a lowercase character (a–z). If not, it returns a zero.

`isupper()` evaluates the character `ch` and returns a nonzero value if it is an uppercase character (A–Z). If not, it returns a zero.

`isxdigit()` evaluates the character `ch` and returns a nonzero value if it is a hexadecimal digit (0–9, a–f, or A–F). If not, it returns a zero.

`toascii()` zeros the upper bit of `CH`.

`tolower()` evaluates `ch` and, if `ch` is an uppercase character, returns the corresponding lowercase character. Otherwise, it returns `ch` unchanged.

`toupper()` evaluates `ch` and, if `ch` is a lowercase character, returns the corresponding uppercase character. Otherwise, it returns `ch` unchanged.

DELAY

DELAY.H and DELAY.C

Name

`delay` routines cause embedded programs to wait.

Description

These routines provide a consistent interface for invoking delays.

Requirements

Requires a part header file or a definitions file.

delay_ms

Name

`delay_ms()` delays a number of milliseconds.

Synopsis

```
#include <delay.h>
void delay_ms(unsigned int8 ms);
```

Description

`delay_ms()` waits the specified number of milliseconds and then returns.

KEYPAD

KEYPAD.H and KEYPAD.L

Name

`keypad` drives a matrix keypad.

Description

The routines in this library operate a matrix keypad connected to a single, 8-bit I/O port.

Requirements

Requires the port and delay libraries.

keypad_getch and keypad_kbhit

Name

`keypad_getch()` and `keypad_kbhit()` scan for and get a character from a matrix keypad.

Synopsis

```
#define KEYPAD_PORT
#define keypad_debounce_delay() delay_ms(0x20)
#include <keypad.h>

unsigned char keypad_getch(void);
unsigned int8 keypad_kbhit(void);
```

Description

The user must define `KEYPAD_PORT` to the register used to read from and write to the port.

A default definition may be available. Consult the source for the keypad library.

The user must define a function KEYPAD_READ to set up KEYPAD_PORT for reading. The implementation will vary depending upon the circuitry of the keypad.

A default definition may be available, depending upon your Code Development System product. Consult the source for the keypad library.

`keypad_debounce_delay()` is called by `keypad_getch()`. If not redefined, `keypad_debounce_delay()` waits 20 milliseconds to debounce the keyboard.

`keypad_getch()` waits for a keypad contact, and returns the appropriate character from the array `keypad_table[]`.

If not defined elsewhere, `keypad_table` defaults to the standard telephone keypad.

```
const char keypad_table[]="123A"
                          "456B"
                          "789C"
                          "*0#D";
```

`keypad_kbhit()` looks for a keypad contact and returns `1` when a contact is made.

LCD

LCD.H and LCD.C

Name

`lcd` provides support for `lcd` controllers.

Requirements

Requires the port library.

Description

The LCD library provides routines to drive a Hitachi HD44780 LCD controller.

A typical LCD module configuration uses 3 wires for read/write, register select (command or data), and enable, and either four or eight wires for data transmission.

The module needs to be initialized by a sequence of writes that sets parameters, including the width of the data bus. This is accomplished by `lcd_init()`. After initialization, the LCD panel may occasionally be busy. `lcd_busy_check()` determines whether the module can accept new data.

`lcd_putch()` and `lcd_getch()` are intended to be used as `putch()` and, less likely, `getch()` for the `stdio` library.

Configuration

`lcd.h` defines a number of important constants for LCD Software Commands.

The following symbols need to be defined. Defaults are provided in `lcd.c`.

```
#define LCD_E_PORT   PORT1     /* LCD Enable */
#define LCD_E_PIN    2         /* LCD Enable */
#define LCD_DATA     PORT1
#define LCD_RS_PORT  PORT0     /* LCD Register Select */
#define LCD_RS_PIN   0         /* LCD Register Select */
#define LCD_RW_PORT  PORT0     /* LCD Read/~Write */
#define LCD_RW_PIN   1         /* LCD Read/~Write */
```

LCD_DATA

Name

`LCD_DATA_IN_CONTROL_OUT()` and `LCD_DATA_OUT_CONTROL_IN()` are macros to control the LCD data and control ports.

Synopsis

```
#define LCD_DATA_IN_CONTROL_OUT() ...
#define LCD_DATA_OUT_CONTROL_OUT() ...
#include <lcd.h>
```

Description

`LCD_DATA_IN_CONTROL_OUT()` sets the LCD data port for input.

`LCD_DATA_OUT_CONTROL_OUT()` sets the LCD data port for output.

`cd_init`, `lcd_send_control`, and `lcd_busy_check`

Name

`lcd_init()`, `lcd_send_control()`, and `lcd_busy_check()` initialize and control the LCD module.

Synopsis

```
#include <lcd.h>
void lcd_init(void);
void lcd_send_control (char control);
void lcd_busy_check(void);
```

Description

`lcd_init()` performs several LCD initialization tasks, including turning on the LCD display and cursor, clearing the display, and setting the display to increment mode.

`lcd_send_control()` (sends a control character to the LCD controller.

`lcd_busy_check()` waits until the busy bit of the LCD controller is clear. You can then safely write to the controller.

lcd_putch, lcd_getch, and 1cd_gotoXY

Name

`lcd_init()`, `lcd_putch()`, and `lcd_getch()` write to and read from the LCD module, and move the cursor.

Synopsis

```
#include <lcd.h>
void lcd_putch(char ch);
char lcd_getch(void);
void lcd_gotoXY(int8 x, int8 y);
```

Description

`lcd_putch()` writes a character to the LCD panel.

`lcd_getch()` reads a character from the LCD panel.

`lcd_gotoXY()` moves the LCD insert point to a specific character cell.

The cells are numbered as follows.

```
X  0  1  2  3  4   5  6  7  8  9 . . .
Y +--------------------------------
 0|
 1|
 . . .
```

Thus, to move the insert point to the final cell of the bottom row of a 2-line, 40-space panel, use

```
lcd_gotoXY(1,39);
```

I2C_EE

I2C_EE.H and I2C_EE.C

Name

I2C_EE provides useful routines for the I²C 24LC01B/02B serial EEPROM.

Description

I²C™ is a standard of Phillips Electronics N.V. It is a serial peripheral interface that operates across two wires. The two lines consist of the serial data line and the serial clock line, which are both bidirectional. It is synchronous.

It is a multimaster, multislave network interface with collision detection. Up to 128 devices can exist on the network. Each device has an address made up of several fixed bits (assigned by the I²C committee) and several programmable bits usually determined by pin connections. In this way, several identical devices can coexist within one system. Either 7- or 10-bit addressing is available.

There are also several reserved addresses for broadcasting to all devices and other expansion needs.

I²C has two speeds: In standard mode, 100 kbit/second, and in fast mode, 400 kbit/second. Effective data rates are dependent upon configuration and addressing mode used.

The standard does not specify a programming interface for controllers that implement it. This section deals exclusively with a serial EEPROM connected by I²C.

Requirements

Requires the port and delay libraries.

Configuration

To configure the I²C port, the following settings must be adjusted. If not changed, the I²C control (clock) line is bit 0 of port 1 and the data line is bit 5 of port 2.

```
#define I2C_PORT_DDR_READ() GPIO_CONFIG = PORTO_RESISTIVE | \
PORT1_CMOS | PORT2_RESISTIVE | PORT3_RESISTIVE; PORT2=0xff
#define I2C_PORT_DDR_WRITE()  GPIO_CONFIG = PORTO_RESISTIVE | \
PORT1_CMOS | PORT2_CMOS       | PORT3_RESISTIVE
#define I2C_PORT_DDR()  GPIO_CONFIG = PORTO_RESISTIVE | \
PORT1_CMOS | PORT2_RESISTIVE  | PORT3_RESISTIVE; PORT2=0xff

#define I2C_CONTROL PORT1
#define I2C_DATA PORT2
#define I2C_SCL 0
#define I2C_SDA 5

#define i2c_bus_delay() delay_ms(1)
```

I2C_write and I2C_read

Name

`I2C_write()` and `I2C_read( )` communicate over the I2C bus.

Synopsis

```
#include <i2c_ee.h>
void I2C_write(unsigned int8 address, unsigned int8 data);
unsigned int8 I2C_read(unsigned int8 address);
```

Description

`I2C_write()` writes the word `data` at the memory location `address` on the serial EEPROM.

`I2C_read()` reads the value at memory location `address`.

MWIRE_EE

MWIRE_EE.H and MWIRE_EE.C

Name

`mwire_ee` creates a MICROWIRE connection to a serial EEPROM.

Description

MICROWIRE and MICROWIRE/PLUS are a proprietary standard of National Semiconductor. In some implementations, they are SPI-compatible.

MICROWIRE/PLUS is a serial peripheral interface that operates across three wires. It is synchronous, relying on either an internal (to the bus master) or external clock. It is bidirectional. A chip-select signal must also be implemented.

The programming interface includes the following.

• A control register `CNTRL` that configures the interface (including the internally-generated shift rate)

• A read/write serial input/output register

These registers are memory-mapped.

The MICROWIRE Shift Clock (SK) is a factor of internal clock speed, dividing the system clock by 2, 4, or 8. Each byte transmitted or received by MICROWIRE requires 8 SK cycles.

Software can cause a transmit by setting the `BUSY` flag of the PSW (processor status word). The `BUSY` flag will clear when the transmit is complete. Some parts provide a vectored maskable interrupt when `BUSY` is reset.

The following routines deal directly with an EEPROM connected via MICROWIRE.

Requirements

Requires a device header file or a definitions file. Requires an external function as shown in the following text.

Configuration

You must define the following symbols before using the `mwire_ee` library. If not defined, default values are used.

`MWIRE_CONTROL`	port used to access the MICROWIRE control lines
`MWIRE_CLK`	pin used for clock
`MWIRE_CS`	pin used for chip select
`MWIRE_DATA`	port used to access the MICROWIRE data lines
`MWIRE_DO`	pin used for data output
`MWIRE_DI`	pin used for data input
`MWIRE_PORT_DDR_READ()`	macro setting port data direction for read
`MWIRE_PORT_DDR_WRITE()`	macro setting port data direction for write
`MWIRE_PORT_DDR()`	macro setting default data direction for MICROWIRE port

mwire_bus_delay

Name

`mwire_bus_delay()` is a user-defined delay function.

Synopsis

```
#include <mwire_ee.h>
void mwire_bus_delay() {
/* Your preferred delay code */
}
```

Description

To properly time the MICROWIRE bus, you must write a delay function to wait between half clock cycles. You can accomplish this by

• defining it as a function containing NOPs or

• define it as a call to a delay function.

mwire_enable, mwire_disable, mwire_write, mwire_read, and mwire_write_all

Name

`mwire_enable(), mwire_disable(), mwire_write(), mwire_read()`, and `mwire_write_all` () communicate over MICROWIRE.

Synopsis

```
#include <mwire_ee. h>
#define mwire_enable()
#define mwire_disable()
#define mwire_erase(ADDRESS)
void mwire_write(unsigned int8 address,unsigned int16 data);
unsigned int16 mwire_read(unsigned int8 address);
void mwire_write_all(unsigned int16 data);
```

Description

`mwire_enable()` and `mwire_disable()` enable and disable, respectively, the MICROWIRE connection to the serial EEPROM.

`mwire_erase()` erases the value at memory location `ADDRESS` on the serial EEPROM.

`mwire_write()` writes the value of data to the location `address` on the EEPROM.

`mwire_read()` reads and returns the value at location `address` from the serial EEPROM.

`mwire_write_all()` writes the same value to all locations of the serial EEPROM.

MATH

MATH.H and MATH.C

Name

`math` implements math functions.

Description

This library implements math functions.

Requirements

Requires `float.h`

acos, asin, atan, and atan2

Name

`acos()`, `asin()`, `atan()`, and `atan2()` are trigonometric functions.

Synopsis

```
#include <math.h>
float acos(float x);
float asin(float x);
float atan(float x);
float atan2(float y, float x);
```

Description

`acos()` returns the angle in radians (from 0 to pi) whose cosine is `x`.

`asin()` returns the angle in radians (from –pi/2 to pi/2) whose sine is `x`.

`atan()` returns the angle in radians (from –pi/2 to pi/2) whose tangent is `x`.

`atan2()` returns the angle in radians (from –pi to pi) whose tangent is `y/x`.

ceil and floor

Name

`ceil()` and `floor()` return the next higher or lower integer value.

Synopsis

```
#include <math.h>
float ceil(float x);
float floor(float x);
```

Description

`ceil()` returns `x` (if an integer), or the next higher integer value.

`floor()` returns `x` (if an integer), or the next lower integer value.

cos and cosh

Name

`cos()`, `cosh()`, `sin()`, `sinh()`, `tan()`, and `tanh()` are trigonometric functions.

Synopsis

```
#include <math.h>
float cos(float x);
float cosh(float x);
float sin(float x);
float sinh(float x);
float tan(float x);
float tanh(float x);
```

Description

`cos()` returns the cosine of `x`, where `x` is an angle in radians.

`cosh()` returns the hyperbolic cosine of `x`.

`sin()` returns the sine of `x`, where `x` is an angle in radians.

`sinh()` returns the hyperbolic sine of `x`.

`tan()` returns the tangent of `x`, where `x` is an angle in radians.

`tanh()` returns the hyperbolic tangent of `x`.

fabs

Name

`fabs()` calculates the absolute value of a floating point number.

Synopsis

```
#include <math.h>
float fabs(float x);
```

Description

`fabs()` returns the absolute value of `x`.

fmod

Name

`fmod()` calculates the remainder of `x/y`.

Synopsis

```
#include <math.h>
float fmod(float x, float y);
float frexp(float x, int * pexp);
float ldexp(float x, int exp);
```

Description

`fmod()` returns the remainder of `x/y`.

`frexp()` calculates a mantissa and exponent for the float value `x`. `frexp()` returns the mantissa and places the exponent in `*pexp`. The exponent is a power of 2.

`ldexp()` calculates a floating point value for the mantissa `x` and the exponent (of base-2) `exp`.

exp, log, and log10

Name

`exp()`, `log()`, and `log10()` calculate exponents and logarithms.

Synopsis

```
#include <math.h>
float exp(float x);
float log(float x);
float log10(float x);
```

Description

`exp()` returns the exponential of `x` (`e` raised to the power `x`).
`log()` returns the natural logarithm of `x`.
`log10()` returns the base-10 logarithm of `x`.

modf

Name

`modf()` calculates integer and fraction portions of a floating point number.

Synopsis

```
#include <math.h>
float modf(float x, float * pint);
```

Description

`modf()` calculates the integer and fraction portions of the value `x`, returns the fraction portion, and stores the integer portion in `*pint`. Both the integer and fraction portions have the same sign as `x`.

pow and sqrt

Name

`pow()` and `sqrt()` calculate a power or a root of a floating point number.

Synopsis

```
#include <math.h>
float pow(float x, float y);
float sqrt(float x);
```

Description

`pow()` returns `x` raised to the `y` power.

`sqrt()` returns the square root of `x`.

FLOAT

FLOAT.H

Name

`float` is a library of floating point definitions.

Synopsis

```
#include <float.h>
#define FLT_DIG
#define FLT_EPSILON
#define FLT_MANT_DIG
#define FLT_MAX
#define FLT_MAX_10_EXP
#define FLT_MAX_EXP
#define FLT_MIN
```

```
#define FLT_MIN_10_EXP
#define FLT_MIN_EXP
#define FLT_RADIX
#define FLT_ROUNDS
```

Description

If you employ floating point variables or operations, the file `float.h` provides some required definitions.

Definitions

`FLT_DIG` determines the number of digits of precision for float variables.

`FLT_EPSILON` determines the smallest possible nonzero value for a float variable.

`FLT_MANT_DIG` is the number of mantissa digits for float variables. The value is of base `FLT_RADIX`.

`FLT_MAX` is the largest possible value for a float variable.

`FLT_MAX_10_EXP` is an integer exponent. When 10 is raised to the power of `FLT_MAX_10_EXP`, the result is the largest power-of-10 value for a float variable.

`FLT_MAX_EXP` is an integer exponent. When `FLT_RADIX` is raised to the power of `FLT_MAX_EXP-1`, the result is the largest power-of-`FLT_RADIX` value for a float variable.

`FLT_MIN` provides the smallest possible value for a float variable.

`FLT_MIN_10_EXP` is an integer exponent. When 10 is raised to the power of `FLT_MIN_10_EXP`, the result is the smallest power-of-10 value for a float variable.

`FLT_MIN_EXP` is an integer exponent. When `FLT_RADIX` is raised to the power of `FLT_MIN_EXP-1`, the result is the smallest power-of-`FLT_RADIX` value for a float variable.

The exponent of `float` type values is an exponent of `FLT_RADIX`.

`FLT_ROUNDS` represents the rounding method used by floating point calculations. The following value for `FLT_ROUNDS` sets the accompanying rounding method:

1 The compiler will round toward the nearest representable value.

UART

UART

Name

UART provides UART functions in software.

Requirements

Requires a part header file or definitions file, and the port and delay libraries.

Definitions

The following settings are required for UART operation.

`UART_TD_PORT`

Users must define this as the port intended for UART transmission. By default, this is defined as `PORT1`.

`UART_TD_PIN`

Users must define this as the pin in `UART_TD_PORT` intended to drive the TD line. By default, this is defined as 1.

`UART_RD_PORT`

Users must define this as the port intended for UART reception. By default, this is defined as `PORT2`.

`UART_RD_PIN`

Users must define this as the pin in `UART_RD_PORT` intended to read the RD line. By default, this is defined as 4.

Variables

`uart_mode`

Configures the `uart` library at run time as described in the following text.

Configuration

Users must set the `uart_mode` variable with an `OR`ed combination of constants.

Baud Rate	Stop Bits	Parity	Data Bits
`BAUD_300`	`STOP_1`	`PARITY_NONE`	`DATA_7`
`BAUD_1200`	`STOP_2`	`PARITY_EVEN`	`DATA_8`
`BAUD_2400`		`PARITY_ODD`	
`BAUD_4800`			
`BAUD_9600`			
`BAUD_19200`			
`BAUD_38400`			
`BAUD_57600`			
`BAUD_115200`			

Example:

```
uart_mode = BAUD_115200 | STOP_2 | PARITY_NONE | DATA_8;
```

`uart_getch`, `uart_putch`, and `uart_kbhit`

Name

`uart_getch()`, `uart_putch()`, and `uart_kbhit()` perform UART I/O.

Synopsis

```
char uart_getch(void);
void uart_putch(char);
char uart_kbhit(void);
```

Description

`uart_getch()` gets a character from the UART.

`uart_putch()` outputs a character to the UART.

`uart_kbhit()` returns 1 if a byte is being received, or 0 if there is no data to be received.

PORT

PORT.H, PORT.C, and PORTDEFS.H

Name

`port` provides platform-independent port access.

Requirements

Requires a part header file or definitions file.

Description

This header file includes some useful functions for manipulating ports. Many Byte Craft libraries depend upon these definitions.

All single-chip MCUs have I/O ports of some nature. This library tries to smooth out the differences between their peculiarities.

`port.h` causes `portdefs.h` to be read in. `portdefs` includes definitions for each possible setting of a data direction register. In these definitions, '`I`' stands for "input" and '`0`' stands for "output." This is to resolve the question of which state (zero or one) stands for input or output. For example:

```
/* DDR uses 1 for output and 0 for input */
#define 00000000 0b11111111
#define 0000000I 0b11111110
  /* ... and so on ... */
#define 0000IIII 0b11110000
  /* ... and so on ... */
#define IIIIIII0 0b00000001
#define IIIIIIII 0b00000000
```

`portdefs` also includes definitions for bit masks to be used in `DDR_MASKED()`. In these definitions, '_' (underscore) means "no change", and '`C`' means change.

DDR(), DDR_MASKED(), and DDR_WAIT()

Name

DDR(), DDR_MASKED(), and DDR_WAIT() manipulate the data direction of a port.

Synopsis

```
#include <port.h>
DDR(port, direction)
DDR_MASKED(port, mask, direction)
DDR_WAIT()
```

Description

These functions manipulate a port's data direction. They use direction and mask definitions read in from portdefs.h.

DDR() accepts a port and direction definition, and configures the port's data direction register to operate accordingly.

DDR_MASKED() performs the same action, but only on the pins selected in the mask definition. DDR_MASKED() helps solve the conflict between several library routines addressing different bits on the same port. To change one or two bits, the compiler may use bit-change instructions if available, leaving the rest untouched. Otherwise, the compiler will preserve the state of masked-out DDR bits when it reads and modifies the DDR value.

DDR_WAIT() inserts a short delay to allow the data direction change to propagate.

Example

To set the bits of port PORTX to all output, invoke:

```
DDR(PORTX, 00000000); /* note letter '0', not zero */
DDR_WAIT();
```

To set the low and high nibbles to output and input, respectively, use:

```
DDR(PORTX, IIII0000); /* letters 'I' and '0' */
DDR_WAIT();
```

To set only bit 1 of PORTX to output, use:

```
DDR_MASKED(PORTX, _______C__, 00000000); /* other '0' bits don't
matter */
DDR_WAIT();
```

Appendix B—
ASCII Chart

It's always difficult to find an ASCII chart when you want one. Here is a chart of hex values and their ASCII meanings.

Table B.1 ASCII characters

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i

(table continue on next page)

(table continued from previous page)

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C		5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Appendix C — Glossary

A

accumulator
Also "A", "AC", or other names. The register that holds the results of ALU operations.

A/D
Analog to digital.

addressing mode
The math used to determine a memory location in the CPU, and the notation used to express it.

ALU
Arithmetic Logic Unit. Performs basic mathematical manipulations, such as add, subtract, complement, negate, `AND`, and `OR`.

`AND`
Logical operation in which the result is 1 if `AND`ed terms both have the value 1.

ANSI C
American National Standards Institute standards for C.

assembly language
A mnemonic form of a specific machine language.

B

bank
A logical unit of memory as determined by addressing modes and their restrictions.

bit field
A group of bits considered as a unit. A bit field may cross byte boundaries if supported by the compiler.

block
Any section of C code enclosed by braces, `{}`. A block is syntactically equivalent to a single instruction, but adds in a new variable scope.

breakpoint
A set location to stop executing program code. Breakpoints are used in debugging programs.

C

CAN
Controller Area Network, developed by Bosch and Intel. It is an intermodule bus that links controlled devices.

cast
*Also **coerce.*** Convert a variable from one type to another.

checksum
A value that is the result of adding specific binary values. A checksum is often used to verify the integrity of a sequence of binary numbers.

computer operating properly
Also COP. A peripheral or function that resets microcontroller function under questionable execution conditions. COP, as a word, is the name of the COP8 microcontroller product line from National Semiconductor.

cross assembler
An assembler that runs on one type of computer and assembles the source code for a different target computer. For example, an assembler that runs on an Intel x86 and generates object code for Motorola's 68HC05.

cross compiler
A compiler that runs on one type of computer and compiles source code for a different target computer. For example, a compiler that runs on an Intel x86 and generates object code for Motorola's 68HC05.

D

debugger
A program that helps with system debugging where program errors are found and repaired. Debuggers support such features as breakpoints, dumping, and memory modify.

declaration
A specification of the type, name, and possibly the value of a variable.

dereference
Also **** or indirection.*** Access the value pointed to by a pointer.

E

EEPROM
Electrically erasable programmable read only memory.

embedded
Fixed within a surrounding system or unit. Also, engineered or intended to perform one specific function in a specific environment.

endianness
The distinction of multibyte data storage convention. Little-endian stores the least-significant byte first in memory. Big-endian stores the most-significant byte first in memory.

G

global variable
A variable that can be read or modified by any part of a program.

H

hysteresis
The delay between the switching action of a control and the effect. Can be enforced to prevent rapid short-term reversals in the control's state.

I

index register
Also known as "X" or other names. The register used to hold a value that becomes a factor in an indexed addressing mode. Frequently used for arithmetic operations, though without as many capabilities as an accumulator.

interrupt
A signal sent to the CPU to request service. Essentially a subroutine outside the normal flow of execution, but with many extra considerations.

J

J1850
An intermodule bus endorsed by the SAE (Society of Automotive Engineers).

L

local variable
A variable that can only be used by a specific module or modules in a program.

logical operator
Operators that perform logical operations on their operands. For example, `!`, `&&`, and `||`.

M

machine language
Binary code instructions that can be understood by a specific CPU.

mask
A group of bits designed to set or clear specific positions in another group of bits when used with a logical operator.

maskable interrupt
Interrupts that software can activate and deactivate.

memory-mapped
A virtual address or device associated with an actual address in memory.

N

NOP
No operation. An instruction used to create a delay.

NOT
Logical negation. A 0 becomes a 1, and a 1 becomes a 0.

O

object code
Machine language instructions represented by binary numbers not in executable form. Object files are linked together to produce executable files.

operator
A symbol that represents an operation to be performed on operands. For example, `+`, `*`, and `/`.

OR
A Boolean operation that yields 1 if any of its operands is a 1.

P

paging
A page is a logical block of memory. A paged memory system uses a page address and a displacement address to refer to a specific memory location.

port
A physical I/O connection.

program counter
Also ***PC.*** A register that holds the address of the next instruction to be executed. The program counter is incremented after each byte of each instruction is fetched.

programmer's model
The description of registers that make up the microprocessor's visible interface. Includes the registers such as the accumulator and index register, program counter, and stack pointer.

PROM
Programmable read-only memory. ROM that can be programmed.

R

real time
A system that reacts at a speed commensurate with the time an actual event occurs.

register
A byte or word of memory that exists within the microprocessor proper. Registers directly interface to the ALU and other microprocessor functionality, as opposed to external RAM.

reset
To return the microcontroller to a known state. This operation may or may not alter processor registers, and memory and peripheral states.

ROM
Read only memory.

ROMable
Code that will execute when placed in ROM.

RS-232
A standard serial communication port.

S

SCI
Also UART (Universal Asynchronous Receiver Transmitter). SCI is an asynchronous serial interface. The timing of this signal is compatible with the RS-232 serial standard, but the electrical specification is board-level only.

SPI
Serial Peripheral Interface bus. A board-level serial peripheral bus.

scope
A variable's scope is the areas of a program in which it can be accessed.

shift
*Also **rotate,** with subtle differences between them.* Move the contents of a register bitwise to the left or right.

side-effect
An unintentional change to a variable, or the work of instructions within a function not directly related to the calculation of its return value.

simulator
A program that recreates the same input and output behaviour as a hardware device.

stack
A section of RAM used to store temporary data. A stack is a last-in-first-out (LIFO) structure.

stack pointer
A register that contains the address of the top of the stack.

static
A variable that is stored in a reserved area of RAM instead of in the stack. The area reserved cannot be used by other variables.

T

timer
A peripheral that counts independent of program execution.

U

UART
Universal asynchronous receiver transmitter. A serial-to-parallel and parallel-to-serial converter.

V

volatile
The quality of a value that changes unexpectedly. The compiler cannot trust that the value of a volatile variable remains constant over time, and therefore cannot perform certain optimizations. Declared explicitly by the programmer, or determined by the compiler.

W

watchdog (timer)
Another name for *computer operating properly* circuitry.

Index

D

E

F

H

I

K

L

M

N

P

Q

R

S

T

U

V

W

What's on the CD-ROM

The CD-ROM the accompanies **C Programming for Embedded Systems** includes a working C6805 Code Development System tailored for the Motorola MC68705J1A microcontroller. The CD also includes:

• A schematic for the thermostat project

• Test programs in source code form

• Complete source for the thermostat control software

• Libraries to support the test and control software

• Pictures of a finished thermostat system

• Supplementary documentation

System Rquirements

The software runs on Microsoft Windows 95, 98, and NT.

To Install the Contents of the CD-ROM

1. Place of CD-ROM in your drive, and choose Start|Run . . .

2. Enter `D:\setup.exe`, replacing "`D:`" for the drive letter of your CD-ROM drive.

3. Follow the instructions given by the installer.