

Appendix A

C Language

■ ■ Historically, micro-controllers had small memories and exotic instruction sets (e.g. the Intel MCS-51 with its Harvard architecture and 16 ALU instructions [2]). Although many could be programmed using languages such as C, they required highly specialized compilers, and very often still required highly skilled assembly language programmers to optimize programs to fit into their limited resources. Since the early 2000's, micro-controllers began rapidly evolving in architecture and memory capacities. Vendors started moving away from the special purpose instruction sets to general purpose instruction sets with extensions. For example, Texas Instruments's new micro-controller line, the Tiva C, starts with an ARM-9 instruction set, a RISC architecture developed original for desktop computers and servers, increases CPU clock frequency, and adds relatively large memories (up to 512 kB of RAM and flash on-chip). Modern micro-controllers take allow for a wider range of programming languages, where the inefficiencies of compiler translation are less significant, which can lead to reduced development costs as a result of using a standard language and libraries.

■TODO: add function prototypes
■TODO: add pragma

The *de facto* standard programming language for micro-controllers is still the venerable C language. It offers a rich set of language syntax, compilers that generate reasonably efficient code, and source code portability. C++, the object-oriented language that was originally based on C, is starting to become popular. In the hobbyist community, compiled BASIC variants are also popular as they aren't difficult to learn, and the novice user isn't typically pushing the boundaries of what the chip can do.

One thing that makes programming micro-controllers different than other types of programming is that there is typically not an operating system. The program runs on the machine without any of the abstraction services provided by an operating system. There is no OS providing device drivers, task schedulers, protection, or security. When the one program crashes, the system is crashed, there is no OS to recover the error.

Instead of an OS, most vendors provide an Application Programming Interface (API) that can help fill-in for the missing operating system. Most vendors provide a basic implementation of the C library, including string functions (e.g. `strcpy()`), memory management (e.g. `malloc()`), and sometimes even basic I/O functions (e.g. `printf()`). In addition, there are usually API routines for configuring the various peripherals built into the chip.

These APIs can be invaluable to a developer, but there may be costs to their use. For example, using `malloc()` can help programmers who are used having multiple giga-bytes of RAM, but on a micro-controller with a few kilo-bytes, there may not be enough space to allocate a heap to hold the dynamic memory. In fact, one of the first signs that a micro-controller project is jeopardy of failure is when the program is written as if it were running on a full OS based system and is ignorant of the limited resources available on the micro-controller.

This chapter provides an overview of basic programming techniques that are useful for micro-controller programming. The reader should already have experience with C programming, typical of that which would be taught in an introductory CS1 course. The reader should also have some experience with computer organization concepts, especially the machine fetch-decode-execute cycle, pipelining, and basic assembly language programming consistent with the Hennessy and Patterson text [5]. The Assembly language used in this text is based on the MIPS R4000 introduced in that text.

Most introductory C courses provide instruction on how to program in C, covering basic language constructs, with an emphasis on making a program work. This is sufficient for an introduction, but to become adept at programming for micro-controllers (and a wide array of other environments), there are many details provided by the

```

#include <X11/keysym.h>
#define F(x,y,w,h,c) G(c),XFillRectangle(A,B,D,x,y,w,h),G(3)
#define G(c) XSetForeground(A,D,C[c].pixel)
#define H(d) addch(y?y-e?ACS_VLINE:ACS_L##d:ACS_U##d)
#define I(i,c) init_pair(i,COLOR_##c,COLOR_##c)
#define J(a,b) j==X[(int[]){ KEY_##b,XK_##a} ]
#define K(q,r,s,t,u,v,w) (q=!!(w)*z[p v]>>8,(1<<(3*!!(((r)>>t)^(s))&u)))?!q)
#define L z[p=rand()&63]

int c[9],e,f,g, h,i,j,k,l,m,n,q ,r,s,t,u,v,w,x,
y,X, z[64]; GC D; Display * A; Window B; char
o[]="tdpsf\73!" "e!&e!&e\1",*Y =o,b[64];XColor
C[8]; XEvent E; int R ( ) { XDrawString(A,B
,D,348,58, b, e ); XNextEvent (
XLookupKeysym(& E.xkey,0)*! (E. type^KeyPress);
} void P(){ g = DefaultScreen(( A=XOpenDisplay(
0 ) ) ) ; do XAllocNamedColor
( A , DefaultColormap(
A,g),5*x+"#fff" "\0#f00\0#0f0\0"
"#000",C+x,C+4) ; while(++x^4) ;
XMapWindow(A,B= XCreateSimpleWindow
(A,RootWindow(A ,g),0,0,460,300
,0, C[3].pixel, C->pixel ) ) ;
XSelectInput(A, B,KeyPressMask| ExposureMask);D
=XCreateGC(A,B, 0,0); } void a(
e; y++)if(X)for (x=k; y%3==0&&y ) { for(y=0; y<=
)XDrawRectangle (A,B,D,3+6*x,6+ ^e&&x<k+t; x+=6
else{ move(y,k) ; H(LCORNER ) ; 12 *y, 36, 36);
}addch(x%6?y%3? for(x=0; ++x^t;
: ACS_HLINE) ; 32:ACS_HLINE: y &&y-e?ACS_VLINE
initscr ( ) ; start_color( ); void Q ( ) {
keypad(stdscr,1 ); curs_set(0);
I(1, RED); I(2, GREEN ) ; I(3 ,
WHITE); } void d(){ for(k=!x ;
... ..

```

Figure A.1: A (winning?) submission to the International Obfuscated C Code Competition and an example of what C can do.[6].

language that are not typically covered. This section provides an overview of these topics. Throughout later sections of the book, these concepts will be referenced and used to develop more complex examples.

A.1 Data Representation

This section describes ways of representation data, both at a machine level as well as at the programming language level. The section includes topics such as Endianness, Literals, Special Data Types, and Type Qualifiers.

A.1.1 Endianness

Processors may be either little- or big- endian *byte* order, referring to how the *bytes* of a value are stored in memory. Little endian architectures store the least-significant *byte* in the first *byte* of memory; while big-endian architectures store the most-significant *byte* in the first *byte* of memory. The *bits* are not reversed. Suppose we store the following 32-bit value, expressed in hexadecimal, 01234567_{16} , in memory *starting* at address, also in hexadecimal, 1000_{16} . Then the value would require 4-bytes, occupying addresses $1000 \dots 1003_{16}$.

endian	1000	1001	1002	1003
big	01	23	45	67
little	67	45	23	01

The endianness of a processor is set by the Instruction Set Architecture (ISA). Low-level tools, such as the Assembler is responsible for performing the proper translation. The Assembler translates assembly code, written in text, into object code, written in binary. The Assembler is responsible for representing values, such as literals and machine instructions, encoded as text in the program, in the proper bit ordering. The C compiler translates its literal values into either immediate values encoded into instructions, or as native values stored in the data-section of a program [5]. Either way, the C compiler translates a C program into an equivalent Assembly code, and then invokes the assembler to convert the program into binary for the target architecture.

As a result, the endianness of a processor can be largely ignored by the C programmer; except when the value is used in a non-native manner. The two most common exceptions are interacting with I/O devices and when examining the data through a debugger.

Listing A.1: Endian conversion routines in C

```
1 // if the system has byteswap.h
2 #include <byteswap.h>
3
4 int get_value( )
5 {
6     int v = read_value( );
7     __bswap_32(v);
8 }
9
10 // if the system does not have byteswap.h
11 int swap(int v)
12 {
13     return
14         ( v >> 24 ) ||
15         ((v >> 8) & 0xff) ||
16         ((v & 0xff00) << 8)
17         ((v & 0xff) << 24);
18 }
```

Debugging tools connect to special ports in the micro-controller and allow control and inspection of the state of the processor and the memory. If we look at the starting address for a variable, we will have to interpret the byte values back into a word value, which will depend on the endianness of the machine.

Another area where endianness can be an issue is when performing low-level input/output between two processors of different endianness. Typically if data is being written to some device in raw binary format, the data may need to be rearranged to be read on another processor. For example the USB standard asserts that all multi-byte transfers are in little-endian order (matching Intel), while the TCP/IP networking standard asserts that all multi-byte transfers are in big-endian order. Endianness issues only apply to multibyte values - for example 16- and 32-bit integer.

Byte values, such as ASCII strings are always in string-byte order, such that the left most character (the first character in the string) is stored at the lowest memory address of the array, and then each character is stored, consecutively, in each of the next bytes with memory addresses increasing by 1.

For example, suppose a big-endian micro-controller collects 32-bit integers and stores the values directly onto a storage device such as an SD card. The SD card is taken to an Intel PC which is little-endian. Either the PC or the micro-controller needs to rearrange the multi-word bytes to be in the proper order. An example of C code that accomplishes this is shown in Listing A.1.

Typically, any communications standard will state the endianness of the data. Any custom format should choose a single standard and adhere to it. The biggest factor in the selection is likely going to be the the device that has the least capability. For

example, in the previous example of the big-endian micro-controller and the little-endian PC, which would be most efficient? The micro-controller is the least capable of the two processors. Selecting a little-endian format will make it easier for the PC but will require adding code to translate the data on the micro-controller will add an additional burden on a device, which may already be near its limit. Selecting a big-endian format will allow the micro-controller to simply write raw values to the storage device, but will require that the PC program translate the values when they are being read or written. It is easier to do this on a PC than it is on a micro-controller.

A.1.2 Literals

Literal values are values that are given in the source code of a program. As a convenience to the programmer, there are different formats to express different numerical bases and/or data types. For a number of reasons, explored throughout this chapter, micro-controller programming makes extensive use of values expressed in different bases, which will be explored in more detail in later subsections. However, Table A.1 presents a clear reason. Recall that a signed two's complement signed-integer uses the *most-significant-bit* as a sign-bit. Therefore, the greatest signed-integer is one with the most-significant-bit equal to 0, and all of the other bits equal to 1. In base 10, these values are rather inconvenient to remember, but in hexadecimal, they all follow a clear pattern.


Table A.1: Maximum Integers

Size	Base 10	Base 16
8-bit	127	0x7f
16-bit	32,767	0x7fff
32-bit	2,147,483,647	0x7fffffff

Integral Types The integral data-types can be values in different bases (radices). The C-compiler needs to disambiguate the radix in some fashion. For example, the literal '10' could be either: 10_{10} , 2_2 , 8_8 , and 16_{16} . This gives rise to a saying (and some cool T-shirts: "There are only 10 types of people - those who know binary, and those

who don't." The C language way of denoting the radix is to use a different prefix in front of the number as shown in Table A.2. One note of caution, not all compilers support the `0b` prefix for binary numbers, this is a recent extension, so it may or may not be available. **Be warned**, encoding binary strings is an error-prone proposition - it is considered better form to encode them as octal or hexadecimal values instead.

Table A.2: Radix Specifiers

Base	Prefix	Example
Binary (base 2)		<code>0b10110001</code>
Octal (base 8)	<code>0</code>	<code>0261</code>
Decimal (base 10)	none	<code>177</code>
Hexadecimal (base 16)	<code>0x</code>	<code>0xB1</code>

One trick when working with different radix values is to remember that three bits corresponds to one octal digit (base $8 = 2^3$) and four bits corresponds to 1 hexadecimal (hex for short) digit (base $16 = 2^4$).

Table A.3: Digits in the different bases

Decimal	Binary	Octal	Hexadecimal	Decimal	Binary	Octal	Hexadecimal
0	0b0000	00	0x0	8	0b1000	10	0x8
1	0b0001	01	0x1	9	0b1001	11	0x9
2	0b0010	02	0x2	10	0b1010	12	0xA
3	0b0011	03	0x3	11	0b1011	13	0xB
4	0b0100	04	0x4	12	0b1100	14	0xC
5	0b0101	05	0x5	13	0b1101	15	0xD
6	0b0110	06	0x6	14	0b1110	16	0xE
7	0b0111	07	0x7	15	0b1111	17	0xF

Signedness C language uses two different types of integers: signed and unsigned. An unsigned integer uses all of its bits to store the magnitude of the value. For example, an unsigned 16-bit number can store all integer values from $0 \dots 65535$, but it cannot store negative numbers. To store negative numbers, we need to use a *signed* integer. Signed integers are stored in two's complement, a technique that allows us



Example A.1.1. Convert `0xcafe` to binary, then use the binary to convert to octal. Each hex digit corresponds to 4 bits: `C = 0b1100`, `A = 0b1010`, `F = 0b1111`, and `E = 0b1110`. Concatenate the strings together to get: `0b110010101111110`.

To convert to octal, rewrite the binary grouped into threes: `0b1 100 101 011 111 110`. Take each group: `1 = 1`, `100 = 4`, `101 = 5`, `011 = 3`, `111 = 7`, `110 = 6`, and concatenate: `0145376`.



to perform arithmetic on signed or un-signed numbers using the same hardware. To determine the two's complement of a negative number, we perform the following operation:

1. Determine the binary representation of the positive value for the negative number
2. Using Boolean negation, negate each bit of the number
3. Using binary arithmetic, add one to the number.

In this representation, the most-significant-bit (MSB) is the sign bit. When it is set, the number shall be interpreted as negative, and when it is clear the number shall be interpreted as positive. In signed notation, a 16-bit number can represent values from $-32,768 \dots 32,767$. Note, the range of values has been halved, but there are still 65,536 different integer values.

Example A.1.2. Determine the two's complement representation of -33.

1. The binary representation of 33 is: `0010_0001`.



2. Negate: `1101_1110`

3. Add one: `1101_1111`

Thus, -33 is represented as: `0b1101_1111` in binary, `0xDF` in hexadecimal, and 223 in positive numbers.



To add two two's complement numbers, we simply add their binary values.

Mixing signed and unsigned operations can be a source of bugs in C language programs. C language interprets a right-shift of a signed number differently than a

Example A.1.3. Calculate $-33 + 50$.




We have $0b1101_1111 + 0b0011_0010 = 0b1_0001_0001$, in 8-bits, the extra carry bit is dropped, so we have $0b0001_0001$, in hexadecimal we have $0x11$, which is also $1 * 16^1 + 1 * 16^0 = 16 + 1 = 17$, which is exactly what we expect from $-33 + 50$.



right-shift of an unsigned number. The unsigned right-shift always fills values on the left with 0's. A signed right-shift always fills in values on the left with the sign-bit.



Example A.1.4. A micro-controller application reads an 8-bit value from a serial shift-register. The value includes a 7-bit ASCII code in the upper 7 bits, followed by an even parity bit in the least-significant bit. For example, the capital letter C, $0x43$, would be stored as $0b1000_0111$ (7-bits of  and the parity is set to make an even number of bits).

```
char ch = read_character();
char wrong = ch >> 1;
unsigned char right = (unsigned char) ch >> 1;
```

the first right-shift will be performed using C's default signed character. The most-significant bit is set, and sign-extension be used to fill in on the left, resulting in the value 1100_0011 ($0xC3$), not $0x43$. In the second example, the right-shift is type-cast to be unsigned before the shift, and a 0 is correctly shifted into the left positions.



Literals are mapped onto a data-type based on their length and radix. If the length of a decimal literal is less than or equal to the length of a native word, it will be mapped into an native `int` type. If it is longer, it will be mapped to a `long long` type. If it is given in a binary, octal, or hexadecimal, then it will be an `unsigned int`. The compiler will use a wider type to match the length of the literal, following the signed for decimal, unsigned for non-decimal literals.

Listing A.2 shows the problem with the compiler's built-in typing. The C compiler

Listing A.2: A Literal Error

```
1 unsigned long long compute(unsigned shamt)
2 {
3     // wrong - 1 is going to be mapped to signed integer
4     // shamt amount >= 32 will be zero!
5     return 1 << shamt;
6 }
```

will generate code that does the following (in order):

1. Starting at line 5, the `return` statement is evaluated.
2. The right-hand-side (RHS) of the return has precedence.
3. The expression `1 << shamt` is evaluated.
4. The left-hand-side (LHS) of the shift instruction has precedence.
5. The LHS of the shift is 1 which is a *decimal* literal that fits in a native int, so it is evaluated as `(int) 1`.
6. The RHS of the shift is `shamt`, which is translated to a `LOAD` instruction to load the value from memory.
7. The LHS of the shift instruction is a native integer, so it is translated to a `SHIFT LEFT` native instruction.
8. The result of the shift instruction, a native signed-integer, is used as the result of the RHS of the return statement.
9. The return-type of the return statement is an `unsigned long long`, so the compiler *type-casts* the value to match
10. The function's epilogue is executed and the value is returned.

So long as the shift amount is less than 32, the code will work, but anything greater will result in a zero, even though the return type is wide enough to hold a greater shift value. This is actually a logical consequence of the code that was given to the compiler. The problem is that the compiler selected a shift amount based on the LHS of the shift instruction, and that literal was a native int. To fix this, we could either *type-cast* the value and rewrite the express as `return ((unsigned long long) 1) << shamt`; or else append a *literal format modifier*, which does exactly the same thing as type-casting, but makes the code more concise. Table A.4 shows the modifiers and their meaning.

Table A.4: Literal Format Modifier

Modifier Suffix	Meaning
u or U	Make value unsigned, does not change width
l or L	Make value long int, does not change signed-ness
ul or UL	Make the value unsigned and long (may still be 32-bit!)
ll or LL	make the value long long, does not change signed-ness
ull or ULL	make the value unsigned long long

Example A.1.5. The code in Listing A.2 suffered from *implicit* length determination by the compiler. To force the literal to match the return-type, and force a *wide* shift operation, we can rewrite line 5 as: `return 1ULL << shamt;`, which will force the 1 bit into a double-long integer (typically 64-bits on a 32-bit machine).

■

Floating-Point Types All floating-point constants are presumed to be 64-bit `double` by the compiler, and can be forced to be `float` type by using a suffix of `f`. For example, `3.14f`. Similarly, in systems that support 128-bit `long double` values a suffix of `l` or `L` can be used for greater precision and magnitude literals. Micro-controllers do not typically have floating-point units (FPU), so any floating-point operations are going to be *software floating-point*, where the compiler turns a floating-point operation into a series of integer operations at a significant penalty in performance. Here, there will be a significant difference in performance between single- and double-precision floating-point, so selecting the proper constants is important.

Character Types All character constants (single characters inside single quotes) are presumed to have an `int` data type. The compiler will convert these to be 8-bit only during optimization. In this way, characters can be used as values in arithmetic expressions.

String Types String literals get special treatment from the C compiler. Whenever a string literal is encountered, the zero-terminated string of ASCII characters will be stored in the *read-only* data section of a program. A *pointer* to the start of that string will be used as its value. For example, we can write the code: `char *msg = "hello world";`. Here the `msg` variable is declared to be a pointer to a character (a 32-bit

Listing A.3: String Format Error

```
char *fmt = "message %d %f\n";  
2  
void oops(int a, float b)  
4  
{  
    fmt[9] = 'f';  
6    fmt[12] = 'd';  
8    printf("message %d %f\n", a, b);  
}
```

value on a 32-bit machine), and the compiler assigns the *address* of the start of the string into the pointer. We now say that *msg* points to the string *hello world*.

To save space, the C compiler will only store one copy of every unique string. Any time that string is encountered, the same address will be used in its place. This will be explored further in Listing A.3.

If we change what `msg` points at, there is no problem, no traps, no “gotchas.” However, if we change the string that `msg` is point at, that is a problem. Even though the string is stored in a read-only section, it is still stored in memory that is mutable. In fact, changing the string is allowed, but it will give a warning at compile time. **Pay attention to these warnings!**

Listing A.3 shows truly horrible code. The compiler will store one copy of the string into the read-only data section of the code. The compiler sees that the string is used in the assignment to `char *msg`, and its also used in the function call `printf`. Both of these uses will get the same starting address of the string. However, this tricky code uses the `msg` string to modify the string. This will assuredly confuse the `printf` function which expected the original string.

Translation of Literals

The Instruction Set Architecture (ISA) will determine how C language literals can be put into the assembly language program. The mechanism will be determined by the size and type of the literal, and how its used. Options include: *immediate* values or constants.

Literals that fit into the immediate field of an instruction can be encoded in a single instruction. For example, the code `a = a + 1` could be translated to the MIPS instruction `ADDI $t0, $t0, 1`. Literals that are larger than the field of an instruction will often be encoded as a series of machine instructions to load the value into a regis-

ter. For example, `a = a + 0x7fffffff` might be encoded as `LUI $at, 0x7ffff;`
`ORI $at, 0xffff;` `ADD $t0, $t0, $at`. Literals that fit this pattern do not take up
memory space, *per se*, but are stored directly in the instruction stream.

For values such as: strings, structs, and other complex data types; the compiler
will store them in the *data section* of the program, and use them through *indirection*,
where the compiler tracks the memory address of where the literal is stored in mem-
ory. When the value is used, the compiler issues instructions to load the memory
address into a register, load the value into a register and execute it. So, we can expect
to see a `LUI`, `ORI`, followed by a `LB`, `LW`, or even an `L.D.`.

The `const` keyword

The best-practice way of working with string literals is to always use the `const` mod-
ifier on a variable. This advises the C compiler that the value is read-only, and any
attempts to modify its value should be prevented. The `const` qualifier modifies the
type directly to its right. For example, in the previous example, using `const char`
`*msg` would ask the C compiler to make the variable `msg` a pointer to `const char`.
This means that code like Listing A.3 would generate an error. An observant reader
should detect that bets are being hedged, hands are being waved, and the other shoe
is about to drop. Type-casting can over-ride the `const` modifier, and the values can
be changed anyway, which is still dangerous.

It is considered a best-practice to *always* use the `const` keyword whenever a
pointer is being passed to a function and that function is not going to change the
value. Listing A.4 shows an implementation of `strcpy` that makes use of the `const`
keyword. This informs the programmer and the compiler that the function will *not*
try to modify the *source* for the copy, and the lack of the `const` keyword will us that
the *destination* may change. Not only does this improve the compiler's ability to de-
tect errors (e.g. unintentionally changing a variable in a function), but it can also help
the compiler optimize code. By assuming that the `const` values won't be changed, the
compiler can assume that the value will be preserved across function calls, which will
save a store-word to save the value before the function call and a load-word to restore
its value after the function call.

Listing A.4: Proper Use of const Keyword

```

1 // note the use of the const modifier
void strcpy(char *dst, const char *src)
3 {
    while (*src != 0) {
5         *dst++ = *src++;
    }
7     *dst = 0;
}

```

If the code in Listing A.4 omitted the `const` keyword, then calling the function with a string literal would generate an error. We *could* type-cast the string, but if the function really does change the string, then we'd be writing truly horrible code. The best-practice way of fixing this is to fix the function, and add the `const` keyword as appropriate. If the function really does change the string, then we will need to *copy the string* to a new, writable, not-shared string; then call the function on the copy of the string. This is what languages like Java do to protect their strings, and in fact, this is why languages like Java do it.

The `const` modifier can be applied to other data-types as well, and will be discussed further in Section A.1.5.

A.1.3 Special Data-Types

Micro-controllers are notorious for using different integer sizes. Recall that the ALU of a processor can perform arithmetic on the native word size. Most ALUs can perform an operation on a smaller size operation at no penalty. In all cases, wider operations will require the compiler to re-write the wider operation in terms of native operations. For example, 64-bit addition on a 32-bit ALU requires three additions to handle lower 32, upper 32, and carry bits.

Table A.5: Integer Sizes

Micro-Controller Family	ISA	Year	# bits for int
PIC16	PIC	1993	16
MSP430	MSP430	1995	16
PIC18	PIC	2000	16
AVR32	AVR32	2003	32
PIC32	MIPS4K	2007	32
Tiva C	ARM9	2013	32

Consider Table A.5, which shows a trend among micro-controllers. As time passes, micro-controllers are trending towards 32-bit ALUs, and more standard instruction sets. The wider instruction are a double-edged sword. The wider instruction set means that addresses are also wider, and often instructions are wider too. These reduce the *code density*. Code density is the number of instructions per byte. Reduced code density can be a significant performance issue for modern micro-controllers. Because they have smaller memories (both working memories and storage memories), reduced code density means that a program will require more bytes to store, to load, and to execute. Micro-controllers also have smaller caches (if any at all), which means that both the instruction and data caches are likely to have lower hit rates as a result of fewer entries per cache storage.

Standard Integer Types

In older C implementations, it was common to use data-types like `int x;` and `long int y;`. These are hold-overs from the early days of C programming. In fact, many programmers are surprised that both the `int` and `long` datatypes are the same size on 32-bit machines! Of course, in legacy 16-bit systems, they are different. Then programmers started using `long long z;` which should be 64-bit, but not always. Table A.6 shows the built-in integer names, their widths, and their ranges.

Table A.6: Built-In Integer Types for PIC32

Definition	Size (bits)	Min	Max
<code>char</code> , signed <code>char</code>	8	-128	127
unsigned <code>char</code>	8	0	255
<code>short</code> , signed <code>short</code>	16	-32768	32767
unsigned <code>short</code>	16	0	65535
<code>int</code> , signed <code>int</code> , <code>long</code>	32	-2^{31}	$2^{31} - 1$
unsigned <code>int</code> , unsigned <code>long</code>	32	0	$2^{32} - 1$
<code>long long</code>	64	-2^{63}	$2^{63} - 1$
unsigned <code>long long</code>	64	0	$2^{64} - 1$

To fix these problems, the C99 standard contains a requirement that the compiler include a header called `stdint.h` which defines a number of standard, fixed-width integers. The standard guarantees that there are standard definitions for 8-, 16-,

Listing A.5: Saturating Addition - will not overflow, rather it will saturate onto maximum value that can be stored in integer.

```

1 #include <stdint.h>
2
3 uint16_t sadd(uint16_t A, uint16_t B)
4 {
5     uint32_t result = (uint32_t) A + (uint32_t) B;
6     if (result > INT16_MAX) return INT16_MAX;
7     else return (uint16_t) result;
8 }

```

32-, and 64-bit integers. Systems that have more exotic ALUs may include other data-types. For example, in Texas Instruments's DSP architecture, it is common to use 40-bit integers, which would give us `uint40_t`; but this is not defined on all systems.

The standard also defines constants named `INTn_MIN`, `INTn_MAX`, and `UINTn_MAX`, which hold the signed minimum, signed maximum, and unsigned maximum value for each datatype. The standard further defines two additional integer formats: `int_leastn_t` and `int_fastn_t`, where $n = 8, 16, 32, 64$. The least indicates the smallest width that the implementation supports (usually 8-bit), and the fastest indicates the fast integer implementation (usually the native word length).

Table A.7: Standard Integers Defined by `stdint.h`

Definition	Size	Signed Range	Unsigned Range
<code>int8_t</code> / <code>uint8_t</code>	8 bit	$-128 \dots 127$	$0 \dots 255$
<code>int16_t</code> / <code>uint16_t</code>	16 bit	$-32,768 \dots 32,767$	$0 \dots 65,535$
<code>int32_t</code> / <code>uint32_t</code>	32 bit	$-2,147,483,648 \dots 2,147,483,647$	$0 \dots 4,294,967,295$
<code>int64_t</code>	64 bit	$-9,223,372,036,854,775,808 \dots 9,223,372,036,854,775,807$	
<code>uint64_t</code>	64-bit	$0 \dots 18,446,744,073,709,551,615$	

An example that uses the standard integers is shown in Listing A.5. This code fixes a common problem in micro-controller programming - often we want to use smaller data-types for efficiency, but we don't want to overflow to smaller / negative numbers when the values get too large, instead, we want to *saturate* onto some maximum value. This code uses the standard definitions to prevent overflow from an ALU operation.

Floating-Point Types

Floating points types are included in some, but not all, micro-controller architectures; and support is included in some, but not all, micro-controller compilers. Most

micro-controller processors do not include hardware Floating Point Units (FPU), the floating-point version of the Arithmetic Logic Unit (ALU). FPUs tend to consume quite a bit of chip area, draw considerable amounts of power, and most micro-controller applications do not make extensive use of floating-point operations. C compilers do frequently, but not universally, include support for soft floating point instructions. The C compiler will convert floating point operations into a series of integer operations. This allows a program to use floating point when it is needed, but there are a number of drawbacks to this approach.

Consider the classic MIPS 4000 architecture as described in Hennessy & Patterson [5]. The MIPS processor did not directly support floating point. Instead, the FPU was an optional co-processor. This provided for an entirely different set of instructions (e.g. `LWC1`, and `ADD.S`) and registers (e.g. `F0`). So, without these extra registers and hardware, the C compiler will schedule a series of integer instructions (pushing integer values out of registers) in order to compute the equivalent floating point value. Soft floating-point will be several times slower than having a hardware unit, but the increased register spilling will also penalize surrounding integer code.

The PIC32 C compiler provides support for three floating point types: `float` (32-bit), `double` (also 32-bit), and `long double` (64-bit). From a C language perspective, the key difference is that the `double` type does not give the familiar 64-bit value. Otherwise, floating point types can be used like they would on any other C language instance.

A.1.4 Local, Global, and Dynamic Variables

In C language, there are three types of variable lifetimes: local, global, and dynamic. Any variable type can have one of these lifetimes. The biggest difference between them is where the memory is allocated from, and how and when they are destroyed.

A local, or *automatic* variable is one whose lifetime is managed by the running code. In general, any variable that is declared inside the curly brackets has *lifetime* as long as the end of the brackets has not been reached. The most common place that a local variable is encountered is at the beginning of a function (which has curly brackets). Any local variables have lifetime as long as the function does (the matching

Listing A.6: Local lifetime with anonymous brackets

```
void local_lifetime()  
2 {  
   int sum;  
4  
   {  
6     int i;  
     for (i = 0; i < 10; i++)  
8       sum += i;  
   }  
10 printf("sum is: %d\n", sum);  
12 }
```

ending brackets have not yet been encountered). If a function call is made, then the local variables *go out of scope* for the duration of the function call, but they still have storage and lifetime. When the function call returns, the variables are again visible they are *back in scope*.

Functions are *named* blocks of code, but local variables can be declared in any *curly bracket* enclosed code. The block does not even need a name. Listing A.6 shows an example where a variable is declared in an *anonymous* block of code. The variable *i* is declared on line 6, and will have lifetime and scope until the block is finished on 9, and then its storage will be given up.

All local variables in C language are declared in the stack. Whenever the compiler determines that a new scope is required, the beginning brace is equivalent to growing the stack pointer to hold the local variables, and then ending brace will shrink the stack pointer to deallocate the local variables. Because the stack is preserved during function calls, the variables have lifetime, but the callee does have information about the caller's stack, they are not in scope. In this way, the normal mechanics used by the compiler to manage the stack enable the compiler to manage local variables' storage automatically.

Global variables are declared outside of any named block. They are called global because they have scope and lifetime of the entire program. Global variables are stored in a special memory segment that is either initialized or uninitialized, depending on how they are declared. Initialized variables have their values stored in the program's code; uninitialized variables are simply allocated when the program is loaded. Any code that has the symbol's declaration visible can use the variable. Since only one line of code can declare a variable, C language uses the *extern* keyword to assert that somewhere else in the program is exactly one declaration for that variable.

Listing A.7: Dynamic example with custom lifetime

```
void process_payment(payment_t *payment)
2 {
    if (!enough_funds(payment))
        write_error(payment);
    else {
        print_check(payment);
        write_log(payment);
    }
    free(payment);
10 }

void dynamic_example( )
12 {
14     payment_t *new_payment = (payment_t *) malloc(sizeof(payment_t));
16     if (new_payment == NULL) abort();
18     process_payment( new_payment );
20 }
```

For example, in a C file, there could be a global variable: `int count_keystrokes = 0;`. In a shared header file, there would be the matching extern declaration: `extern int count_keystrokes;`. The C code actually allocates the space for the variable, the extern declarations are just promises that the variable will actually be declared somewhere else.

Global variables can be used to make some tasks easier, especially for critical information to the program's operation. For example, a structure that holds the program's active configuration might be a good candidate for being a global variable. The main reasons that discourage their use is that there really is only one, and code that is running in a multi-threaded environment cannot have more than one. Code that is built to work with the global variable cannot also work with a value that is passed into it.

Finally, *dynamic variables* are allocated on the program's heap. These are variables that are created through calls to `malloc`. The programmer is responsible for allocating and deallocating these variables at run-time. The call to `malloc` gives back a pointer to the block of memory that was created by the `malloc` call. Typically, this block of memory is *typecast* and assigned to the desired pointer variable with the desired type.

Listing A.7 shows a classic dynamic example. The function `dynamic_example` uses `malloc` to allocate a chunk of memory, typecast and assign that chunk, and then ensure that the pointer is valid. The block is sent to another function `process_payment` which works with it, and when it is finished, it destroys the variable with a call to `free`. In this example, the lifetime of the block of memory allocated with `malloc` is

managed by the program explicitly; there is no matching pattern as either a local or global lifetime.

A.1.5 Type Qualifiers

The three most common type qualifiers are `const` (which was introduced in Section A.1.2), `volatile`, and `static`. They are all critical for effective micro-controller programming. They provide important information about the way the programmer *intends* to use the variable. This in turn can aid in error-detection and in optimization.

Constant Qualifier

The `const` qualifier, as noted previously, makes a variable read-only after its initial assignment. The read-only behavior is only enforced through compiler statements, it is not enforced by hardware. For example, a type-cast can override the `const` behavior. The qualifier is important for two reasons: help with debugging and improved optimization.

First, it can help give sanity to a program. For example, if a function is defined to accept a parameter that is `const`, then if that function tries to modify that variable, it will be trapped as an error by the compiler. This also helps during debugging, when trying to locate changes to a value, if it is marked as `const`, we can largely ignore that function, and focus our attention elsewhere.

Second, marking values with `const` helps the compiler optimize the code. When a value is passed into a function, that value may be loaded into a register. If the value is `const`, then under certain circumstances, the compiler can assume that the value will be in the register after the function call, and it will not need to re-load the value from memory. This can save an extra load/store to spill an old value and reload the pointer value.

The `const` keyword may seem straightforward, but there are some traps that can really confuse programmers. The qualifier *only* modifies the value immediately to its right.

A pointer to a `const` means that the pointer can change, but the contents cannot. For example, `const char *ptr` is a *pointer to a const char*. We can change the start-

ing address that `ptr` points to, but cannot change the contents in the block of memory. We can write code like `ptr = "hello"`, but we cannot write `ptr[0] = 'a'`; The typical form is `const char *ptr`.

const pointer A const pointer means that the pointer cannot change, but the contents can change. So, we cannot change the block of memory that a value points at, but we can change its contents. For example, `char const *ptr = "hello";`, we can write `ptr[0] = 'c'`; (and now the text is “cello”), but `ptr` cannot be made to point to any other value. This is handy when using `malloc` and other routines that return a pointer that should never be lost. The typical form is `char * const ptr`.

const pointer to a const A restrictive combination of the two, neither the pointer nor its contents can change. For example, `const char * const ptr = "hello";` This pointer can not refer to anything else other than its initial value and the value cannot change (using `ptr`). The typical form is `const char * const ptr`.

Memory Organization The `const` qualifier can also impact where in memory a variable is placed. First, most micro-controllers have limited RAM (e.g. many have 16K or 32K), but can have larger program memories (e.g. 512K). Because variables that are declared as `const` are given a literal initialization value, they are known at compile time. As a result, variables that are declared as `const` can be placed in the larger data section (by the compiler). This can be of vital importance for certain common micro-controller applications.

Example A.1.6. One common application of micro-controllers is to use them for PID controls, where there is an array of a constant values stored in memory. Suppose a PID controller needed 1024 32-bit integers, which would require 4K of space. Declaring the array `array` as just an array of `int` with an initializer would cause the array to be stored in RAM, and would cause it to use 1/8th of the total available memory, while declaring the array with the `const` modifier would allow the compiler to optimize the layout and use the copy of the data stored in the program's memory image, leaving the limited RAM used for other purposes.



Listing A.8: Infinite loop due to register optimization.

```

1 void wait_for_count( )
2 {
3     uint32_t *count = 0x10000000;
4
5     *count = 0;
6     while (*count < 1000) ;
7 }

```

Volatile Qualifier

One of the most important optimizations done by the C compiler is to minimize the loads / stores of variables in memory. Instead, the compiler will aggressively schedule code to keep values in CPU registers. In this way, the compiler is treating the registers like a *cache*, assuming that the only way the value will change is through instructions executed by the processor. Therefore, we can assume that if a value has been read from memory into a register, its value will stay the same (in both memory and in the register). In normal applications programming, this is a safe assumption. When working with hardware, this is frequently not a safe assumption. The `volatile` keyword disables this optimization.

Hardware typically makes use of *memory-mapped I/O*, where a hardware device is mapped into the memory space used by the processor. Performing loads and stores to this address causes the hardware device rather than the memory to handle the request. If compiler's memory optimization were used, then the first time the value is read, it will be copied into a register and the device won't be read again.

Listing A.9: Assembly code due to register optimization

```

1 wait_for_count:
2     xor    $a0,$a0        % a0 used for count
3     lui    $a0,0x1000      % after this, a0 holds address 0x1000_0000
4     xor    $t0,$t0,$t0      % set t0 to be 0
5     sw     $t0,0($a0)       % write the 0 to address 0x1000_0000
6     slti   $t1,$t0,1000     % set t1 = 1 if t0 is < 1000
7     bnz    $t1,-8           % if t1 is not 0 jump back to which is slti
8     nop                    % nop in branch delay slot
9     jr     $ra              % otherwise, return

```

Consider the code shown in Listing A.8. Let us assume that there is hardware counter at memory address `0x10000000`, that counts some event such as clock ticks, button presses, etc. As this code is written, the C compiler will make use of the register optimization. Line 2 creates a pointer to an integer at a specific hardware address. Line 4 sets the count to be 0. Most C compilers will convert this code into a more optimal form, as shown in MIPS assembly Listing A.9. Note that the register, once written, isn't ever read back. As far as the compiler is concerned, why should it be

read again? Syntactic analysis of the C source code shows that the variable `count` is written two once, and then only read. There is no reason to assume that it should change. As a result, this code will run forever, even if the counter is changing. This can be a really confusing element in debugging - it is easy to assume that the counter is not incrementing.

This is a case for using the `volatile` keyword. It disables the register optimization. Changing line 3 of Listing A.8 to `volatile uint32_t *count = (volatile uint32_t *) 0x10000000;` will instruct the compiler to always re-read the value and not assume that the copy in the register reflects the current value. Note that the integer on the right-hand-side will also need to be type-cast to a volatile pointer. Most compilers need this type-cast to avoid compilation warnings.

Whenever hardware resources are being accessed the `volatile` keyword should be used. It is never correct to omit the `volatile` when setting up a hardware register. In fact, anytime there is a thread of execution that is reading a value, and a different thread of execution writing the value, the `volatile` qualifier should be used. Here, we can consider the counter to be running in its own thread of execution and sharing the memory bus with the CPU. This is also true for multi-threaded programming where multiple threads share the memory space but could be running on different processors.

Like the `const` qualifier, the `volatile` qualifier also applies only to the term on its immediate right. When applied to pointers, it is almost always the case that it should apply to the data type rather than the pointer.

pointer to `volatile` data The data that the pointer is pointing at is volatile and show always be read from memory. The typical form is `volatile int *ptr`.

`volatile` pointer to data The address for the pointer shall always be re-read, but the data can be register optimized. This is most likely a degenerate case or a mistake. There would be need to be some process that updates the pointer, but the data from the pointer will only be modified by the task after its been loaded. The typical form is `int * volatile ptr`.

`volatile` pointer to `volatile` data The register optimization is turned off for both

Listing A.10: Three examples of the `static` qualifier

```
1 // global static variable -  
2 // lifetime is entire program  
3 // scope is this compilation unit  
4 static int max_value = 0;  
5  
6  
7 // static function  
8 // lifetime for all functions is always global  
9 // scope is only this compilation unit  
10 // cannot call from any other unit  
11 static int getValue( )  
12 {  
13     return max_value;  
14 }  
15  
16 // normal function  
17 int getCount( )  
18 {  
19     // with static variable  
20     // scope only this function  
21     // lifetime is entire program!  
22     static int count = 0;  
23  
24     return count++;  
25 }
```

pointer and its data. The is also a degenerate case, and is probably a mistake.

The typical form is `volatile int * volatile ptr`.

Static Qualifier

The `static` qualifier is context sensitive, where its meaning depends entirely on how it is used. It can be applied to variables and functions, and generally aims to impact the scope and lifetime of an entity. The three ways of using using the `static` qualifier are shown in Listing A.10.

When `static` is applied to a *global variable*, it causes the global variable (which already has global lifetime) to have scope only in the compilation unit in which it is declared. For example, if both the code in Listing A.10 and another file declared a `static int max_value`, then there would be two independent “global” variables, one in each of the compilation units. Only the compilation unit could see its own copy. An example of this technique is shown on Line 4.

Lines 10-13 show a static function. The `static` qualifier limits the function to have scope of this and only this compilation unit. The function cannot be called from outside the compilation unit (“c file”). It does not make sense, and will probably be a warning or an error, to declare a static function in a header file.

This technique is really useful when developing code, such as Abstract Data Types, that have *helper functions* that are called to do internal processing and that should

never be called by any other part of the program. By marking these functions as static, the compiler will prevent these functions from being callable except for this compilation unit.

Lines 17-25 show the third way of using the static qualifier. In this example, there is a *local variable* that is declared static. A static, local variable will have a lifetime of the entire program, but will remain local to the function. The assignment statement of Line 22 will be executed once during program initialization and *not on each function call*! In this example, each time the `getCount` function is called, it will increment the `count` variable. Since the variable has global lifetime, the count will keep increasing each time the function is called (of course, it will reset to zero whenever the program is run).

The static qualifier is an important modifier for variables and functions. Its use allows some elegant solutions to problems involving persistence and controlling the visibility of functions and variables. For example, suppose there were a number of a functions that we wanted to track the count of function calls. Without the static modifier, the only way to accomplish this would be to include a number of global variables, which would be visible throughout the entire program. It would be easy to accidentally code a function to read or write the incorrect global variable, and would thereby introduce any number of bugs into the code.

The use of static local variables is far less than the use of the static qualifier on a function. This is an important tool for locking down code in a compilation unit and preventing it from being called, even accidentally, by other parts of a program. It is somewhat equivalent to marking a function as *private* in Object-Oriented Programming (OOP) languages like Java and C++.

Restrict Qualifier

This qualifier is used to communicate additional information to the C compiler to allow it to make better optimization decisions. The `restrict` qualifier can only be applied to pointers, and it tells the compiler that the pointer is the only way to access the memory that is being pointed at.

The C language compiler has an optimization phase that tries to make more effi-

cient code. By marking a variable with `restrict`, we are telling the compiler that it can safely assume that only by explicit changes to this pointer will the memory change. This may lead to reduced memory loads and stores and more efficiency.

A.2 Special Operators

This section introduces a number of special operations that are commonly used in C language. They highlight some of the powerful capabilities of C that make it an attractive language for systems development (and that also make for powerful traps for new programmers). They include using evaluated values as a side-effect, ternary operators, and compound statements.

A.2.1 Evaluations

As in most programming languages, each *statement* in C language is a line of code, terminated by a semi-colon. Statements can be gathered together into a block enclosed in braces. Named blocks are *functions* if they return a value or *procedures* if they have a void return type. Every statements have some type of *evaluation value* consisting of a *type* and if the type is non-void, a *value*. For example:

```
1 int x = 30;
```

Is *evaluated* using a left-to-right parsing strategy. The expression is determined to be an assignment statement which follows the *right-hand-side* (RHS) is evaluated, and assigned to the *left-hand-side* (LHS). In this example, the RHS has the value 30, which is assigned to the variable `x`. What many newcomers to the language don't know is that the entire statement evaluates to the *type* and *value* that was assigned.

For example, an assignment statement can be used in another expression. In the following example, the assignment is made, but because the assignment evaluates to the type of `x`, and the value that was assigned, the result can be used in other expressions, in this case, the `while` can compare the value that was assigned as part of its conditional.

```
1 while ( (x = 2 * x) < 100) do_something();
```

Using evaluations as part of a larger expression can be a powerful statement, but should also be used with care. The code of the following example illustrates what happens when an assignment statement is used (erroneously) in place of a comparison statement.

```
1 while (y = 2) y = get_next_value();
```

The while loop will evaluate the assignment state to be 2, which is non-zero. C uses non-zero values to indicate a logical true. The result is that the while loop will be an *infinite loop*, but most importantly, because assignment evaluates to a valid value, it won't be a compile-time error.

The evaluation result of a numerical operation will be the type and value of the operation. The evaluation result of a function call will be the function's return type and value. For C library functions, this is documented in the manual. For example, the familiar function `printf` is defined to return an `int` that represents the number of characters actually printed.

A.2.2 Ternary Operator

The *ternary operator* is a shorthand expression that is provided by a number of languages including C. The syntax for the operator is: `(expression) ? true_part : false_part;`. The reason it is called a ternary operator is that it has three parts. The expression is evaluated (as in the previous section), and if the expression evaluates to true (non-zero integer), then the `true_part` is executed, otherwise the `false_part` is executed. This works exactly like an `if...else...` statement.

```
1 a < b ? min = a : min = b;
```

As a general rule, the ternary operator should only be used for simple expressions where it really makes sense and improves code readability to include the whole statement on one-line. Any other circumstance should be written as the `if...else...`

A.2.3 Short-Circuited Statements

Short-circuiting is a technique where a number of statements are conjoined with logical operators (`&&` and `||`). Because C is a left-to-right language, the first expression on the left is *evaluated*, and if it is non-zero it is logical true, and if it is zero it is logical false. If the operator is logical AND (`&&`) and the left-hand-side expression is false,

then further evaluation is stopped (its shorted). If the operator is logical OR (`||`) and the LHS is false, then evaluation will stop.

```
1  int x = 10;
3  (x < 10) || printf("too small");
   (x == 10) && printf("OK");
```

In the previous example, for the first conditional, because `x` is not less than 10, the left-hand-side is false. The truth-table for OR shows that *false OR true* could still be true, the right-hand-side needs to be evaluated, and the `printf` will be called. However, if `x` were less than 10, then the LHS would already be true, and there is no reason to evaluate the RHS, and the `printf` would not be called. The same is true for the second statement. Because the LHS is true, the result could still be false if the RHS is false, so the RHS needs to be evaluated, and the `printf` will be called. However, if the LHS is false, AND will always be false, so then the RHS will never be called.

The use of short-circuiting to replace traditional control structures such as `if...else...` or even the ternary operator is questionable. For readability and maintainability, it should almost always be avoided. Although the technique works, it is an irregular coding pattern and a likely source for bugs.

One place where short-circuiting should be used is building compound statements for `if... statements`. When ordering the conditionals, a little thought about the *likelihood* of a condition can improve code. Because short-circuiting will terminate the evaluation early, the clauses should be organized so clauses that are most likely to terminate the evaluation are on the left.

For example:

```
if (likely_false && likely_true)
```

will be faster than:

```
1  if (likely_true && likely_false)
```

because in practice, the first one will only need to evaluate the first clause to terminate the evaluation. Remember that for short-circuiting to work, the compiler must schedule Assembly instructions that branch after each clause. So, each clause has the

overhead of a conditional branch at the assembly level. The goal is to minimize those branches. For example, the first example could be re-written as:

```
1  if (likely_false)
3      if (likely_true)
        do_something
```

Notice how the Boolean connectives were replaced with the nested if statements. In reality, these boolean connectives are just that - a shorthand for nested conditionals. Knowing that, another pattern that should be avoided is the use of common, repeated sub-expressions in conditionals. For example:

```
1  if ((a < b) && (c == d)) do_something();
3  else if ((a < b) && (c < d)) do_something_else();
    else do_yet_another_thing();
```

Notice that the `if` statement uses the same clause: `(a < b)`. Because the compiler has to schedule loading the values, comparing the values, and branching, this quickly adds up to a number of unnecessary CPU cycles. For efficiency, these redundant clauses should be eliminated by re-writing the statements using nesting to eliminate repeated evaluations and branches.

```
1  if (a < b)
3      if (c == d)
        do_something();
    else if (c < d)
5        do_something_else();
    else
7        do_yet_another_thing();
```

A.2.4 Compound Statement

One final special operator in C language is the *compound statement* operator `(,)`. The comma is used to separate multiple statements. The result of the evaluation of a compound statement is the last statement (in left-to-right order). Compound statements can be inserted anywhere where a single statement is expected.

For example:

```
1  for (i = 0, j = 3; i < 8 && j < 15; i++ , j+=2)
```

Uses the compound statement in both the loop initialization and the post-loop update expressions in the `for` loop. The proper way to interpret how this code should be read is to convert this `for` loop into an equivalent `while` loop:

```
1  i = 0;  
   j = 0;  
3  while (i < 8 && j < 15) {  
       // body  
5     i++;  
       j++;  
7  }
```

The compound statements are only used to keep track of the two different variables, they don't fundamentally alter the behavior of the loop itself (i.e. they don't turn a single loop into a nested `for` loop).

Aside from their use in loop control statements, the main use of compound statements includes variable declaration:

```
1  int x = 3, y = 2, z = 1;
```

Later, when the C Pre-Processor is introduced, compound statements can also be used when defining macros, where the macro defines a single statement, but that statement is composed of sub-statements, and the evaluation result is the last statement.

A.3 Bit-Wise Operations

C Language provides a number of bit-wise operations which perform some Boolean operation on each of the bits of the given values. The prevalence of bit-wise operations is another difference between general applications and micro-controller programming - micro-controller programs tend to use bit-wise operators very frequently. These operators are arithmetic instructions that are defined for all integral data-types. This section provides a brief overview of their use as a preamble for later specific patterns that are used extensively in micro-controller programming.

A	B	AND	A	B	OR	A	B	XOR	A	NOT
0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	1	1	1	0
1	0	0	1	0	1	1	0	1		
1	1	1	1	1	1	1	1	0		

Figure A.2: Boolean truth-tables

Figure A.2 shows the four primitive Boolean logic truth tables. The Boolean AND function (a.k.a conjunction and written symbolically using \wedge) outputs true only when all of its inputs are true. The example shown here uses only two inputs (it is a *binary operator*), but can easily be extended to an arbitrary number of inputs. The Boolean OR function (a.k.a. inclusive OR, disjunction, and written symbolically using \vee) outputs true when any of its inputs are true. Again, this can be extended to any number of inputs. The Boolean XOR function (a.k.a. exclusive OR, and written symbolically using \oplus) is true when only an *odd* number of its input bits are true. The Boolean NOT function (a.k.a. negation and written symbolically using *or*!, or with a bar over a variable \overline{X}) is a *unary operator*, it only operates on a single input, and flips the bits from true to false or vice versa.

Most programming languages provide two approaches to using the Boolean operators: logical operators and bit-wise operators. The logic operators are used to connect Boolean expressions to create more complex tests. For example, the common *logical operators* give rise to expressions like `while (A < B) && (C != 0)`, which is read as “while a is less than B and C does not equal 0, do something.” This example uses the *relational operators* to compare the values, and return zero for false and non-zero for true. The *logical operators* connect these results together, resulting in a single

value of 0 for false or non-zero for true for the entire formula. So, `0b10 && 0b01` is true, `0b10 & 0b01` is 0.

The bit-wise version of the Boolean operators are used to perform Boolean arithmetic. Values are loaded into the ALU, and instead of adding bits together, we can apply the operations to each pair of input bits to produce an output bit. Let n be the width of the ALU, $A[0..(n-1)]$ and $B[0..(n-1)]$ be n -bit inputs, and $C[0..(n-1)]$ be an n -bit output value, \otimes be a Boolean operator, then the bit-wise Boolean operator results in:

$$\forall_{i \in 0 \dots n} C[i] = A[i] \otimes B[i] \quad (\text{A.1})$$

<u>0b1101_0001</u>	<u>0b1101_0001</u>	<u>0b1101_0001</u>
<u>0b0011_0000</u>	<u>0b0011_0000</u>	<u>0b0011_0000</u>
AND 0b0001_0000	OR 0b1111_0001	XOR 0b1110_0001

Figure A.3: Bit-wise operations applied on 8-bit values.

Figure A.3 shows the results of the three basic operations. Notice how in each example, the Boolean function is applied to only one pair of bits at a time to arrive at its output value. The same is true of the negation operator, although its not shown. From Equation A.1 and the figure, it should be clear that the bit-wise operators can easily be extended to wider values. For example, in modern desktop processors there are 64-, 128-bit, and even 256-bit implementations of these instructions [3].

Table A.8: C Bit-Wise Operators

C Operator	MIPS Instr.	Effect
<code>&</code>	AND	Boolean AND
<code> </code>	OR	Boolean Inclusive OR
<code>^</code>	XOR	Boolean Exclusive OR
<code>~</code>	NOT	Boolean Negation
<code><<</code>	SLL	Shift Left
<code>>></code>	SRA	Signed, Shift-Right Signed Ext.
	SRL	Unsigned, Shift-Right Logical

Table A.8 summarizes the C language operators for the bit-wise operations. The table shows the correspondence between the C language operator and the equivalent

assembly instruction from the MIPS32 instruction set. Programmers must remember a few details that are common traps:

1. the `&&` and `||` operators are *logical* operators for conditional execution, they must not be used for bit-wise operations, and
2. the caret `^` is not the familiar exponent operator from arithmetic, but is the XOR operator. `4 ^ 2` will be `0b0100 XOR 0b0010 = 0b0110 = 6`, not 16, and
3. the `!` operator is the logical not, to get bit-wise negation make sure to use the tilde `~`, and
4. the shift-right operator (`>>`) will change its behavior depending on the type of left-hand value, such that if the left-hand value is signed, then it will give sign-extended left shift, and if the value is unsigned it will give logical left shift.

A.3.1 Bitwise vs. Logical Operators

It is very important for programmers to know the difference between bit-wise and logical Boolean operators, and to take great care not to confuse the two. The first difference is that the logical operators consider only the integer 0 to be false, and any non-zero value to be true. Consider the following expression:

```
1 int x = (a && b) | c;
```

There is an error in the use of the `&&` operator. If either `a` or `b` are 0, then the AND clause will also be 0. That worked as advertised. But then we have to consider the second clause which uses the bit-wise OR operator. Since the left-hand side is 0, it will OR `c` into 0, assigning `c` into `x`. So, still OK.

But, wait, there's more! For example, let `a = 0x10` and `b = 0x20`. Both of `a` and `b` are non-zero, which from a logical perspective is true, so the result is true. But if we used the correct *bitwise* version, then we would try to bitwise AND these values together, and get: `0b0010_0000 AND 0b0001_0000 = 0b0000_0000`. So, we have the case that the *logical* operation results in `TRUE`, but the *bit-wise* operation results in 0. What *should* have happened if we used the bit-wise version is that we

will bit-wise OR the value of `c` into 0, resulting in `c`. But, since the incorrect *logical* value was used, we will bit-wise OR the value of 1 into `c`. This makes `C` odd if it were even, and no change if otherwise.

The second problem with mixing the bit-wise and logical operations comes from a language optimization called *short-circuit evaluation*. Whenever the left-hand-side of a logical expression is conclusive and evaluating the right-hand-side will not change the result, the right-hand-side is never called. In effect, short-circuiting is a quick way of writing if statements without using all of the extra syntax. For example:

```
1 index = index + (index < length) && insert_value(y);
```

If the left-hand-side is true, then the right-hand-side will be evaluated. In this case, if and only if the current index is less than the length of a buffer, then we will insert the value into the buffer. Only if both were successful will the logical AND return true (1), which will be added to the index. The short-circuiting behavior can be a major problem if the wrong operators are used.

```
1 int x = a || b;
```

The previous example should have used the *bit-wise* version of OR. In this case, if the value of `a` is non-zero, then `x` is equal to one, and `b` is never read. These bugs work both ways. Using the bit-wise operators where the logical operators were needed is also a significant bug.

```
1 index = index + (index < length) & insert_value(y);
```

In this example, the bit-wise operator does not short-circuit when the index is not less than the length, and every-time this is called, we will call `insert_value`. What is worse is that in addition to calling `insert_value`, if it returned 0 to indicate a failure, then index does not go up, but we probably lost the value we were trying to insert.

Mixing the bit-wise and logical operators is a source of nasty bugs. In this author's experience, the bugs are of above average difficulty to find. During testing with small

values for data often causes these tests to pass because they just happen to evaluate to equivalent values. But as the data get larger (usually after the code ships out to production), then the bugs start to show up.

Some languages, such as Java, actually fix the problem by introducing a *boolean* data-type that takes values *true* and *false*. The logical operators don't accept any other values. The bit-wise operators don't take *boolean*, and the logical operators don't work on integers. Any unintentional mixing of the operator types results in a syntax error at compile time.

Some programmers try to repair the situation in C by adding their own data-type extensions, but ultimately, the built-in operators in C are simply mirroring the instructions of the CPU, which only process integers. Short of redefining the C language (and breaking every C program written since 1971) there is no remedy except for programmer vigilance.

A.3.2 Bit Masking

Bit-masking is a very common operation, especially in micro-controller programming. There are three techniques that are commonly used: AND-Masking or off-masking, OR-Masking or on-masking, and XOR-Masking or toggling bits.

AND Masking, or off-masking, makes use of the AND bit-wise operation where a any valued AND'd with 0 will return 0, and any value AND'd with 1 will return that value. The AND mask is used anytime we wish to selectively "turn-off" part of a value but leave the rest untouched. For example, suppose we have a 32-bit value in a variable *x* and wish to extract the upper 16-bits into a variable *y*. The C language expression *y* = *x* & 0xffff0000; will cause *y* to have only the upper 16-bits of *x*, with the lower-16 bits zeroed out.

OR Masking, or on-masking makes use of the OR bit-wise operation where any bit that is 1 in the mask will turn that bit on the result, regardless of the original value. For example, suppose we wanted to turn-on the lowest 3 bits of a value without changing any of the other bits, then we could use the following: *y* = *x* | 0x7;. As with all of the masks, the mask can be arbitrarily complex. For example, to turn on every even bit, we could use *y* = *x* | 0xAAAAAAAA; (where 0xA = 0b1010).

Listing A.11: XOR Mask Example

```

1 void xor_mask( )
2 {
3     int i,j,k,l;
4
5     int v = 0;
6     for (i = 0; i < 2; i++) {
7         for (j = 0; j < 2; j++) {
8             for (k = 0; k < 2; k++) {
9                 for (l = 0; l < 2; l++) {
10                     printf("%d\n", v);
11                     v = v ^ 0b0001;
12                 }
13                 v = v ^ 0b0010;
14             }
15             v = v ^ 0b0100;
16         }
17         v = v ^ 0b1000;
18     }
19 }

```

XOR Masking is used to toggle bits. Recall from Figure A.2 that the XOR operation will be true whenever exactly one of its inputs is true. Consider the following 4-bit example: $0x1100 \oplus 0x0101 = 0x1001$. If we assume the right-hand value is the mask, then in any position with a 1 in the mask has its output flipped from the source value, and any position with a 0 in the mask is unchanged from the source value.

A more complex XOR mask example is shown in Listing A.11. This listing uses four nested `for` loops. The inner most loop uses variable `l` prints the current value of `v`, and then uses an XOR mask to toggle the least significant bit. After the `l` loop completes, the `k` loop toggles the second bit of `v`. After the `k` loop completes, the `j` loop toggles the third bit of `v`, and finally after that loop completes the outermost `i` loop toggles the fourth bit of `v`. This strangle program is the basis for Exercise 6.

Masking is often part of a larger operation, where different masks are connected with shifts to create complex bit-wise transformations. Continuing the previous AND mask example, to make the lower 16-bits of `y` hold the upper 16 bits of `x`, we could add a right-shift operator to get: `y = (x & 0xffff0000) >> 16;`

Masking is especially useful for testing whether particular bits. For example, suppose we wish test whether the 3rd and 5th bits of a value are set. Then we can construct a mask with those values enabled (set to 1) and then write code like: `if (y & 0b0101000) {` . Bit-testing is a very common operation, especially when working with the hardware available in the micro-controller.

Listing A.12 shows a classic use of an AND-mask. In this case, one of the 32-bit wide special-function-registers has several fields, one of which indicates when there is space to write another value. Because there are multiple fields in this register, we

Listing A.12: AND Mask Example

```

1 void wait_for_space( )
  {
3   // wait for U1STA bit 9 to become 0 (buffer not full)
   while( (U1STA & (1 << 9)) == 1 ) ;
5 }

```

cannot just compare the value to 0 or 1. Instead, our decision about whether there is space depends on only bit 9 of the 32-bit register.

The AND mask can be used to extract just that one field. In this case we also use the shift-left operator to generate the mask. The shift works by moving the single bit, in the 0th position, left 9 positions. This is a common way of creating a mask.

Table A.9: Single bit masks and shift values

Shift	Mask	Shift	Mask	Shift	Mask	Shift	Mask
1 <<0	0x1	1 <<8	0x100	1 <<16	0x10000	1 <<24	0x1000000
1 <<1	0x2	1 <<9	0x200	1 <<17	0x20000	1 <<25	0x2000000
1 <<2	0x4	1 <<10	0x400	1 <<18	0x40000	1 <<26	0x4000000
1 <<3	0x8	1 <<11	0x800	1 <<19	0x80000	1 <<27	0x8000000
1 <<4	0x10	1 <<12	0x1000	1 <<20	0x100000	1 <<28	0x10000000
1 <<5	0x20	1 <<13	0x2000	1 <<21	0x200000	1 <<29	0x20000000
1 <<6	0x40	1 <<14	0x4000	1 <<22	0x400000	1 <<30	0x40000000
1 <<7	0x80	1 <<15	0x8000	1 <<23	0x800000	1 <<31	0x80000000

Table A.9 shows the 32 possible shift values and the equivalent hexadecimal mask. A programmer can decide to use either the shift or the hex value to create a mask. Further, these values can be connected with the bit-wise OR operator (an OR mask) to construct additional masks. It turns out that because each of these is a power of two, we simply add these values together instead of converting them to bits. For example, to create a mask of the first 4 bits, we can simply add the masks together: $0x1 + 0x2 + 0x4 + 0x8 = 0xf$.

For example, in the while loop of Listing A.12, if we wanted to terminate the loop when any of the lower-four bits or the 9th bit is set, then we can re-write the loop as:

```

1 while (U1STA & ((1 << 9) | 0xf))

```

Note that this example combines both shift and explicit masks. We could just have easily made it:

```
1 while (U1STA & 0x20F)
```

or even

```
1 while(U1STA & (1 << 0 | 1 << 1 | 1 << 2 | 1 << 3 | 1 << 9))
```

each of which are equivalent as far as the compiler is concerned.

Each of the three examples evaluate to the same value (0x20F). A common mistake new programmers make is understanding how they can be equivalent. The second example that uses the explicitly computed value seems like it should be more efficient than the third example with all of those shifts and bit-wise OR operations. Most C compilers perform significant optimizations on the code. This is an example of one such technique, *constant folding*, where the C compiler can evaluate all of these operations at *compile time* [1]. The C compiler can perform the shifts and OR operations at compile time because all of the inputs to the operators are constants. Thus, due to constant folding, any time the C compiler can determine the value of any expression at compile time, it gets optimized into a constant value. Since the compiler optimizations make these equivalent, we need to decide which version is the most readable / maintainable and use that.

A.3.3 Mask of Least Length

One often over-looked trick in constructing mask values is to use determine whether the direct mask or its negation (~) takes fewer values. For example, suppose we wanted to turn-off only the 3rd bit of a 32-bit value. While we could construct the mask as 0xfffffff7, humans need to think about exactly which bit is being turned on/off in that mask. Instead, we could just as easily construct the mask as (~ (1 << 3)), and let constant folding turn that into the larger, complex mask above.

A.3.4 Flag Variables

Flag variables are closely related to the masking of the previous section. A flag variable is an integer variable where various bits are set to indicate particular conditions. The advantage is that a number of conditions can be represented in one flag variable.

Listing A.13: Snippet from Linux `fcntl.h`

```
1 // from Linux <bits/fcntl.h>
2 #define O_ACCMODE      0003
3 #define O_RDONLY      00
4 #define O_WRONLY      01
5 #define O_RDWR        02
6 #define O_CREAT        0100 /* not fcntl */
7 #define O_EXCL         0200 /* not fcntl */
8 #define O_NOCTTY       0400 /* not fcntl */
9 #define O_TRUNC        01000 /* not fcntl */
10 #define O_APPEND       02000
11 #define O_NONBLOCK     04000
12 #define O_NDELAY       O_NONBLOCK
13 #define O_SYNC          010000
14 #define O_FSYNC        O_SYNC
15 #define O_ASYNC        020000
```

This makes it easy to pass these values between functions, keeps the status values together, and makes efficient use of the program memory.

Listing A.13 shows a classic example of flag variables. The `open` system call takes a single integer called `mode`, which is a bit-wise OR of one or more of these flags. For example to open a file for reading and writing, and to create the file if it does not exist, and to append data if the file does exist, we would call `open(filename, O_RDWR | O_CREAT | O_APPEND)`. These three values are OR'd together (at compile time using constant folding). The Linux kernel will handle this system call, and will need to test each of these flags using AND-masking. For example, there could be code like `if (mode & O_CREAT) create_file()`.

Listing A.14: Example of a struct

```
typedef struct {  
2   int curr_count;  
    int max_count;  
4   int num_overflow;  
    int alarm_status;  
6 } device_status_t;  
  
8 device_status_t status = { 0, 1000, 0, 0};  
  
10 void status_update(device_status_t * const status)  
{  
12     status->curr_count++;  
    if (status->curr_count > status->max_count) {  
14         status->curr_count = 0;  
        if (++status->num_overflow > 10)  
16             status->alarm_status = 1;  
    }  
18 }
```

A.4 Structs

Structs are the work-horse data structure for any C language application, are part of the basic C background students should have before reading this text. A brief overview is presented here. Structs allow a number of values to be stored together in memory and identified with a common name. This block of data can be passed around the program through a single pointer to the start of the struct. This adds in consistency by keeping related data elements together. It also is a convenient way to both send multiple values into a function and return multiple values from a function.

A `struct` consists of a number of fields or members (use terms interchangeably). The compiler keeps track of where each member starts relative to the start of the struct. The members can be accessed through the base address of the struct using the “.” notation if the struct base is an automatic variable, or using the dereference operator (“->”) if the variable is a pointer to a struct (pointers are discussed in Section A.8).

A common convention is to use the C keyword `typedef` to declare a new type name to define the struct. The typical form is to suffix the type name with an “_t” to indicate that it is a type.

Listing A.14 shows the declaration of a new struct type (lines 1-6), declaration of a global, automatic type; and static initialization on line 8. It also shows passing a struct into a function on line 10, and finally using the members in the rest of the program.

The struct will occupy as much space as the total of all of the members in the struct. For example, the struct in Listing A.14 uses 4 integers, each of 32-bit (4 bytes), for a total of 16-bytes. The C built-in instruction `sizeof(status)` would return the

Listing A.15: Nested Structures

```
1 struct date {  
2     uint8_t day;  
3     uint8_t month;  
4     uint16_t year;  
5 };  
6  
7 struct log {  
8     uint32_t id;  
9     struct date log_date;  
10    struct date clear_date;  
11    char message[1024];  
12 };  
13  
14 void make_entry( )  
15 {  
16     struct log entry;  
17     entry.id = 100;  
18     entry.log_date.day = 0;  
19     entry.clear_date.day = 19;  
20 }
```

value 16.

A.4.1 Nested Structs

C allows structs to be “nested,” that is, one struct holds another struct. Just like any other type, a struct can be used to create a field inside another struct. As a field of a struct, the nested struct gets a name, and that is part of the access path to get to the nested fields. Listing A.15 shows a simple example of a pair of structs, one holds a date, and is nested inside a struct that holds an event to be logged. Note that there are two dates, each field has a different name and each provides its own set of date fields.

A.4.2 Copying Structs

Because structs are stored contiguously in memory, they can be copied in memory using any of the built-in memory copy functions. For example, to copy the bytes of memory of one struct to another struct, we can use `memcpy(dst, src, sizeof(src))`. This makes them very handy for doing bulk storage to another device. For example, we could use the `read` or `write` system call in traditional C to read or write the bytes of a struct to/from an I/O device.

Listing A.16 shows an example of this. Suppose the function `dangerous` could modify the contents of the struct, and we don’t want that to happen. In this listing, a copy of the struct is made using a local (automatic) struct. Then, the `memcpy` func-

Listing A.16: Example of making a copy of a struct

```
1 #include <string.h> // prototypes for memcpy
3 // call a function that might change the fields
  // of a struct, without letting it change the
5 // original src struct.
void dangerous_fun(const data_t *src)
7 {
  // declare a temporary copy
  data_t tmp;
9
  // copy the bytes of the src to the tmp
  memcpy(&tmp, src, sizeof(data_t));
13
  // call the function on the new copy
15 dangerous(&tmp);
}
```

tion is used to copy the bytes from the `src` struct into the temporary struct. The `dangerous` function is called on the temporary struct (using the address of operator to get a pointer to the local variable). After the call to `dangerous`, the function exits, and the local, automatic copy is discarded, and the original struct is untouched.

Shallow vs. Deep Copy The previous example works only when the struct does not include pointers to other data structures. The copy made in the previous example would copy the pointers, not the values that they point to. This is called a *shallow copy*, which leaves the pointers intact, and exposed to the `dangerous` function. If there were pointers, then the shallow copy can be made to the temporary copy, but then for each member that is a pointer, another chunk of memory would need to be allocated, the `src` would need to be `memcpy()`'d. Before the function exited, the allocated memory would need to be discarded (unless `dangerous` made copies of that memory itself). Managing this deep memory can be one of the most challenging aspects of programming. Fortunately for us, this typically does not occur in micro-controller programming where the memory is too small to even admit this kind of technique. The topic is brought up here to made the reader aware that this is a limitation of the copy technique, and that this deep-copy should be avoided by redesigning the application if at all possible.

Listing A.17: Example of a bitfield

```
typedef struct {  
2   unsigned red:5;  
   unsigned green:6;  
4   unsigned blue:5;  
} rgb565_t;  
6  
rgb565_t pixels[320][200];  
8  
10 pixels[0][0].red = 3;  
   pixels[0][0].green = 16;  
   pixels[0][0].blue = 4;
```

A.5 Bit-Fields

Bitfields are an important variation of the normal C language struct, and are not typically covered in introductory C courses. A bit-field allows declaration of fields in the struct that are *smaller than* a build-in data type! The syntax is to declare the type to be a signed or unsigned value, give the name, and then a “:” followed by the width in bits. The total size of the struct will be the size of the bits, all packed together, and rounded up to the smallest built-in type that will hold the values. The C compiler will generate the optimized instructions to extract the proper value from the bit field.

Example A.5.1. A common way of representing colors in computer graphics is known as the RGB565 format. This format fits the color of a pixel into a 16-bit value, which is a very compact representation while preserving much of the important color data. The format gives 5 bits each for the red and blue channel, and 6 bits for the green channel (the human eye can see more green than the other colors).

Listing A.17 shows a bit-field declaration that can store this format. The struct has three members named `red`, `green`, and `blue`. Because this is a bit field (we know this because of the `:` size in the struct declaration), we can add up the bits, divide by 8 bits per byte, and take the ceiling to arrive at the proper number of bytes. There are 16-bits, which means this struct should occupy 2 bytes.

After the declaration of the bit-field, an array of $320 \times 200 = 64,000$ `rgb565_t` pixels is declared (this array will require 128,000 bytes). Then the first pixel is initialized to a color by accessing each of its member fields (a slightly greenish-gray color).



When using the bit-field members as a destination for another operation, the C compiler will truncate the results to fit in the bit-field so that over-flows don't clobber the other fields. When using the bit-field members as source, their values will be sign-extended to fit into a matching source register.

Listing A.18: Example of a bitfield

```
1 typedef struct {  
2     unsigned red:5;  
3     unsigned green:6;  
4     unsigned blue:5;  
5 } rgb565_t;  
  
7 rgb565_t pixels[320][200];  
  
9 pixels[0][0].red = 3;  
10 pixels[0][0].green = 16;  
11 pixels[0][0].blue = 4;
```

A.5.1 Data Alignment in a Bit-Field

Memory alignment, the rules that determine where data can start in memory, is an important aspect of programming that can be largely ignored by applications programmers; but is of vital importance when working with micro-controllers. Most 32-bit processors have stringent rules stipulating that data loads and stores must be aligned at a word boundary. Non-aligned loads and stores are done so at penalties, often involving multiple memory operations followed by bit-wise operations to give the illusion of an aligned operation.

Because of this, the C compiler will go to great lengths to ensure all data items are aligned. The bit-field of Listing A.17 appeared to only require 16-bits. However, applying the `sizeof` operator shows that the C compiler actually counts it as 4 bytes due to word-alignment. Applying the `sizeof` operator to an array of two elements shows that the array occupies 8 bytes. This means that our previous example of a graphics buffer would require $320 \times 200 \times 4 = 256,000$ bytes, or twice the storage we expected.

Not only does this waste space, but when working with hardware, would be wrong. In the case of our graphics buffer, the graphics hardware will expect each pixel to be adjacent. It would expect that each pixel is aligned on a *pixel boundary*, not the processor's *word boundary*. The C compiler has no way to know this without additional information from the programmer.

To request that the compiler over-ride its default alignment rules on a particular definition, we need to give the compiler additional information. This comes in the form of a C compiler extension called an *attribute*. In this case, to get the data packed into the smallest space possible, that is, *byte aligned*, we need to ask that the compiler *pack* the data. To do this, we use the modifier: `__attribute__((packed))` on

Listing A.19: Definition of a SFR

```

1 typedef struct __attribute__((__packed__)) {
2     unsigned :1;
3     volatile unsigned RC1:1;
4     volatile unsigned RC2:1;
5     volatile unsigned RC3:1;
6     volatile unsigned RC4:1;
7     unsigned :7;
8     volatile unsigned RC12:1;
9     volatile unsigned RC13:1;
10    volatile unsigned RC14:1;
11    volatile unsigned RC15:1;
12    unsigned :16;
13 } portc_t;
14
15 volatile portc_t *PORTC = (volatile portc_t *) 0xBF886090;
16
17 PORTC->RC1 = 0x2;

```

the struct containing the bit-field. Listing A.18 shows the proper use of this attribute. This instructs the compiler to align the members on bytes, and arrays of structs to be consecutive in memory. The start of a struct or an array of structs will still be word-aligned.

In our prior example, using the packed attribute will cause each pixel to be 16-bit, and the graphics buffer of 320×200 pixels to be the expected 128,000 bytes.

A.5.2 Bit-Field Definitions of Special Function Registers

Listing A.19 shows the definition of a *special function register* (SFR). These will be explained in more detail in Section A.13. Micro-controllers can have hundreds of SFRsB, each SFR can have many different fields mapped into it. The SFR's values are mapped to the micro-controller's hardware control lines, so to change or query the state of the machine's hardware resources, we need to be able to read and write these SFRs.

Listing A.19 shows the definition of the definition of an actual SFR from the PIC32 architecture. The definition shows unnamed entries as place-holders, they will consume space, but cannot be used to store values. Each of the 8 bit fields are each 1-bit wide. The total storage of the struct is 32-bit. Because each bit corresponds to hardware, each one is volatile. The typedef'd struct is used to declare a value called PORTC, which is const pointer to a volatile portc_t. The initializer tells the compiler where the register is defined in memory. The type-cast is used to ensure that the compiler doesn't balk at the odd combination of modifiers.

Line 18 of the code is interesting. The right-hand side evaluates to the value 2

Listing A.20: Example of bit-field C code

```

1 struct {
2     int a:1;
3     int v:6;
4 } base;

6 void example(int x)
7 {
8     base.a = x;
9 }

```

Listing A.21: Translation of Listing A.20 without bit-fields.

```

1 struct {
2     int a_and_v;
3 } base;

5 void example(int x)
6 {
7     int tmp = base.a_and_v & ~1; // mask out the least-bit
8     int tmp2 = x & 1;           // keep only the least-bit
9     base.a_and_v = tmp | tmp2;  // or the two values together
10 }

```

(in binary it is 0b0010). The left-hand-side of the assignment statement is a 1-bit wide field in the bit-field. The C compiler will truncate the right-hand-side of the assignment to be 1-bit wide to match the left-hand-side, which means the right-hand-side value will become 0. Thus, the bit that corresponds to RC1 of the bitfield will be assigned the value 0, all other bits will remain untouched. Because this is volatile, the write to memory will happen immediately.

A.5.3 Instruction Set Extension

Bit-fields are great examples of “syntactic sugar,” it hides the complexity that the underlying machine may experience in executing the code. Listing A.20 shows a simple use of bit-fields. The example copies the least-significant bit of `x` into the least-significant bit of the `base` value. This is not an operation that most general-purpose instruction sets are optimized to handle. Many general purpose processors will translate the bit-field operation into bit-wise operations.

Listing A.22: Assembly interpretation of Listing A.20

```

L1:          ; base.a = x;
2  lw      $t0,-8($sp)    ; load first byte of base
   lw      $t1,-16($sp)   ; load x into $t1
4  li      $t2,1          ; load 1 into $t2
   and     $t1,$t2        ; $t1 only has bit 0 of x
6  not     $t2,$t2        ; $t2 not has 0b11111...1110
   and     $t0,$t0,$t2    ; and mask least bit off of $t0
8  or      $t0,$t0,$t1    ; or mask lsb o $t1 into $t0
   sw      $t0,-8($sp)    ; store $t0 into base.a

```

Listing A.21 shows what this might look like. The least-significant bit of the `base.value` must be cleared without changing the rest of the bits. Only the least-

significant bit of the `x` value can remain, so the upper-bits are masked off. Finally, it is safe to logically OR these words together. Only the least-significant bit can change. As one can imagine, this will require a number of machine language instructions. Listing A.22 shows the general MIPS32 Assembly language translation of this C code. The same progression of AND, OR, and NOT instructions can be seen in both the C and Assembly language versions of this code.

Listing A.23: Extended Assembly interpretation of Listing A.20

```

L1:      ; base.a = x;
2 lw     $t0, -8($sp)    ; load first byte of base
   lw     $t1, -16($sp)   ; load x into $t1
4 ins     $t0, $t1, 0, 1   ; insert into $t0, from t1, 1 bit start @ 0
   sw     $t0, -8($sp)    ; store $t0 into base.a

```

As mentioned in Section A.5.2, bit-fields are natural choices for representing the special-function registers that are used extensively throughout a micro-controller. The typical micro-controller program executes a considerably greater percentage of instructions that interact with these SFRs than the typical desktop program. Efficient support for this language construct is vital for efficient micro-controller programs; as a result most micro-controller instruction sets provide special instructions for this purpose.

Table A.10: PIC32 extensions for bit-fields

EXT	Extract from bit-field	$Rt = \text{ExtractField}(Rs, pos, size)$
INS	Insert into bit-field	$Rt[pos:pos+size] = Rs[0:size]$

Table A.5.3 shows the two instructions added by Microchip in the PIC32 instruction set. These two instructions allow the processor's ALU to either extract (read) or insert (write) values in a single cycle (rather than the five cycles in Listing A.22). Listing A.23 shows the PIC32 translation of the same example. Note that the five logical operations to extract and move bits in the standard MIPS code are replaced with one INS instruction on the PIC32.

Instruction set extensions like this are useful to improve performance, but also to improve code-density. Considering the large number of SFR operations in a typical micro-controller applications, this can lead to substantial improvements in performance and reduced program size. There is one important caveat - instruction set extensions are only useful if the programmer uses them. In this case, the C compiler

must be modified to convert the bit-field instructions to the extended instructions. So, although the PIC32 uses a MIPS32 processor core, if we used the standard GNU C compiler for the MIPS32, we would not get support for these instructions and would not be able to realize their improvements.

A.5.4 Other Uses for Bit-Fields

Bit-fields are frequently employed by micro-controller applications for general data storage. Although available memory continues to increase, there is still strong pressure to use that memory as efficiently as possible. Bit-fields allow data to be packed into minimal amounts of memory, at a cost of limiting the range of values.

Example A.5.2. Consider a micro-controller that is being used for data-logging. The micro-controller samples 8 inputs, each gives 12-bit values, ranging from $-2048 \dots 2047$. Without using bit-fields, each sample would need at least a C `short`, which uses 16-bits per value. With 8 inputs per sample, that gives $8 \times 16 = 128b/8 = 16B$. If we used a bitfield like:

```
typedef struct __attribute__((__packed__)) {  
2   unsigned sampleA:12;  
3   unsigned sampleB:12;  
4   unsigned sampleC:12;  
5   unsigned sampleD:12;  
6   unsigned sampleE:12;  
7   unsigned sampleF:12;  
8   unsigned sampleG:12;  
9   unsigned sampleH:12;  
10  unsigned sampleI:12;  
11 } sample_t;  
12
```

The `packed` attribute causes the C compiler to ignore word boundaries, which means each sample requires $8 \times 12 = 96b/8 = 12B$. By packing the data, this allows the data logger to store $16B/12B \approx 33\%$ more data without any hardware or other software changes!



Using bit-fields to store the smallest number of bits to hold the data is a very useful technique. As the example demonstrated, we can store more values in the same memory. This is important, as a simple software change can improve the number of data points stored without having to replace hardware, which is a tremendous cost savings.

Bit-fields are also incredibly useful for storing values that are in a fixed number

of states. The next example shows how the 12-bit values could be mapped into one of four logical states. Using bit-fields to store a unique number assigned to each state can really increase the number of values that can be stored in the same memory space.

Example A.5.3. Based on the data-logger of the previous example, suppose the user were only interested in knowing the range the sample was in:

$-2048 \dots -1$	LOW = 0b00
$0 \dots 99$	OK = 0b01
$100 \dots 1024$	HIGH = 0b10
$1025 \dots 2047$	VERY_HIGH = 0b11

Based on this mapping, the data-logger need only store one of the four states: LOW, OK, HIGH, and VERY_HIGH. The minimum number of bits needed to store n values is found using the formula: $\lceil \lg n \rceil$. With $n = 4$, we find we need 2 bits for each sample. Using a packed bit field with 2-bits per sample, we only need $8 \times 2 = 16b/8 = 2B$ per sample. This allows us to store $16B/2B = 800\%$ more samples using the same hardware!



A.6 Enumerations

Enumerations are special data types that associate an alpha-numeric symbol with a compiler tracked integer value. They are used to bind names to values. A simple example is:

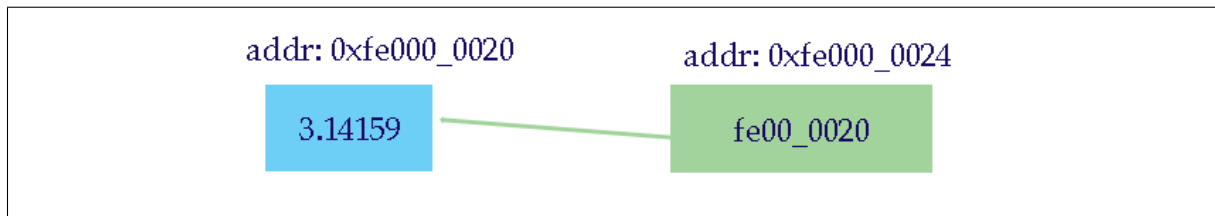
```
1 enum daysofweek { SUN, MON, TUE, WED, THU, FRI, SAT } days;  
3 days = TUE;
```

This helps avoid having magic numbers appear throughout the program. It also helps restrict the range of acceptable values the enum may take. Finally, when using a switch/case statement with an enum, the compiler will give warnings if there are enum values that don't have matching cases.

Listing A.24: Floats as integers

```
1 float f = 3.14159;  
3 int *i = (int *) &f;  
5 printf("%x %x\n", (int) f, &i);
```

A.7 Unions

**Figure A.4:** Variable and Pointer

Before defining a union, consider Listing A.24. The code gives rise to a *memory diagram* as shown in Figure A.4. The code creates a local variable, `f`, which is of type `float`, and has 4-bytes allocated from the stack (hence its high memory address). The variable `f` is assigned the 32-bits that are the `float` encoding of 3.14159. This is shown in the blue box of the figure. The code creates another local variable, `i`, which is of type `int *`, that is, it is a pointer to an integer. The size of a pointer is determined by the architecture of the machine, let's assume it's a 32-bit machine, so `i` also uses 4-bytes. The value of `i` is equal to the memory address of `f`. Because it is a pointer, the memory address is the value that is stored in `i`. This is shown in the green box of the figure. So, we have `float` with its IEEE-754 encoding of 3.14159, and we have `i` pointing to the same 32-bits as if they were an integer. As far as the machine is concerned, this is not a problem - 32-bits can be an `int`, a `float`, or even a machine instruction. The `printf` instruction exposes the magic. The format instructs `printf` to print two hexadecimal values. The first is the value of `f` *typecast* as an `int`, in this case, it will print 3. The second value will be 5832bbac, which corresponds to the bits of the floating point value in `f`, interpreted as if they were an integer.

The important concept here is that the same blob of memory can be interpreted in many different ways. In this case, we wanted to treat the same 32-bit blob of memory as if it were both an integer and a floating-point value. The example explicitly created pointers of different types and made that pointer reference the same block of memory. The `union` construct automates this process, eliminates the extra pointers, and keeps

Listing A.25: A simple union

```
1 void foo( )  
2 {  
3     union {  
4         float f;  
5         int i;  
6     } v;  
7  
8     v.f = 3.14159;  
9  
10    printf("%x %x\n", (int) v.f, v.i);  
11 }
```

Listing A.26: A union for byte-swapping

```
1 typedef {  
2     unsigned int intval;  
3     char bytes[4];  
4 } word_t;  
5  
6 int byte_swap(int v)  
7 {  
8     word_t in, out;  
9  
10    in.intval = v;  
11    out.bytes[0] = in.bytes[3];  
12    out.bytes[1] = in.bytes[2];  
13    out.bytes[2] = in.bytes[1];  
14    out.bytes[3] = in.bytes[0];  
15  
16    return out.v;  
17 }
```

the values consistent with each other.

A better definition of the union construct is that it provides multiple names for the same memory address. The union allocates enough memory to hold the longest data type. All of the members start at the same memory address. The union gives a standard way of accessing the member values.

Listing A.25 shows the declaration of a simple union variable. The union has two fields, *f* and *i*, just like our previous example. Both of the fields are 32-bit (4-bytes) long, so the union will occupy only 32-bit (4-bytes). Both *f* and *i* will be stored at the same memory address. The union uses the same syntax to access its members as an ordinary `struct`. The code stores a value into *f*, which writes the 32-bit value of a the floating point value into memory. The variable *i* is just another name for that same memory address, but it will be interpreted as integer.

The power of unions may not be obvious at first. Recall the endianness problem of Section A.1.1. Data that is in little-endian format may need to be swapped around. While we can do this with bit-field operations, we can also do this with union. Listing A.26 shows how the union can help. We create a union of an integer (4 bytes), and 4 8-bit bytes. The union operator allocates only enough memory for the longest member, they are both 4-bytes. We can write the integer value of the `in` union, then

Listing A.27: A union for storing data

```

1 typedef struct {
2     int sensors[8];
3 } sensors_t;

4
5 typedef struct {
6     int alarm_code;
7     int time_generated;
8     int time_cleared;
9 } alarm_t;

10
11 typedef union {
12     alarm_t alarm_event;
13     sensors_t sensor_event;
14 } payload_t;

15
16 typedef struct {
17     int record_type;
18     payload_t payload;
19 } record_t;

20
21 void log_sensors(record_t *record)
22 {
23     int i;

24
25     record->record_type = 1;
26     for (i = 0; i < 8; i++)
27         record->payload.alarm_event.sensors[i] = getSensor(i);
28 }

29
30 void log_alarm(record_t *record)
31 {
32     record->record_type = 2;
33     record->alarm_event.alarm_code = getCode( );
34     record->alarm_event.time_generated = getNow( );
35 }

```

read from the bytes of the `in` and write to the bytes of the `out` union, and finally return the integer from the `out`. The union operator handles all of the byte accesses to make this work.

Another example of the power of unions is shown in Listing A.27. This is a more complex example, involving mixing structures and unions together. This code might be from a data-logging application. Each event must be written to the log. Without using unions, we would need to manually structure the log-file in order to handle the different types of entries (alarms versus sensor data). Still without using unions, we would need to manually code all of the I/O routines to read and write the two different types of data, which can be tricky to keep consistent.

Using unions in this example greatly simplifies the process. Typically, there is a parent structure that holds the common payload. In this example, the common payload is the type of record (1 indicates sensor, 2 indicates alarm), but one could easily imagine other common fields being stored here (e.g. log entry number, time and date, serial number of the device, etc.). The struct also has a member that is a union. The union unifies the two different types of records. In this case, the sensor data uses 8 integers or 32-bytes, while the alarm code uses 3 integers or 12-bytes. The

union allocates the length of the largest item, so the union is 32-bytes (yes, there is wasted space in the alarm record). The code also shows how to pass these structures around and access them using indirect and direct addressing.

A.8 Pointers

Pointers are one of the most misunderstood topics of C language programming. Their use arises from the physical organization of the CPU. The registers in a CPU can hold small, primitive values such as characters, integers, and floating-points; but they cannot contain larger values. Programs work with larger data structures such as strings and structures one primitive element at a time. For example, to find the length of a string, we read one character a time, looking for a null character. This illustrates the trade between program complexity (having to loop over the string) and the limited size and number of the registers.

As a general rule, every data element, from single bytes to huge blobs of data are stored in memory, with only the actively used parts resident in the CPU's register; and it is the programmer's responsibility to keep track of where in memory everything is. Consider the case of computing the length of a string. The MIPS32 assembly code for a version is shown in Listing A.28. To compute the length of the string, the program needs to load each character, and compare it to null. The function needs to know where to start loading bytes, so the *starting address* is passed into the function in the *a0* argument register. After a byte is loaded, the program needs to compute where the *next value* should come from. Since the code is loading bytes at a time, and memory addresses are *byte oriented*, the next byte is one more than the current byte. The next byte is computed by adding 1 (byte) to the current address, and repeating the loop.

Listing A.28: Assembly version of strlen

```

# $a0 has address of string
2 # $v0 will get length of string
strlen:
4   xor    $t0, $t0, $t0 # zero out $t0
   xor    $v0, $v0, $v0 # zero out $v0 too
6   lb     $t0, 0($a0)    # load byte from first address
   beqz   $t0, 12         # if zero, brach to end
8   addi   $a0, $a0, 1     # set $a0's address to next byte
   addi   $v0, $v0, 1     # $v0 = $v0 + 1 (add 1 to length)
10  j      -20            # jump back up to lb
   jr     $ra             # jump to $ra

```

Listing A.29: Assembly version of findzero

```

1 # $a0 has address of string
# $v0 will get length of string
3 findzero:
   xor    $t0, $t0, $t0 # zero out $t0
5   xor    $v0, $v0, $v0 # zero out $v0 too
   lw     $t0, 0($a0)    # load byte from first address
7   beqz   $t0, 12         # if zero, brach to end
   addi   $a0, $a0, 4     # set $a0's address to next byte
9   addi   $v0, $v0, 1     # $v0 = $v0 + 1 (add 1 to length)
   j      -20            # jump back up to lb
11  jr     $ra             # jump to $ra

```


The use of strings in this example hid one important detail. The value that was added on line 8 was the *size* of the value that we are working with. Suppose that instead of finding the length of a string (by looking for null characters), we wanted to find the position of the first zero value in an array of integers? Recall that integers are 4-bytes each, and that arrays are *always* defined as a structure that holds a fixed number of elements of the same type and size, in a contiguous block of memory sufficient size to hold the number of elements times the size of each element. Listing A.29 shows the complete code.

A careful comparison of Listings A.28 and A.29 shows that there aren't hardly any differences. There are three:

1. the label for the assembly function on line 1,
2. the load instruction on line 6, and
3. the increment for the next address on line 8.

In summary, because machines have limited registers, we use memory to store data values. Unlike registers which have names understood by the processor (e.g. `$t0`), memory is identified by its numerical byte address. Values must be explicitly brought in from or sent out to memory using an address. The programmer or programming language is responsible for organizing the memory, tracking the start of where data is stored and its size.

A.8.1 Variables

To help organize the memory of the machine, programming languages, such as C, provide for *variables*, which should be a familiar concept to the reader. Recall that variables are *named locations* in memory, and that their type defines their size. The declaration `int x;` is a declaration to the C compiler that the programmer desires a named location, somewhere in memory, named *x*, and that it should be 32-bit (4-bytes long). The compiler keeps track of a *symbol table*, which tracks what variables are in scope, where they start, their type (and size). Whenever the value of the variable is read or modified, the compiler issues instructions to read or write the memory

Name	Type	Start	Length
example	anonymous struct	\$sp - 32	16

address associated with the symbol table that holds the variable. The type of the load/store is determined by the type of the variable. In the case of complex data, like arrays and structs, the compiler also keep tracks of how to find all of the members of the data structure.

For example, suppose we have:

```

1 struct {
2     int a;
3     char b;
4     double c;
5 } example;

```

The struct stores a 4-byte int, a 1 byte char, and an 8-byte double. Section A.5.1 described the proclivity of the compiler to optimize memory performance by using *word alignment*. Since this struct does not have the *packed attribute*, the three elements will be aligned on word boundaries, with three bytes of filler added between `b` and `c`. Therefore, the struct will occupy 16 bytes (not the 13 bytes it would if it were packed).

The name of the struct variable is `example`. If the struct is declared as a local variable, it will be allocated from the stack, which means its address will be relative to the stack pointer. The symbol table will have a high simplified entry that might look like:

Because the compiler has seen the declaration of the struct, and the symbol table tracks where the variable starts, it can now access each of its members. For example, the expression `example.a = 4;` would ultimately become `a sw $t0, -32($sp)`, while `example.b = 'a'` would become `a sw $t0, -28($sp)`. *Note: because the fields are word aligned, it is safe to use the store-word instruction, which is faster in most cases than a store-byte instruction.*

So, using the simple techniques of the symbol table and definition of a variable's type, the compiler can organize memory and issue the proper assembly language instructions load and store values. This artful trick gives the illusion that the computer stores variables, but that is all it is, an illusion; and one that is designed to take the burden of managing memory away from the programmer. The cost for ceding control

Listing A.30: A simple example using address of and pointer to

```
1 void foo(int *v)
2 {
3     printf("v: %d\n", v);
4 }
5
6 int main()
7 {
8     int x = 4;
9     foo(&x);
10 }
```

of memory to the compiler is that we no longer know where our data is stored. Do we care? The answer is, yes we do.

Consider the string length function of Listing A.28 at the start of this section. The string is actually represented by its start in memory. That is the only information that we need in order for the string length function work correctly. The starting address of the string, which is just an integer address, was stored in a register, and then the function was started. But, both the *find zero* that worked on integers and the *string length* functions expected a starting address. By sending just an address, we loose the *type* of the variable; but as far as the machine is concerned, there is no such thing as a type, that is a compiler construct. So, we need to look to the language and the compiler to address all of these issues.

A.8.2 The address of and pointer to operators

The C language adds the *address of* operator, (&), which will return the integer memory address of the variable. The operator can be used on any variable (even members of structs or elements of an array). To the compiler, the integer memory address is only part of the picture, it also needs the type. But the start of a type isn't the same thing as a type. C language introduces the concept of a *pointer to a type*. The syntax to declare a variable with *varname*, that holds an integer start address of another variable of type *typename* is: `typename * varname;` The expression is read "varname is a *typename* pointer," or "varname is a pointer to a *typename*."

Listing A.30 shows the mechanics of how addresses and pointer references work. The function `foo` is declared to have a single argument: *v that is pointer to an integer*. The function `main` declares a local variable *x* and assigns it a value, then calls `foo`. Because `foo` expects an address of an integer, the address of operator is used to get

the start address of x . If the program were to be run on a MIPS32 debugger, the `$a0` register (the first argument) would contain the address (somewhere in the stack) of where the value of x was stored by the compiler. When the function `foo` comes into scope its `$a0` register (which the program locally calls v) is a number that points to an integer. The call to `printf`, as it is written, will generate a compile time warning. But, when it runs, the program will display the contents of v . The contents of v isn't the original value of x , but rather the address that was passed in on `$a0`.

This illustrates one of the most important concepts about pointers. *All pointers in C fit inside an integer that holds the address of the variable that they point to.* This can be a difficult concept to really, totally grasp. In the previous example, the address was passed into the function `foo` through a function argument variable. That variable itself has storage - it will be big enough to hold the address. Because it is an integer, it can be used just like any other integer. In the example, it was used as an integer value in `printf`, which is not what was desired.

To get the value that is pointed at, there needs to be a way to tell the compiler to follow the pointer to get to the actual value. This is done, for primitive types, by the `*` operator, which was also used to tell the compiler that the variable is a pointer. The dual-role of the `*` operator is a common source of confusion. When it is used in the declaration of a variable, it tells the compiler that the variable is a pointer to that type. If it is used as a prefix to a variable, it tells the compiler to load the value stored in the pointer as an address, then then load the value at that address. This is called *dereferencing* the pointer, *but only on primitive variables*. The error of Listing A.30 could be fixed by simply modifying the `printf` call to use the dereference operator, and make the `printf("%d", *v);`.

Listing A.31: Dereference MIPS32 Code

```
dereference:
2  lw    $at, -8($sp)  # load value of pointer variable
   lw    $t0, 0($at)   # dereference pointer.
```

Listing A.31 shows the MIPS32 assembly code for dereferencing a pointer. The basic concept is that instead of directly loading a value with a load instruction, the address of the value is loaded into a register first. Often this will use addressing that is relative to the either the *frame pointer* or more often, the *stack pointer*. After the actual address has been loaded into a register, the second load will be absolute at that

address (0 bytes away from the given register) since the address is known.

A.8.3 Dereferencing Pointers

In Section A.4, the syntax of C's structures was introduced. To access members, we used the name of the struct, followed by '.', followed by the name of the member (e.g. `log.id = 1000`). Recall from the earlier discussion of the symbol table that the name of the struct is really just a reference to an entry in C's symbol table. The compiler is using that name to track where the variable is stored in memory. The '.' operator is syntax that the compiler uses to determine where the field is located relative to where the symbol table says the variable starts in memory. The compiler uses the definition of the struct to determine where all of the fields start in the struct (the offset), and it uses the declaration of the struct variable to determine where the struct starts in memory (the base). Simple addition will determine where the data is located in the computer's memory space.

If the variable is a pointer to a struct, then the entry in the symbol table will hold *where the pointer is stored*, and no matter how large the struct is, the symbol table's length will record the size of a pointer (4-bytes on a 32-bit machine, 8-bytes on a 64-bit machine). The compiler cannot just use the address of the pointer plus the offset of the field, the pointer variable *holds* the base address of the struct, it isn't the base itself. To avoid confusing the compiler, C language introduces a different operator, the *indirection operator* ("`->`"), which tells the compiler, in essence "follow the pointer" to get the base address.

For new C language programmers, knowing when to use the '.' and when to use '->' can be confusing and a source of errors and warning when compiling a program. Here are some guidelines that will help:

- If the variable to the left of the operator is a struct (not a pointer to a struct) then use the '.' operator.
- If the variable is a pointer, then use the '->' operator.
- If the compiler gives a warning about using the wrong type - don't ignore this, the compiler is probably correct.

A.8.4 Call By Reference

When making a function call, there is the *caller* which initiates the call, and the *callee* that is called by the *caller*. When the caller initiates the call it places the values that are to be passed to the callee into the registers and/or stack based on the *calling conventions* of the instruction set architecture (ISA). When the callee is finished, it cleans up after itself (the *the epilogue* of the function) and then places a return value into a register/stack location, also determined by the ISA.

Call by reference describes when an address of a value (i.e. a pointer to that value) is passed between caller and callee, or vice versa. The C version was shown in Listing A.30. The previous section explained *how* it is done, now consider *why* it is done. There are three principle reasons for exchanging pointers: performance, changing values, and returning multiple values.

The first reason for exchanging pointers is to avoid the cost of copying huge blocks of data into the stack to make a function call. Using a call by reference, a constant sized pointer is exchanged no matter how much data is being referenced. A pointer to megabytes of data is the same size as a pointer to a single byte. Consider the alternative. Without the ability to call by reference, a function like `strlen` would need to copy the value of the string into the stack before calling the function (so that we wouldn't need to pass a reference to it). It would take as long to copy the string as it did to compute the length of the string. In a reasonably complex program, all of these copies would quickly saturate memory with copies of copies of copies of huge data structures. The performance would be poor, to say the least, and in many cases there simply would not be enough memory to allow the program to work at all.

The second reason for using pointers is that it allows the callee to change the caller's value of the variable. When *call by value*, the caller sets-up the machine for the function call by copying the values of the function's arguments into the registers and/or stack. If the function changes those values, it doesn't matter - they were copies. When the caller gets control back, it discards those copies. There are times when it is desired to allow the callee to change values so that the caller can use them. Listing A.32 illustrates this with a simple example. Two pointers are passed to the `get_input` function. Note that because the variable `age` is already a pointer, its

Listing A.32: Call by reference - allows multiple values to change

```
1 int getinput(int *age, char *name, int max_len)
2 {
3     printf("Enter your name: ");
4     fgets(name, maxlen, stdin);
5
6     if (strlen(name) == 0) return 0;
7
8     printf("Enter your age: ");
9     fscanf("%d\n", age);
10
11     if ((*age < 0) || (*age > 110))
12         return 0;
13
14     return 1;
15 }
```

value can be sent to `scanf` which is already expecting a pointer. The same is true for `fgets`, the string `name` is already a pointer, so no need to dereference here. However, statement that checks the value of `age` does need the value (not the address), and does need to be dereferenced.

The final reason that pointers are used when calling functions is to return a value that is larger than a primitive value. **Before continuing, there are times this is OK and there are times that this is not correct in any way, and it is important to know the difference.** Suppose we have a function that converts a data structure into a string and returns it (similar to Java's `toString`). To indicate that a function returns a pointer, we use the same `*` modifier used anywhere else. For example, a function could be written like: `char *to_string(struct data *src)`. This declaration tells the compiler that the function accepts a pointer to a `struct data` variable named `src`, and will return a `char *` which is also known as a string. In order to return the string, it has to be allocated by the function. There are two ways of allocating memory: from the stack or from the heap.

The correct way to allocate memory in the callee and return it to the caller is to use `malloc` function which allocates the data from the heap. Allocations from the heap remain until they are manually released by the `free` function. Some language, such as Java, including a *garbage collector*, that can automatically track when allocations from the heap are no longer being referenced, and can clean them up. However, C language does not have a garbage collector, and it is up to the programmer to manage this memory. A *leak* happens when a program allocates memory from the heap, and then references to those allocations are lost, and the allocations were not released with the `free` function. Leaks don't cause a program to crash (right away), but will even-

Listing A.33: Example using `strdup` to make a copy of a local string. Note the use of `snprintf` to print to a string without overflowing the bounds.

```
1 #include <string.h>
3 char *to_string(struct data_t *value)
4 {
5     char buff[1024];
7     snprintf(buff, 1024, "Value %d %d\n", value->min, value->max);
8     return strdup(buff);
9 }
```

tually cause the program to use more and more memory until there is none available. A related problem is the *double-free* bug, where an allocation is released a second time. This causes corruption in the heap, and will cause the program to abort. Some implementations of the C libraries will detect this and fail fast (immediately at the point of the double-free), others will not detect the double free, and later the corruption will cause the program to crash (often at the point of the next call to `(malloc)`). Heap management has, historically, been a tough problem for C programmers. Modern tools such as *valgrind* help pin-point the causes of both leaks and double-frees.

The incorrect way to return a pointer is if the variable comes from the stack, then the stack pointer will “grow” to hold the variable, but when the function is finished, the stack is “shrunk¹.” If a pointer that was allocated from the stack is returned to the caller, then that pointer is referencing memory that will be overwritten by a future function call. This can be difficult to debug, because the values will remain until they are over-written, which may not happen for some time. In practice, pointers should *never* be returned that reference the stack.

One tool that can help this situation and reduce wasted memory comes from the standard C libraries that usually include a function called `char *strdup(const char *s)`. This function constructs a copy of a string, using only enough bytes to hold the string. The copy is allocated from the heap and will need to be freed when it is done. But it allows the program to use a large string allocated from the stack (a kind of *buffer*), fill that string with the desired contents, and then use `strdup` to get a copy of the string that was allocated from the heap. An example is shown in Listing A.33.

¹stack grows top-down, so grow = subtract, shrink = add

Listing A.34: Rewrite of Listing A.28 using pointer arithmetic

```
int strlen(const char const *str)
2 {
  int len = 0;
4 // while (str[len] != 0) len++;
  while(*str++ != 0) len++;
6 return len;
}
```

A.8.5 Pointer Arithmetic

There are times when it is useful to perform arithmetic on pointers. Remember, pointers are really just integers whose values store a numerical address in memory. As far as the CPU is concerned, pointers don't exist as a concept - it is a programming language artifice. Since they are just numbers, it logically follows those numbers can be used in arithmetic expressions. Suppose the variable x in the following expression is a pointer to an integer, whose base address was 0x1000. There are two possible interpretations of $x = x + 1$; . If this were normal, integer arithmetic, then after this expression, $x = 1001$. However, it is a pointer, and it makes more sense to add the *size* of the data type of the pointer - in this case, $x = 1004$ (4 bytes for an int). This is an example of pointer arithmetic. C language interprets any arithmetic expression on pointers to be relative to the size of the data type. Listing A.34 show a version of string copy that uses pointer arithmetic. The old array based version is shown commented out.

What is really interesting here is that the array notation is really not necessary in C. The familiar square-bracket and index notation is really just more syntactic sugar to make the code cleaner. Instead, we could do everything using pointer arithmetic. In fact, the version that uses the pointer increment will generate more efficient code in some compiler / architectures combinations. In the array based version, the array index must be multiplied by the size of the data type, and then added to the base address. In the pointer arithmetic version, there isn't a multiplication, the pointer is being incremented each time. Some C compilers will optimize the array index to the pointer arithmetic version depending on the optimization settings.

A.8.6 Void Pointers

Recall that all pointer variables are the same size, and to the computer, memory is just a big array of bytes. A void pointer allows a C program to create a pointer that points to anything. A void pointer can be type-cast to any type. A void pointer can pointer to any type. A void pointer used in arithmetic should be avoided, and different C standards have different rules about the results.

The `malloc` function returns blobs of memory from the heap and returns a void pointer, in facts its prototype is `void * malloc(const unsigned size)`. The memory that `malloc` runs can be cast to any other type, and any other type can be cast to a void pointer.

Because the void pointer has no type, it has no defined length. Usually, any length will be given in bytes (but that is up to the programer). By type-casting the void pointer into a char pointer, we can coerce the language to do pointer arithmetic on it using bytes. This type of operation is most common when interfacing with low-level systems. For example, a function to transfer blocks of data from a device to a program could be defined like `int read_data(void *start, int length)`. Internal to the function, the start address would be type-cast as a character pointer so that any arithmetic will be performed on byte addresses.

Consider the string length function of Listing A.28 at the start of this section. The string is actually represented by its start in memory. That is the only information that we need in order for the string length function work correctly. The starting address of the string, which is just an integer address, was stored in a register, and then the function was started. But, both the *find zero* that worked on integers and the *string length* functions expected a starting address. By sending just an address, we loose the *type* of the variable; but as far as the machine is concerned, there is no such thing as a type, that is a compiler construct. So, we need to look to the language and the compiler to address all of these issues.

Listing A.35: Function Pointers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int get_input_text( )
5 {
6     char str[64];
7     printf("Enter the next number: ");
8     fflush(stdout);
9     fgets(str, 63, stdin);
10    return atoi(str);
11 }
12
13 int get_input_random( )
14 {
15     return random();
16 }
17
18 int main(int argc, char **argv)
19 {
20     int debug = atoi(argv[1]);
21
22     int (*get_fn)() = debug ? get_input_random : get_input_text;
23
24     // call the function pointer (just like any other fn call)
25     // will call either get_input_random or get_input_text based
26     // on the value of debug, which isn't known until run-time.
27
28     int x = get_fn();
29     printf("%d\n", x);
30 }
```

A.8.7 Function Pointers

C language, in addition to data pointers, supports function pointers. A function pointer is a pointer variable that has, as its value, an address to code. A function pointer can be called just like any other function. Function pointers allow programs to dynamically reconfigure function calls by updating a pointer. Listing A.35 shows a simple example of using the function pointer. The important lesson from this example is the syntax for declaring a variable that is actually a pointer to a function. The type of the variable is the return type of the function, to indicate that this is a function pointer, the variable name is enclosed in parenthesis, and is prefixed with an asterick. Finally, the parameter for the function is listed without variable names. For example, to declare a function that returns a double, and takes an integer and a constant character pointer as its input argument would be: `double (*funptr)(int, const char *)`.

Function pointers are frequently used as call-back function. A call-back is most often used as part of the interface to a library routine. The library is coded to allow the use of a call-back so that the library can make calls to the application that is using the library.

Listing A.36 shows a call-back example. The function `decode_stream` takes two

Listing A.36: Function Callback

```
2 // in the library
3 void decode_stream(FILE *stream, void (*decoded_handler)(int, const char *))
4 {
5     char data[2048];
6
7     int n;
8     do {
9         n = decode(stream, data);
10        if (n > 0)
11            decoded_handler(n, data);
12    } while(n > 0);
13 }
14
15 // code that uses it - uses a call back to a function called data_dumper
16 int main(int argc, char **argv)
17 {
18     decode_stream(stdin, data_dumper);
19 }
```

variables, a stream variable, and a function pointer. The function is typical of a library routine that uses call backs. The function processes some data and then invokes the call-back function on line 10. The `main` function calls the decoder, and can fill in any function that meets the prototype given in `decode_stream`, namely a function that takes an integer and a character buffer. Note that the `decode_stream` function is not aware of exactly what the callback routine will do with the data - it simply makes the function call. when the call back is finished, it returns back to the decoder.

Call-backs, like this example, are actually fairly common in systems-level programming. Think about the implication of using a call-back. If we did not have the option to make a call-back function, then the usefulness of the library is determined by the available of functions that do exactly what the programmer needs the `decode_stream` function to do. For example, if the programmer wanted to have the decoded data to be written to a file or displays to a window, the library would need to do both of these. But with the call-back function pointer, the library's author does not need to concern himself with the way the data will be used; and the user of the library only needs to handle the call-back when it occurs, he doesn't need to concern himself with how it was computed. This insulates the programmers from each other while providing the most flexibility.

Another area that call-backs are used in is creating a configuration variable for a program. Listing A.37 shows a simple example of a struct that contains a function pointer that could be used as part of a global configuration. By using a definition like this, the program can read a configuration file, and then set the value of the function pointer based on the string in the configuration file. The example shows how a pointer

Listing A.37: Function Pointer in Struct

```
1 typedef struct {  
2     int value;  
3     int (*function)(int);  
4 } config_t;  
5  
6 read_config(FILE *stream, config_t *config)  
7 {  
8     config->function = NULL;  
9  
10    while(!feof(stream))  
11    {  
12        fgets(str, 1024, stream);  
13        if (!strcmp(str, "HANDLE_EVENTS=A"))  
14            config->function = handler_a;  
15        else if (!strcmp(str, "HANDLE_EVENTS=B"))  
16            config->function = handler_b;  
17    }  
18 }  
19 }
```

to the struct that contains the function pointer can be passed into a function. As the `read_config` function reads the configuration file, when it encounters a string, it can set the function pointer to the name of function in the code. Then, whenever the program needs to invoke one of the handlers, it can check to see if the pointer is `NULL` and skip making the function call, or it can call the one that was set at run-time by the configuration file.

A.9 Advanced Program Control

As far as the CPU is concerned, there are no for loops, while loops, function calls, or even if statements. These are artificial constructions to promote better programming. They were developed in order to prevent *spaghetti code*, which was the pejorative name to describe code that contains jumps all over the place. Languages such as Assembly and BASIC make use of branches (either conditional or unconditional) to back and forth throughout the program.

For example, consider the very simple BASIC program:

```
1 10 X = 0
2 20 PRINT X
3 30 X = X + 1
4 40 IF X == 10 GOTO 60
5 50 GOTO 20
6 60 END
```

The numbers at the beginning of each line are the line numbers and are used to identify each line. The program runs in order of increasing line numbers². Note the two GOTO statements in the program. An equivalent C program is:

```
2 int x = 0;
3 while (x < 10) {
4     printf("%d\n", x++);
5 }
```

The C program eliminates the branches and uses the loop statements instead. Although the BASIC program is easy to follow, it is harder to understand the structure of what is happening. The GOTO statements really confuse things. A famous paper, “Go To is Considered Harmful,” written Edsger Dijkstra in 1968, really put the nail in the coffin for this type of programming [4]. He not only laid out several indictments against this code, but laid out the beginnings of *Structured Computer Programming* (SCP) with its loops, conditions, and functions, and which is still the pre-dominant way of organizing a program.

Structured Computer Programming may be the way to organize 99% of source code, but there are a handful of times when it is useful to break out of that paradigm and use more custom program control. Custom program control must be used with

²The reason they go up by 10 is an old trick - it allowed a couple of new lines to be inserted between two existing lines without having to recode a program.

caution and under very limited circumstances. This is to avoid degrading back to spaghetti code; and may create serious run-time errors if used improperly. Aside from program readability, the problem is the program stack.

The C language includes three advanced control instructions: `exit/abort`, the `goto` statement, and a pair of functions called `setjmp/longjmp`.

A.9.1 Exit and Abort

Normally, a program terminates normally when the last instruction in its `main` method executes. There are times when the program cannot complete normally and should exit sooner. C language provides two mechanisms, `exit` and `abort`. The `exit` system call will cause the program to exit back to the operating system with an exit value to tell the OS why the program exited. Normally, the exit value is whatever `int main` returns. There is a related C function called `atexit` that can register a function to be called when the program terminates, and can be used to clean-up any OS resources. For example, when opening a file, the `atexit` handler could flush all file data and close the file cleanly prior to exiting.

The `abort` statement indicates that the program is terminated due to a critical failure. The `abort` function does not call any `atexit` handlers. In fact, the `abort` call will immediately terminate the program and invoke the operating system's aberrant behavior process. In UNIX systems, this normally causes the program to write a `core` file so the program can be debugged.

These are useful for systems that have operating systems, which is not true for most micro-controller systems. Calling `exit` or `abort`, if supported at all, typically invokes an infinite loop, where the CPU spins forever until it is reset. In an OS based system, program termination is normal and expected. In a micro-controller system, it is not expected, and typically indicates a critical failure of the product.

Most micro-controller programs should *not* call `exit` or `abort`. The circumstances where this is allowed is if the problem is truly one that cannot be handled by the source code. If an error is detected deep in the program, then one strategy may be to call `exit`, let the infinite loop happen. In a later chapter we will introduce the Watchdog, a device built into many micro-controllers that can detect when the device

Listing A.38: Use of `goto` for error handling.

```
1 int update_records(record_t *record )
2 {
3
4     int status;
5
6     status = lock_record(record);
7     if (status < 0) goto error_out;
8
9     status = get_record( record );
10    if (status < 0) goto error_release;
11
12    status = compute_record(record);
13    if (status < 0) goto error_release;
14
15    status = update_record(record);
16    if (status < 0) goto error_release;
17
18    unlock_record(record);
19    return 1;
20
21    error_release:
22        unlock_record(record);
23
24    error_out:
25        fprintf(stderr, "Error: could not update\n");
26        return 0;
27 }
```

is “stuck,” and then reset the device and then start running the `main` function again. It is still considered sloppy to rely on this behavior, and the use of `exit` should be avoided.

A.9.2 Labels and Goto

C language allows for unconditional branching through the `goto` statement. The mechanism works by including a label in the source code (an identifier followed by a colon). When the `goto` statement is executed, it will set the program counter to the address specified in the label. The `goto` statement can only jump to a label that is in the same function. It cannot be used to jump out of a function.

The typical use of a label / `goto` combination is to create an error handler for a function. There may be some sort of shared clean-up action that needs to happen at the end of a function.

Listing A.38 demonstrates a classic use of the `goto` statement. Some update must be made to a database record. There are two labels: `error_release` and `error_out`. If the record cannot be locked (Line 6), then jump to `error_out`, which displays an error message and returns an error code. If the record has been locked and an error occurs, jump to `error_release`, which will unlock the record (Line 24), and then continue executing the function (Lines 25-26). If there were no errors, then the normal

unlock and return OK code (Lines 18 and 19) will execute.

In this example, the `goto` statements actually do clean-up the code. Without the `goto` statements, each of the `if`, the error statements would need to be repeated, and instead of a 27 line function, the code would balloon to 39 lines, with the repeated error handling code. Not only would the code be less organized, but the final executable would be larger as a result of the additional instructions.

In this limited example, the `goto` statement was actually beneficial, but this is one of the few cases where it would. As a rule of thumb, if the introduction of `goto` statements will make the code cleaner and more efficient, then it is OK to use it. It is *never* OK to use `goto` as a replacement for the `break` and `continue` statements.

A.9.3 Jumps

C language provides a pair of functions, `setjmp` and `longjmp`. Both use a data structure of type `jmp_buf` that is used to store information such as stack pointer, registers, and the program counter. The structure is initialized with the call to `setjmp`. Essentially, `setjmp` stores the state of the CPU and then returns 0. The rest of the program can continue running.

Later, if there is some condition that requires that the program goes back to that original state, then the `longjmp` function can be used. The `longjmp` restores the stack pointer, the register, and the program counter that were stored in the `jmp_buf` by `setjmp`. By setting the program counter back to this original value, the next instruction will be the exit of `setjmp`!

Listing A.39: Function call in Assembly

```
1 somefun:
   addi $sp, $sp, -4      # $sp = $sp + -4
3   sw   $ra, 0($sp)      # spill the return address
   jal  bar               # jump to bar and save position to $ra
5   lw   $ra, 0($sp)      # restore $ra
   addi $sp, $sp, 4       # $sp = $sp + 4
7   jr   $ra              # jump to $ra
```

To better understand how this works, recall the mechanism used to implement function calls in Assembly language. Listing A.39 shows a simple Assembly language function, `somefun`, that grows the stack, spills RA, makes another function call, and then restores everything, and finally returns. This is the normal behavior of a function call.

Listing A.40: An example of using setjmp/longjmp

```

1  #include <setjmp.h>
2
3  static jmp_buf resetbuf;
4
5  void function( )
6  {
7      if (try_this() == FAILED)
8          longjmp(resetbuf,1);
9      if (try_that() == FAILED)
10         longjmp(resetbuf,2);
11 }
12
13 int main( )
14 {
15     if ((rc = setjmp(resetbuf)) != 0) {
16         // return from a long jump (an error happend)
17         fprintf(stderr, "CAUGHT ERROR: (%d) resetting\n", rc);
18         clean_up_and_reset( );
19         if (rc < 0)
20             abort();
21     }
22
23     // continue with main
24     do_stuff( );
25 }

```

What `set jmp` does is nearly identical to this. Instead, `set jmp` has a single argument, the address of where to store the information. The code for `set jmp` would be to store all of the registers of the machine (including the stack pointer, frame pointer, global pointer, and the program counter) into the buffer instead of the normal stack.

The code for `longjmp` takes two arguments, the address of a buffer previously filled by `set jmp`, and an integer that will be the return value. When `longjmp` runs, it restores from the `jmp_buf` all of the machine registers, including the frame pointer, stack pointer, but not the program counter (yet). It then stores the return value into the `v0` register (where return values go in the MIPS), and then restores the program counter into a temporary register, and then executes the jump instruction (`J $t0`), which makes the program counter the value that was read from the `jmp_buf`, which was the value that was saved previously by the `set jmp` function. So, now the next instruction will be an instruction in the `set jmp` function and not from the `longjmp` function. But because the stack and frame pointers have been reset, as far as the machine is concerned, it is just returning from `set jmp` as if nothing ever happened!

Listing A.40 provides an example of this control structure. Early in `main` (Line 15), the `set jmp` function is called on a global `jmp_buf` variable. The first time `set jmp` is called, it will return 0, so the function call continues as normal. The program runs as normal. Eventually, the code in `function` is called, and it detects an error (e.g. Lines 7-8). When this happens, the `longjmp` function is executed to initiate the error

handling. The behavior described above occurs, and control is resume in `main` at line 15, but this time, `rc` has the value of the second argument to `longjmp`. In this way, it may be possible for error handling to clean-up the error and resume normal operation, or else terminate the program (see the `if` statement on line 19).

It should be noted that when the `longjmp` instruction is executed, the state of the machine is discarded. The stack frame is not *unwound*, meaning that any functions that were on the stack aren't cleaned up. The heap is not reset, and will quite possibly be corrupt. Any open operating system resources are still allocated to the process, and are not automatically released.

This pair of functions provides functionality that is often compared to the exception handling mechanisms available in other programming languages such as Java and even C++. The problem with this comparison is that those languages do unwind the stack, and allow for the interception of the exception during the unwinding step. Although similar steps *could* be taken with `setjmp`, it would be difficult.

For example, it would be *possible* to have the global variable be a pointer to the *current* `jmp_buf`, and function calls that needed to be unwound could save the old global `jmp_buf` in their local stack, make a new `jmp_buf` by calling `setjmp`, and making the global pointer point to their local `jmp_buf`. Essentially, each function creates a linked-list of `jmp_buf`, and they can then call their own clean-up routines to handle errors, and finally restore the `jmp_buf` back to the the previous state, and either propagate the exception (by calling `longjmp`) or terminate the exception by executing code as normal. This would be a very tedious process, and one that is very prone to error. Instead of trying to force C language to do this, it may be better to either redesign the program or choose a language that actually supports exceptions (e.g. C++).

A.10 C Pre-Processor

One of the most powerful parts of the C language isn't, strictly speaking, part of the language at all! The C pre-processor is a tool that is invoked as part of the compilation process, and as its name suggests, is invoked prior to the actual C compiler. The pre-processor prepares C source files for the compiler, reading the raw C code and writing out 'cooked' code that the compiler can process.

When the C compiler (e.g. `gcc`) is invoked on a target the pre-processor is invoked on the source file. It starts by splitting the source file into lines. A scan is made to determine if the last character (except for new-line) on a line is the "continuation character" (the standard is the backslash character, '`\`'). When this is encountered, the backslash is removed, and the next line is joined with the current line, and the search is repeated. After this is complete, the pre-processor scans for comments, and then removes them. After this pre-pre-processing, the next step is tokenization. The source is tokenized into language statements. Any pre-processor instructions are interpreted (discussed later) and any defined symbols are replaced, that is, the symbol is replaced with the defined symbol's value.

A.10.1 Defined Symbols

The pre-processor has a mechanism to define symbols and attach values to them. Using these symbols allows the programmer to associate names with values, conditionally compile code, and include system, compiler, or compilation dependent values into their code. The symbols are resolved at compile time, and do not take up additional data or instruction memory. For these reasons, they are used extensively in C language and its dependents, and the lack of them in languages such as Java has lead to challenging design challenges.

There are two ways to define a new symbol to the pre-processor: command line arguments to the compiler, and the `#define` directive. In addition to the user-defined symbols, there are also compiler defined built-in symbols. Either way, the symbol name is assigned a string value, and that string value is substituted whenever the symbol is encountered. This can be done recursively, allowing for interesting and

powerful substitutions to be performed.

Symbol names can include printable characters (letters and numbers), as well as the underscore character. It is a standard convention that any symbol name is all upper-case. It is also standard convention that built-in and system provided symbols start with two underscores, while user-defined special symbols start with a single underscore, and generic names omit the starting underscore. For example: `__FILE__` is a built-in symbol name, while `_CONFIG_H` is a user-defined special symbol, while `MAX_ROWS` is a user-defined symbol.

Command-Line Arguments

Normally, when compiling C code, the compiler can be invoked such as:

```
1 gcc foo.c -o foo
```

To define a new pre-processor symbol at compile-time, using the command line, we can add an argument `-Dsymbol=value` to the command-line. For example, to define the symbol `MAX_ROWS` to be 100 at compile time, we use the following command-line:

```
1 gcc -DMAX_ROWS=100 foo.c -o foo
```

#define Directive

To define a new symbol to the pre-processing in the source code, the `#define` directive is used. A name is required, but a value is optional. Listing A.41 shows a simple example. The first pair of symbols that are defined are `BEGIN` and `END`, which have a value of `{` and `}` respectively. The next pair of symbols define `TRUE` and `FALSE` to be 1 and 0.

As a result of these definitions, whenever the C pre-processor encounters one of the defined values in the raw source code, they are replaced with their value in the cooked source code. A useful metaphor is that the symbols are replaced as if someone did a find-and-replaced in a text-editor. If the code in Listing A.41 is given to the pre-processor, it will be cooked into the code shown in Listing A.42. The pre-processor

Listing A.41: Simple #define directives

```
1 #define BEGIN {
2 #define END {
3 #define TRUE 1
4 #define FALSE 0
5
6 int procedure(int i)
7 BEGIN
8
9     if (i > 32) BEGIN
10         return TRUE
11     END else BEGIN
12         return FALSE
13     END
14 END
```

Listing A.42: Code of Listing A.41 after pre-processing.

```
1 int procedure(int i)
2 {
3
4     if (i > 32) {
5         return 1
6     } else {
7         return 0
8     }
9 }
```

directives are removed, and the symbols that were encountered are replaced with their values.

A.10.2 Conditional Compilation

The pre-processor includes a series of directives that instruct the pre-processor to include code based on the presence or value of defined symbols. The following list of conditional compilation pre-processor instructions will control whether the enclosed code will be included in the cooked output or not.

#if *constant_expr* If the *constant_expr* is true, then the code enclosed to the matching **#else** or **#endif** will be included in the cooked output.

#ifdef *symbol* If the *symbol* is defined (no matter its value), then the code enclosed to the matching **#else** or **#endif** will be included in the cooked output.

#ifndef *symbol* If the *symbol* is *not* defined (i.e. it was never **#define**'d, or it was **#undef**'d), then the code enclosed to the matching **#else** or **#endif** will be included in the cooked output.

#else Begins an alternate path to the matching conditional directive. If the matching conditional was not taken, then the **#else** path will be.

Listing A.43: Conditional compilation example

```

1 #define ROWS 100
2 #define COLS 100
3 #define INIT_ARRAY rand( )
4 #undef DEBUG
5
6 int *make_array( )
7 {
8     #if ((ROWS * COLS) > 1000000)
9         fprintf(stderr, "ERROR: ROWS * COLS is too large.\n");
10        abort();
11    #endif
12
13    int *a = malloc(ROWS * COLS * sizeof(int));
14    #if defined(DEBUG) || defined(CHECK_MALLOC)
15        if (a == NULL) {
16            fprintf(stderr, "%s:%s error could not allocate array.", __FILE__, __LINE__);
17            return NULL;
18        }
19    #endif
20
21    #if defined(INIT_ARRAY)
22        int i;
23        for (i = 0; i < ROWS * COLS; i++)
24            a[i] = INIT_ARRAY;
25    #endif
26
27    #ifdef DEBUG
28        printf("Array is allocated: %d %d %d", a[0], a[ROWS/2][COLS/2], a[ROWS-1][COLS-1]);
29    #endif
30 }

```

#elif *constant_expr* Begins an alternate path to the matching conditional, but also adds a conditional test on the *constant_expr*. It is a combination of the `#else` and the `#if` as above.

#endif Ends an `#if` conditional execution block (but not used in an `#else`).

The `#if` and `#ifdef` conditional compilation directives also allow the use of a *constant expression* term. The pre-processor recognizes certain operators:

defined(symbol) True if *symbol* is defined, false otherwise.

arithmetic The set of basic arithmetic operators: `+`, `-`, `*`, `/`, `%`

comparison The set of basic comparison operators: `<`, `<=`, `>`, `>=`, `==`

logical operator The basic Boolean logical operators: `and (&&)`, `or (|)`, and `not (!)`.

Conditional Compilation Example

Listing A.43 gives an example of using conditional compilation. Lines 1-4 include the definition of three symbols, and the un-definition of another. Whenever the pre-processor encounters the token `ROWS`, it will replace that string with the string `100`.

Whenever it encounters the token `INIT_ARRAY`, it will replace that token with the string `"rand()"`.

Line 8 opens with a `#if` directive. This is a constant expression. The pre-processor checks the values of the symbols and then evaluates them as integers, does the arithmetic, and then evaluates the comparison. If that directive is true, then lines 9-10 are included in the cooked output. Line 8 only works because the values are symbols, they are not *variables*. Variables cannot be used here, they only have a value at run-time, this is happening at compile-time (actually, even before compile-time, its a pre-processor). Since $100 * 100 = 10,000$ which is less than 100000, lines 9-10 are not included in the cooked output.

The symbols `ROWS` and `COLS` are used almost like variables in Line 13. It is incorrect to think of them as variables. They cannot be assigned values (i.e. they cannot be used in the left-hand-side of an expression), they don't take up any memory, and thus don't have a pointer value. In fact, they are really just symbols that the pre-processor will replace with their original values. So, the pre-processor will re-write the line of code to be `"malloc(100 * 100 * sizeof(int))"`, as if the programmer wrote that originally. So, as far as the compiler is concerned, this becomes a literal times a literal times a compile-time constant (`sizeof`). Whenever the C compiler encounters this pattern, it will evaluate these at compile time. The assembly code that is generated will be to setup a call to `malloc(40000)`.

If variables were used in place of the symbols, then the compiler cannot evaluate this at compile-time, but instead, emit code that will load the values of the two variables, and multiply them together and then multiply them times four, and then call `malloc`.

The next condition occurs on lines 14-19. This uses the `defined` operator, and if either of the symbols are defined, then the lines 15-18 are included in the cooked output, and they are omitted otherwise. Because of line 4, we know that the `DEBUG` symbol is not defined. The `CHECK_MALLOC` symbol is not defined in this file, but we cannot conclude that 148-18 will be skipped - the symbol could be defined on the command line.

The next condition occurs on lines 21-29. The condition on line 21 starts a check to

Listing A.44: Cooked version of Listing A.43

```
int *make_array( )
2 {
    int *a = malloc(sizeof(100 * 100 * sizeof(int)));
    int i;
    for (i = 0; i < 100 * 100; i++)
    6     a[i] = rand();
}
```

ensure that the symbol `INIT_ARRAY` is defined. If it is not, then the condition will select lines 22-24 and omit lines 25-29 (the `#else` part). If the symbol isn't defined, then we check to see if the `#DEBUG` symbol is, and if so, print an error message. However, in this example, the `INIT_ARRAY` symbol is defined, so lines 26-28 are baked into the output file.

Line 28 is interesting syntax: `a[i] = INIT_ARRAY;`. Remember that the pre-processor does a find-and-replace on tokens (like Listing A.41). If that is the case, this line will be cooked into: `a[i] = rand();`. What is really interesting is that using the pre-processor will allow us to replace the initialization function used by our code by simply changing the defined symbol's value, and remember to use its name consistently throughout the program.

.

The cooked example is shown in Listing A.44. Notice how the the pre-processor symbols are gone - they've been processed and aren't used by compiler. All of the conditional compilation is factored out, and only the code that was selected is present in the output. The raw version had quite a bit of debugging code, but this version has it all stripped away (presumably for production). Because the pre-processor stripped it out, before compile time, there is no run-time overhead to make checks to determine if debugging is included or not. The upshot is that this makes for much smaller, faster, and efficient code. The down-side is that it means we need to re-compile the code to enable debugging.

A.10.3 Defined symbols with identifiers

A special case of the `#define` directive is to declare a symbol that includes identifiers. The C pre-processor includes a special syntax that rather looks like C language's function declaration syntax; but under no circumstance should the two be confused -

Listing A.45: Defined symbol with identifier

```

1 #define PUTERR(str) fputs(str,stderr)
3
3 #define DBPUTS(str) \
  #ifdef DEBUG \
5 fputs(str,stderr) \
  #endif
7
7 PUTS("message");
9
9 DBPUTS("message");
11
11 DBUTS(0);

```

the pre-processor does not declare callable functions, but rather emits code with the values substituted for the identifiers.

The syntax appears as:

```

1 #define SYMBOL() replacement
1 #define SYMBOL(identifier1) replacement
3 #define SYMBOL(identifier1,identifier2,...identifierN) replacement

```

Listing A.45 shows two examples. The first, `PUTERR` defines a symbol with an identifier of `str`. There are two parts in this definition, the left-hand-side (`PUTERR(str)`) and the right-hand-side (`fputs(str,stderr)`). If the pre-processor encounters a line of code, for example `PUTERR("message");` which only looks like a normal C language function call. The pre-processor will:

- parse the expression to enumerate the arguments; then
- assign the first argument, the string "message", to the first identifier, `str`; then
- parse the right-hand-side of the symbols *expansion*; then
- replace any occurrence of an identifier string in the right-hand-side with the value assigned in step 2;
- replace the occurrence of the symbol with the expanded result.

In this case, the `PUTERR("message")` will be re-written as `fputs("message",stderr)`.

The second example, `DBPUTS`, uses the pre-processor's line-continuation operator (the slash at the end of each line). What happens here is that the pre-processor

Listing A.46: Get a bit from a word

```
1 #define GET(val,start) ((val >> start) & 0x01)
3 // OK
  printf("%d", GET(3235,3));
5
  // OK - if a is an integer type
7 printf("%d", GET(a,3));
9
  // OK - if a and b are integers
  printf("%d", GET(a,b));
11
  // Not OK - GET is not a function
13 // if it were, then this would print its address
  printf("%p", GET);
```

proceeds as in the previous example, except that the replacement text spans multiple lines. Some of those lines include pre-processor instructions, which are evaluated recursively until there are no further pre-processor lines.

Because that example uses conditional compilation directives, the expanded text with either be a blank line if the `DEBUG` symbol is not defined; or else it will be an `fputs` instruction with the identifier replaced as before.

The third example of Listing A.45 gives a value of 0 for the identifier. The pre-processor does not perform type-checks. When the pre-processor encounters this, it will re-write Line 12 to be `fputs(0, stderr);`, which will probably give a compilation warning about a null-pointer exception (or not depending on your compiler and enabled warnings and errors).

The important idea here is that the replacement happens, no matter what is placed in the argument list. If this is done well, it can make complex code more readable and higher-performing; but if done poorly can make it almost impossible to decipher.

Listing A.46 shows a defined symbol, `GET`, that takes two *identifiers*, `val`, and `start`. The symbol's expansion is an expression that will be evaluated to be whatever `val` is, shifted to the right and masked to get the least-significant bit.

The example on Line 4 is straight-forward. When the pre-processor encounters this line, it *expands* the `GET` symbol to its defined, and maps the values to the identifiers. After expansion, the line becomes:

```
printf("%d", ((3235 >> 3) & 0x01);
```

When the C compiler encounters this line of code, it may be able to apply *constant folding* to evaluate this ultimately become:

```
1 printf("%d", 0);
```

Lines 7 and 10 uses a variable, *a*, in the expansion. Using the expansion rules described above, the cooked code becomes:

```
1 printf("%d", ((a >> 3) & 0x01));  
printf("%d", ((a >> b) & 0x01));
```

The trouble happens on Line 14. In this example, the `printf` function is using the `%p` format specifier, meaning it will print the memory address of a pointer. This can be useful to display addresses when debugging. For example, `printf("%p", printf);` will print the address of the `printf` function. But the code on Line 14 will fail. The `GET` symbol is neither a function or a variable. It does not have a memory address. It is not called. It does not store things. This will either generate an error when the pre-processor encounters this and tries to expand it, or it will generate a compilation error when the compiler tries to evaluate the input to `printf`. Either way, this code won't compile.

New Scope and Temporary Variables

C language provides the capability to create a new scope to create variables by simply placing code inside a set of curly-braces. Whenever the curly-braces are encountered, any variables declared inside them become local to those braces, and are destroyed at the end of that scope. When writing defined symbol macros, this can be really useful. For example, a macro could be written that uses a temporary variable to perform some operation. So long as the variable is declared inside a new scope, it won't impact other variables. In fact, it can even *shadow* another variable per C language's scope rules.

The implication of using a new scope is that there are a series of operations, which for readability purposes, should not be encoded onto one line of source code. Therefore, it is often the case that the *continuation operator* is used to tie multiple lines of a definition together.

By convention, variables that are declared in an anonymous scope should be prefixed with two under-scores (`--`). This should avoid colliding with local variables

Listing A.47: Nested Scope

```
1 #define DOXTIMES(x,op) { \
2 int __i; \
3 for (__i = 0; __i < x; __i++) { \
4     op; \
5 }
6
7
8 void foo(c)
9 {
10     int x = 1;
11     DOXTIMES(c, x = x * c);
12 }
```

Listing A.48: Don't use semi-colons

```
1 #define XCUBE(x) ( x * x * x );
2
3 int main()
4 {
5     int z = XCUBE(3);
6     printf("%d", XCUBE(3));
7
8 }
```

(even if the language allows for it) because it could alter existing variables.

Listing A.47 provides an example of a defined symbol, `DOXTIMES` that has two identifiers, `x`, the number of times the operation is to be performed, and `op`, the operation that should be performed. The expansion uses a nested, anonymous scope. It declares a temporary variable, `__i`, following the double under-score convention. The for loop embeds the `op` expansion in its body. The continuation characters inform the pre-processor the whole body is part of the definition for the `DOXTIMES` symbol. Finally, the macro is used on Line 11, the raw source code hides the details of running the for loop.

Ending Semi-Colons

Whenever the pre-processor encounters a line that starts with the `#`, it interprets this as a directive and will parse it. If the directive is a `#define` directive, the line is parsed into the symbol name, and then everything after the symbol name is used as its value.

It is very common to see symbols *exclude* the semi-colon that ends a C language statement. Remember that the pre-processor isn't the compiler, and the pre-processor doesn't use statement terminators. So, if a symbol's definition includes the semi-colon, that will be part of its expansion.

The problems with using semi-colons in the definition of symbols is illustrated in

Listing A.49: Cooked version of Listing A.48

```
1 #define XCUBE(x) ( x * x * x );  
2  
3 int main()  
4 {  
5     int z = (3 * 3 * 3);;  
6     printf("%d", (3 * 3 * 3));;  
7  
8 }
```

Listing A.48. Line 1 shows the definition of a macro that will evaluate to the cube of its argument, but it includes a semi-colon. When the pre-processor cooks the code, it will encounter the symbol on lines 5 and 6. The cooked code is shown in Listing A.49.

Notice how, in the cooked version, Line 5 is structurally OK, but it contains two semi-colons. This is OK because C language allows for *null or empty statements*, which have no code. So, Line 5 is actually two statements, the assignment statement and a null statement.

Line 6 becomes a syntax error. The pre-processor did what it was told to do, replace the symbol `XCUBE` with its expansion, which included a semi-colon. The expansion created an error in the `printf` function because it was used as a function argument.

This example illustrates the problem with including semi-colons. It may be a great temptation because the symbol's definition looks like regular C code. However, it isn't C code yet - it will only be after the pre-processor expands it.

A.10.4 Built-In Symbols

The pre-processor defines two very useful built-in symbols, `__FILE__` and `__LINE__` (thats double-underscore) that contain the name of the C file that being compiled and the current line number as a string and an integer respectively.

For example, we can write:

```
printf("%s:%d hello world\n", __FILE__, __LINE__);
```

which will display something like:

```
1 file.c:323 hello world
```

Listing A.50: Example of architecture specific code

```

1  #if defined(__x86_64__) || defined(__i386__)
3  if (!read_config_file()) {
4      display_popup_window("ERROR: could not open file.");
5      return -1;
6  }
7  #else defined(__PIC32__)
8  if (!read_config_eeprom()) {
9      turn_on_error_LED();
10     printf("ERROR: could not open file.");
11     return -1;
12 }
13 #endif
    return 0;

```

The pre-processor also includes a set of built-in symbols that provide information about the compilation environment. These can be useful to enable code to take advantage of particular extensions and optimizations, or to dynamically get compile-time dependent values.

Symbol	Sample Value	Meaning
__BYTE_ORDER__	__ORDER_LITTLE_ENDIAN__	Endianess machine
__POINTER_WIDTH__	64	Bits in pointer
__x86_64__	1	Defined on x86-64 machines
__i386__	1	Defined on i386 machines
__mips__	1	Defined on mips machines

Compilers provide their own way to get the list of defined symbols. For GCC, the following command will provide this:

```
1 gcc -E -dM -</dev/null
```

Architecture Specific Code

The conditional compilation technique coupled with the built-in symbols provided by the C compiler enables a developer to integrate different versions of the code to be compiled for different target machines, in the same source file.

Listing A.50 shows a example of this technique. The preprocessor selects either the PC or the micro-controller branch based on the defined symbols. Many micro-controller packages provide even more fine-grained symbols to select the family and even specific versions of the chip. This can be useful to provide access to special features that are available on one part of a family and not on others.

Listing A.51: Example of error and warning directives

```
1 #ifdef __i386__  
2 #error This code only compiles on an embedded machine  
3 #endif  
4  
5 #if (__POINTER_WIDTH__ != 32)  
6 #warn Project was not tested for anything other than 32-bit pointers.  
7 #endif
```

A.10.5 Error and Warning Directives

Through the combination of conditional compilation and defined symbols, the pre-processor can be used to prevent code from building successfully when pre-requisites are not met. By using the error and warning directives of the pre-processor, it is possible to check either a built-in or a defined symbol, and then issue a specific error message that describes exactly what is missing or what the problem is.

The pre-processor supports a message, `#error message` that stops the compilation process and displays the text contained in the *message*. The other directive, `#warn message` displays a compilation warning, but the compilation proceeds.

Most compilers do not expand the text contained in the messages for these directives, as a result the message is displayed as is. The message is never displayed at run-time, and the messages are not included in the final executable. So the messages are expected to be read by other programmers. The text of the message should contain the reason for the error and give a clue as to what is needed to fix it.

When working with architecture specific code (see previous section), the programmer should take care to include warning/error messages when the code is being compiled for a different architecture than has ever been tested. For example, if the code has been tested for one particular chip in a family of devices, there should at least be a warning to that effect.

The other place that the `#error` directive can be used is as the default case in an `#if ... #elif` tree. If none of the conditions are met, then there may be a critical failure that should prevent compilation from continuing.

A.10.6 Include Files

The pre-processor provides a directive, `#include` which causes the line containing the directive to be replaced by the contents of the named file. The technique is used

Listing A.53: Definition of a Queue ADT

```
1 typedef struct {  
2     int num_items;  
3     int max_items;  
4     int item_size;  
5     char *items;  
6 } queue_t;  
7  
8 queue_t *queue_init(int max_items, int item_size);  
9  
10 void queue_destroy(queue_t *queue);  
11  
12 int queue_isempty( );  
13  
14 int queue_isfull( );  
15  
16 void queue_getitem(queue_t *queue, void *dst);  
17  
18 void queue_additem(queue_t *queue, void *src);
```

to provide a separation of definition from declaration. The declaration of data-types, global variables, and function signatures are contained in the *header file*.

Including the header file causes the compiler to enforce the type rules on anything that uses those declarations. This technique allows the compiler to make sure that both the implementation of the definition of the entities and any code that uses them all share the same declaration.

For example, by using the `#include` directive to include the `stdio.h` file, user's code can use the definitions of functions such as `printf`, and the compiler can verify that the function call is setup correctly and matches its declaration.

Separating the definition and the declaration allows developers to share their code as compiled libraries without sharing the source code. For example, a developer can create a proprietary library containing their code. In order for another developer to use this library, they will need both the library and the header files containing the declarations. The separation of declaration from definition is important both for commercial interests - keeping the source code private helps protect the rights of the developer; and it also helps large teams of developers by allowing them to divide the development work amongst the team and share only the compiled library and headers.

Typically the declaration of data-types that are *shared* are placed in the header file, which has a name that ends in `.h`. The definition is placed in a source file, which has a name that ends in `.c`.

Listings A.53 and A.54 show a sample of code that implements a Queue Abstract Data Type (ADT). The header file (Listing A.53) describes the `queue_t` data type, and

Listing A.54: Implementation of the Queue ADT

```

1 #include "queue.h"
2 #include <string.h>
3
4 static void queue_clear(queue_t *queue)
5 {
6     memset(queue->items, 0, queue->max_items * item_size);
7     queue->num_items = 0;
8 }
9
10 queue_t *queue_init(int max_items, int item_size)
11 {
12     queue_t *queue = malloc(sizeof(queue_t));
13     queue->items = malloc(sizeof(max_items * item_size));
14     queue->max_items = max_items;
15     queue->item_size = item_size;
16
17     queue_clear(queue);
18 }

```

then lists the function signatures for each of the functions that are expected to be used by other parts of the code.

The source code (Listing A.54) contains the definition of the declared items. The source code starts off by including the matching header file. This ensure that the source code is working with the same declarations that are used by the rest of the program. Lines 4-8 describe a *static* function (see Section A.1.5), which *should not*, and does not, appear in the header file, it is a helper function. This helper function is used to clear the contents of the queue, and could potentially be called by other functions (e.g. a delete all operation). Lines 10-18 contain the implementation of the `queue_init` function that was declared in the header file. Note that it is required that the *signature* matches. The variable names in the method signature in the header file are ignored and are not used, but the types certainly must match. The reason behind this comes from the Assembly language which underlies the C language. Recall that in Assembly language, there are *calling conventions* that govern where function arguments are placed, where return values shall be found, and what registers must be caller vs. callee saved. The method signature contains all the necessary information to setup the machine - we can setup the registers and memory for the function call, and then read back the return code (if any). The actual code isn't important (e.g. we can use `printf` without ever looking at its implementation).

Include Guards

One stipulation of the C language is that symbols cannot be redefined (unless they have the weak attribute `__attribute__((weak))`). With that being the case, then an include file can only

Listing A.55: Include Guard

```
1 #ifndef _HEADER_H
2 #define _HEADER_H
3
4 // body goes here
5
6 #endif
```

be included once - otherwise its subsequent includes would redefine every symbol, causing compile-time syntax errors. Without some mechanism for preventing a file being included multiple times, managing the order of including files would be onerous at best.

The technique of using *include guards* is a C language convention that enables programmers to add a header and footer to a header file, and then trust the pre-processor to prevent it from being duplicated.

Listing A.55 shows the template that is followed by convention. The pattern is:

1. Create a symbol from the header file's name.
2. Make the first line an `#ifndef` (if not defined) on that symbol
3. Make the next line define the symbol
4. Make the last line of the file be `#endif` to close the `#ifndef`

The symbol is derived from the header file's name by translating the file name to upper-case (because it is a pre-processor symbol), replacing the '.' in the filename with an underscore '_', and then adding an underscore as a prefix. If the header file were named `linked_list.h`, then the symbol should be `_LINKED_LIST_H`.

The use of the *if not defined* statement at the beginning of the file causes the pre-processor to check its list of defined symbols. If the symbol has not been defined, then the code is included. The very first thing the header file does next is defines the symbol.

This pre-processor trick now allows include files to be included in almost any order. It is safe, now, to include other include files without worry of creating duplicate symbols elsewhere.

Most Integrated Debugging Environments (IDE) automatically add the include guards to the source code whenever they detect that they are creating a header file.

The convention of the file name to symbol name is followed by most C libraries and most editors.

A.11 Attributes

Many C compilers provide a mechanism to communicate additional information about a variable or function to the compiler. Attributes can be used to alter the behavior of the compiler for one specific item. For GNU C and its derivatives, this mechanism is known as an attribute. Attributes can be applied to functions, variables, labels, and types. No matter the type, the syntax for an attribute is always:

`__attribute__((attribute-list))`

A.11.1 Function Attributes

The GNU C manual lists a number of function attributes, and the majority of them are to support very specific purposed beyond the scope of this text. The attributes that are most applicable to this text are listed here.

deprecated Mark a function as deprecated. Any call to a deprecated function will generate a compile-time warning.

format(archetype,string-index,first-to-check) The function uses a format string with type checking. The *archetype* is the name of a function from a list, such as *printf*, *scanf* (see GCC manual). The *string-index* indicates which parameter holds the format string, and the *first-to-check* indicates the first argument in the list to start checking.

interrupt Mark the function as an interrupt handler. Different targets have different options.

unused Mark the function as being possibly unused, and do not emit a warning if it isn't unused.

used Mark the function as used, even if it isn't reference, and always include in the output.

weak Mark the function as *weak*, meaning that another (strong) function can overwrite the one.

Listing A.56: A sample showing the *weak* attribute

```
1 // a library function
3 char getchar( )
4 {
5     return 0;
6 } __attribute__((weak));
7
9 // in a user's application
10 char getchar( )
11 {
12     while (! serial_port_rx_ready());
13     return serial_port_read_char();
14 }
```

The *weak* attribute is commonly used where a library function will provide default behavior, but another, more specific implementation, will override the default. Listing A.56 shows an example of two functions. The first one, lines 3-6 might appear in a library somewhere. The code has the *weak* attribute, and if nothing overrides this implementation, then any calls to the `getchar` function will return 0. The second part of the listing, lines 10-14 provide a *strong* implementation of the `getchar` function. Without the weak attribute, this would generate an error - only one `getchar` function can exist. But, because the first one is weak, the second one can override it. There can be only one strong implementation.

A.11.2 Variable Attributes

There are a number of variable attributes, again only an abbreviated list is given here:

aligned(alignment) specifies the alignment (in bytes) for the given variable. This causes the variable to be aligned on a memory address that is modulo the alignment value. For example, `alignment = 4` would align the variable on a word boundary.

cleanup(function) calls the named function when the variable goes out of scope.

deprecated generates a warning whenever the variable is used. useful for debugging and finding references to variables.

packed forces the struct to have all of its members packed nose-to-tail, useful for doing raw input/output.

section(name) puts the variable in a particular section.

weak marks the variable as weak, allowing a strong reference to override it.

A.11.3 Type Attributes

packed applied to a variable definition requests that the members are all packed
nose-to-tail

Listing A.57: C file with an empty signature for the Assembly Function

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void copy(const char *src, char *dst, int n);
5
6 int main(int argc, char **argv)
7 {
8     char src[1000];
9     char dst[1000];
10
11     copy(src, dst, 1000);
12
13     printf("hello world\n");
14 }
```

Listing A.58: Template function for the copy routine

```
1 void copy(const char *src, char *dst, int n)
2 {
3     return;
4 }
```

A.12 Mixing Assembly and C Language

When C language was developed, it was really just a thin veneer of Structured Computer Programming language over the assembly language. Remember that the C compiler translates the C source code into Assembly source code, and then uses the assembler to compile the program into an executable. It should not be a great surprise to learn that the compiler makes it relatively easy to embed assembly language code right into a C program.

There are three ways this can be accomplished: linking, assembly intrinsics, and embedded assembly code.

A.12.1 Linking

The most straightforward way of integrating C and Assembly (or really any other language) is through the *linker*. The linker is the final stage of building code. It has the task of collecting all of the files that contain object code and linking them together into one image, verifying that all named symbols are found in those object files or else in the included libraries, and then writing an executable.

Integrating C and any other language is as simple as constructing the object code, and including it in the linking phase.

Listing A.57 shows the C file with an empty function signature for a copy function. The signature serves as a place-holder and tells the compiler how the function should

Listing A.59: Assembly file with the code for an optimized copy function

```

1  .abicalls
2  .option pic0
3  .text
4  .align 2
5  .globl copy
6  .ent copy
7  .type copy, @function
8  .set nomips16
9  .set nomicromips
10 copy:
11     xor $8, $8, $8
12 copyL1:
13     slt $9, $8, $6
14     beqz $9, copyL2
15
16     lw $9, 0($4)
17     sw $9, 0($5)
18
19     addi $4, $4, 4
20     addi $5, $5, 4
21     addi $8, $8, 4
22     j copyL1
23     nop
24 copyL2:
25     jr $ra
26     nop
27
28 .end copy

```

be called. The rest of the C source is as normal.

The Assembly code is a little different. Listing A.59 contains the Assembly code. The C compiler can actually help us arrive at this code. By using a command line switch of `-S`, the compiler can be told to stop after writing the assembly file. So a simple function can be written in C, and then compiler can be invoked to create the Assembly file. Listing A.58 shows the template. This code was saved in a file “`copy.c`”. Then, C compiler for the MIPS processor was invoked:

```
1 mips-linux-gnu-gcc -S copy.c
```

The result was a MIPS assembly file. Based on our knowledge of the MIPS assembly, the instructions were rewritten to those shown in Listing A.59. The code uses the `lw` and `sw` instructions to move whole words at a time (instead of byte by byte as the C code did). Since the data is *aligned* on word boundaries, this should be safe to do, and quite a lot faster than byte operations.

The Assembly source was the compiled using the Assembler directly:

```
1 mips-linux-gnu-as -c copy.s
```

Compiling the Assembly source resulted in an object file. The same step was done on the C file:

```
1 mips-linux-gnu-gcc -c main.c
```

This results in both a `main.o` and a `copy.o`. These two object files can be linked with the linker, but to do this we invoke the C compiler to put them together. The C compiler understands how to build the final executable from the file names that end in `.o`.

```
1 mips-linux-gnu-gcc -o main main.o copy.o
```

The linker looks for a symbol named `main`, which it found in `main.o`. As it continued to analyze the objects, it found that the `main` function made a function call to a function with the symbol `copy` which it found in `copy.o` (the actual file names don't need to correspond to the function call names). Since all the symbols the linker was looking for were resolved, the linker is done searching and can write the final executable, which was called `main`. This executable can now be run on a MIPS system.

When the `main` function makes the call to the `copy` function, it knows how to setup the arguments to the function because of the function signature that was written in `main.c`. Of course, if the Assembly code did something different and we did not update the signature the whole will fall apart.

Ultimately, this is one way to integrate C and any other language. As long as the other language uses the same *Application Binary Interface* or ABI, this method will work. For a long time, a language called FORTRAN was the *de facto* standard in scientific computing (and there is still a lot of FORTRAN code available). Both languages use the same ABI, so it was easy to have C call a FORTRAN function or vice versa, and they could both call out to Assembly.

A.12.2 Embedded Assembly

Most C compilers provide a mechanism to embed Assembly language directly in a C function. The syntax shown here is for the GNU compiler, but most compilers provide similar functionality. The GNU compiler includes the `asm` keyword that indicates that what follows will be Assembly code. The format for the command is:

```
1  asm ( assembler template
      : output operands          /* optional */
3  : input operands             /* optional */
      : list of clobbered registers /* optional */
5  );
```

The *assembler template* is a string of assembly language instructions. Remember that strings can be placed on multiple lines in a source file and the compiler will concatenate them together. Thus, the template can either be one multi-line string enclosed in a single pair of quotations, or it can be multiple lines, each consisting of its own string. In the former case, the source code contains new-lines and tab characters which are needed by the assembler. In the latter case, the individual strings will not technically span lines, so they will need explicit new-lines and tabs.

The reason it is a *template* and not code, is that the C compiler will process the template first. The compiler has to map local variables into registers. The mapping allows symbolic names to be used in place of actual register names. The template is cooked down and then emitted into the Assembly code by the compiler.

After the assembler template there is a list of three sets of operands. The *output operands* tell the C compiler which *local variables* should be updated from which registers *after* the Assembly code is finished. The *input operands* tell the C compiler which *local variables* should be loaded into registers *before* the Assembly code is started. The final list, the *clobbered registers* tells the C compiler which registers will be over-written by the Assembly code, which means any local variable that was stored in a register must be spilled and reloaded.

The operand lists follow a pattern of

```
1 [asmName] "constraints" (cExpr)
```

This expression is interpreted by the C compiler. For input operands, the C compiler will evaluate each of the `cExpr` values, and map them to a machine register. Since we don't know *a priori* what the mapping will be, we can refer to the mapped value in the Assembly code using the `asmName`. The `constraints` is a string of character that represent *constraint codes*.

The *constraint codes* give information to the compiler about how the value is expected to be used. Common constraint codes are [7]:

- m A memory operand is allowed with any kind of address that the machine supports
- r A register operand is allowed, provided it is a general purpose register
- i An immediate register operand is allowed
- n An immediate integer operand with a known numeric value is allowed
- E An immediate floating operand (const double) is allowed, but only if matches machine type
- F An immediate floating operand (const double) is allowed
- s An immediate integer operand whose value is not an explicit integer is allowed

When given a choice of constraints, the compiler will choose what it determines to be the best for the target architecture and the surrounding C code. The interpretation of the constraint codes can be modified with one or more *modifier codes*:

- = The operand is write-only, previous values are discarded and replaced, and never-read
- + The operand is read-and-write. Without either = or +, the operand is read-only.

For example, to map a value, C local value `numElements` to an in-line assembly value referenced as `num`, that can be either in a register or in memory, and is read-only:

```
1 [num] "rm" (numElements)
```

An In-line Assembly Example

Listing A.60 gives a complete example. The assembly code should be somewhat familiar - it is a typical MIPS loop, with the `SLT/BEQZ` at the top of the loop (at label `sum1`), the body of the loop loads each value from the array and accumulates it, and then updates the index and array address in preparation for the next iteration, and finally the jump back to the top of the loop, with a trailing `NOP` in the branch delay slot.

Listing A.60: In-line Assembly Example

```

1  /*
2  Compute the sum of the given array A with N elements
3  */
4  int sum(const int *A, int N)
5  {
6      int sum = 0;    // initialize sum to zero
7
8      // use a MIPS assembly loop to iterate over the array
9      // $t0 holds the index of the current value
10     // $t1 is used as a temporary value
11     asm("\n\t"
12         "      xor      $t0, $t0, $t0          \n"
13         "sum1:  slt      $t1, $t0, %[rN];        \n"
14         "      beqz     $t1, sum2;              \n"
15         "      lw       $t1, 0(%[rA]);           \n"
16         "      add      %[rsum], %[rsum], $t1;   \n"
17         "      addi     %[rA], %[rA], 4;         \n"
18         "      addi     $t0, $t0, 1;            \n"
19         "      j sum1;                               \n"
20         "      nop;                                   \n"
21         "sum2:                                     \n"
22         : [rsum] "=rm" (sum)
23         : [rA] "r" (A), [rN] "r" (N)
24         : "t0", "t1"
25     );
26
27     return sum;
28 }

```

The first observation to make is that the block starts with the `asm` keyword, and then each line is enclosed in quotation marks, and is terminated with an explicit new-line symbol. Just as in any other Assembly programming, only labels start in the first column, instructions start later. The semi-colons at the end are places for comments. Other than that, the code *looks* like Assembly.

The next observation is the use of symbols that are not registers: `%[rN]`, `%[rA]`, and `%[rsum]`. These are the `asmName` described above. The C compiler will re-write this template with the actual registers that it picks. For example, on line 13, the `SLT` instruction, as written, will compare `$t0` to `%[rN]`, but if the compiler chose to put `rN` in the `$s0` register, then the actual instruction will be re-written as `SLT $t0, $t1, $s0`.

The operands can be observed on lines 22-24. The first operand (line 22) is the output operand, and it maps the C local variable `sum` to the in-line Assembly symbol `rsum`. The modifiers indicate that the register is write-only. As a result, the compiler *does not* copy the value of the local variable `sum` into a register before code runs (which is why the Assembly code starts with `xor`). The input operands map `A` to `rA` and `N` to `rN`. Finally, the list of clobbered registers informs the compiler that the Assembly code will destroy values in `$t0` and `$t1`. If the compiler was using those values before, it needs to spill them to memory before this code starts and then resurrect them when

they are used after the Assembly code is finished.

After the in-line Assembly code is finished, the value of `sum` will be in the machine register represented by `rsum`, so the compiler will finish this block of code by treating the output registers like an assignment statement. Finally, the C compiler resumes normal operation, and the value of `sum` is returned.

Using the `-S` trick, as in the previous section, we can get the C compiler to display the cooked Assembly code. The Assembly file will show all of the converted C code, and in the implementation of `sum`, we can find the converted in-line code, but with all operands replaced with the actual register assignments.

Defined Assembly

The C Pre-Processor can be combined to wrap in-line Assembly and make it easier to use. For example:

```
2  #define NOP() \
    asm("    add %0,%0,%0\n");
```

This creates a symbol, `NOP()` that can be used in C code to stall for one instruction cycle (it adds the zero register to itself and stores the result into the zero register).

A.12.3 Intrinsics

One final way that the C compiler exposes the underlying hardware is through *intrinsics*, which are functions that are built-into the C compiler, which can be called as if they were functions. When an intrinsic value is encountered, the C compiler will emit the equivalent assembly language code. The intrinsic may be handled as a C pre-processor directive or as a built-in operation.

Intrinsics are very tightly coupled to the compiler and the architecture. An example of the intrinsics supported by the XC32 C compiler for the PIC32MX microcontroller is provided in Section 3.1.5.

A.13 Memory Mapped Input/Output

Historically, there are two ways to interact with the input/output (I/O) devices in a computer system: special instructions and memory mapped I/O. There are advantages and disadvantages to both approaches. Modern architectures have moved to being exclusively memory mapped, but there are still successful examples of systems that use both.

Architectures such as the Intel's 80x86 (still) include special I/O instructions (e.g. `IN` and `OUT`) that read or write data to a *port* at a particular address. The idea was that by separating I/O from RAM, there can safely be an overlap between I/O port address and memory address. By using the special instructions, the CPU can direct a read/write request to either I/O or memory. However, with the development of powerful Memory Management Units (MMU) and huge address spaces, many of the arguments for special I/O instructions have gone away. In fact, having to support both a memory and a separate I/O bus has become a design liability, especially for space constrained systems.

IN a memory mapped I/O system, the system's physical address space is broken up into regions. One of those regions is usually the system's memory. The other regions are allocated to devices. The system includes some form of an bus switch that can recognize loads and stores and route them to the appropriate device. To access a device we only need to read and write to its memory address. The loads/stores will be redirected from the system's RAM to the device.

One important difference between accessing memory versus accessing a device is that typically a program does not need a specific address of memory, rather the program asks for memory (it grows the stack or requests it from the heap). Whatever memory address is returned is just as good as any other. But if we are trying to access a peripheral that has a specific memory address, it is critical that we get a reference to that particular address.

As a simple example, the PIC32 architecture, there is a peripheral that holds information about the configuration of the MCU. The details of what is stored there aren't as important as how the information can be accessed. The value, called `DEVCFG0`, is a 32-bit word that is always at memory address `0xBFC02FFC`. Values like `DEVCFG0`

are often called *Special Function Registers*, *Device Registers*, or sometimes just *Registers*. They are registers, but not in the same sense as the CPU registers. These registers are registers in the peripheral's silicon that are mapped into the CPU's memory address space. The peripheral's registers actually store the values on writes and return the values on reads.

The 32-bit special function register is actually built from a concatenation of bit fields:

bit 31 Reserved - always read/write 0

bit 30-29 Reserved - always read/write 1

bit 28 CP - Code Protect Bit

bit 27-25 Reserved - always read/write 1

bit 24 BWP - Boot Flash Write Protect

bit 23-20 Reserved - always read/write 1

bit 19-12 PWP - Program Write Protect

bit 11-4 Reserved - always read/write 1

bit 3 ICESEL - ICE/ICD Communications Selection

bit 2 Reserved - always read/write 1

bit 1-0 DEBUG - Background Debugger Enable

A.13.1 Pointers

One way to access these values is to define a pointer to a matching primitive data-type. In this case, since the register is a 32-bit value, we can declare a pointer as in the following example. The example uses the *volatile* keyword, which should be applied to any device register that can change the value of its register on its own, or where we don't write to it before reading from it (which is the case here).


```
1 volatile uint32_t *devcfg0 = (volatile uint32_t *) 0xBFC02FFC;
```

To read the value that is being referenced, we use code as in the following snippet. The pointer is *de-referenced* to get the value it points at, and then we use an AND mask to get just the lowest two bits.

```
1 if ((*devcfg0 & 0x3) == 0x2)
    debug = 1;
3 else
    debug = 0;
```

Using pointers in this manner is relatively easy to setup. It requires that the programmer connects the pointer to the correct address and data type. Accessing the fields requires that the programmer knows the numerical start and sizes. Another programmer reading this code may not be as familiar. The snippet above uses the AND mask to get the DEBUG bits, but what is special about the value 0x02? Are there any other special value?

This method leads to an explosion of *magic numbers* all over the code. What is special about the numbers? Only the programmer who put them there may know their significance, and that is very poor engineering methodology.

A.13.2 #defines

An improvement to the pointer method of the previous section is to introduce `#define` C Pre-Processor directives to give symbols to numerical values. This can really improve readability and encourage consistency. The following snippet shows the definition of the `DEVCFG0` register, as well as two of its fields.

```
#define DEVCFG0 ((volatile uint32_t *)0xBFC02FFC)
2
#define DEVCFG0_DEBUG_MASK (0x3)
#define DEVCFG0_DEBUG_SHIFT (0)
4 #define DEVCFG0_DEBUG_DISABLED (0x3 << DEVCFG0_DEBUG_SHIFT)
6 #define DEVCFG0_DEBUG_ENABLED (0x2 << DEVCFG0_DEBUG_SHIFT)

8 #define DEVCFG0_ICESSEL_MASK (1 << 3)
#define DEVCFG0_ICESSEL_SHIFT (3)
10 #define DEVCFG0_ICESSEL_ENABLED (1 << ICESSEL_SHIFT)
#define DEVCFG0_ICESSEL_DISABLED (0 << ICESSEL_SHIFT)
```

The defines follow a consistent pattern. The register is defined in a way that it can be used on the left-hand-side or right-hand-side of an expression. Then each bit-field

of the register is defined to include its:

- mask - a value that can be used as an AND mask to get the value out of the larger integer,
- shift amount - an integer that can be used to set the shift value, and
- named values - for each (reasonable) value, give the name and the value for the name.

With this definition, we can now write if statement that was shown in the previous section:

```

1  if ((DEVCFG0 & DEVCFG0_DEBUG_MASK) == (DEVCFG0_DEBUG_ENABLED)
    debug = 1;
3  else
    debug = 0;

```

This strategy is certainly more readable. When reading code, it is less important what the values are than what they represent. It is much more clear what is happening in if-statement with the names than in the previous one with just numbers. Further, this method makes it easy to access either the defined fields or the integer as a whole. The major problem with this technique is that creating all of the #defines is cumbersome; and for all of that work, the C compiler cannot verify that the proper symbol is being used in the proper way.

A.13.3 # bitfields

Bit-fields were introduced in Section A.5, and are ideal for working with values like this. For example, the DEVCFG0 register could now be defined as:

```

#define DEVCFG0 ((volatile uint32_t *)0xBFC02FFC)
2 volatile struct {
    unsigned DEBUG:2;
    unsigned :1;
    unsigned ICESEL:1;
    unsigned :8;
    unsigned PWP:8;
    unsigned :4;
    unsigned BWP:1;
    unsigned :3;
    unsigned CP:1;
12 } *devcfg0 = DEVCFG0;

14 if (devcfg0->DEBUG == 0x02)
    debug = 1;
16 else
    debug = 0;

```

The C compiler is tracking where the individual bit-fields are stored, and now the C compiler determines how to access each bit field. The big disadvantage is that the symbolic names are lost. Introducing the defines as in the previous section creates many of the same problems, especially connecting the proper bit-field to its value. Despite the problems, this is almost exactly the way the registers are actually defined by Microchip (the PIC32's Vendor). The next snippet shows this definition.

```

1  extern volatile unsigned int      DEVCFG0 __attribute__((section("sfrs")));
   typedef union {
3      struct {
         unsigned DEBUG:2;
5         unsigned :1;
         unsigned ICESEL:1;
7         unsigned :8;
         unsigned PWP:8;
9         unsigned :4;
         unsigned BWP:1;
11        unsigned :3;
         unsigned CP:1;
13    };
        struct {
15        unsigned FDEBUG:2;
        };
17        struct {
            unsigned w:32;
19        };
    } __DEVCFG0bits_t;
21  extern volatile __DEVCFG0bits_t DEVCFG0bits __asm__ ("DEVCFG0") __attribute__((section("
   sfrs")));

```

The primary difference is that there is a *union* called `__DEVCFG0bits_t`, composed of three union fields: a struct of the bit fields (almost identical to our previous example), a struct that is 2 bits long (gives another name to the `DEBUG` field), and a third struct called `w`. The point of this union is to enable access to the bit fields or the whole integer.

Another difference is that in our first example, we used a pointer to the memory address. To access one of our bit fields, we had to remember to *de-reference* the struct pointer with the `->`. The Microchip version uses an elegant trick. Note the use of the `extern` symbol, which tells the compiler that the symbol is defined somewhere else. The `extern` modifier and `__attribute__((section("sfrs")))` are used to tell the C compiler that the linker will *resolve* the symbol `DEVCFG0` in the later stages of compilation. So, instead of declaring a local variable that will hold a pointer to the real memory address, the linker will actually put this global variable at that memory address.

There are two important differences: first, there isn't a local variable to take up

space, and because the structure is actually at the address of the register, we don't need to dereference the pointer. Using the Microchip code, the if-statement would look like:

```

1  if (DECFG0bits.DEBUG == 2)
    debug = 1;
3  else
    debug = 0;

```

A.13.4 Symbolic Accessing

As mentioned before, there are times when it is beneficial to associate a name with a value instead of a magic number. The defined symbols worked reasonably well, but the compiler does not enforce any type checking rules. Since defined symbols don't have a restricted data type, there is considerable chance for error.

The alternative is to add a *wrapper function* around the access to the bit-field, very much like object oriented languages include a *getter* and *setter*. The wrapper will use an *enumeration* to restrict the acceptable values engage the compiler in strict type checking. An example is shown in the following snippet.

```

typedef enum { ENABLED=2, DISABLED=3, RESERVED=0 } devcfg0_debug_t;
2 typedef enum { ENABLED=1, DISABLED=0 } devcfg0_bwp_t;

4 static inline devcfg0_debug_t devcfg0_getdebug( )
{
6     switch(DEVCFG0bits.DEBUG) {
        case 2: return ENABLED;
8         case 3: return DISABLED;
        default: return RESERVED;
10    }
}

12 static inline void devcfg0_setdebug(devcfg0_debug_t newvalue)
14 {
    DEVCFG0bits.DEBUG = (uint) newvalue;
16 }

18 if (devcfg0_getdebug() == ENABLED)
    debug = 1;
20 else
    debug = 0;

```

Using wrapper functions with enumerations enabled the C compiler to perform type checking, and completely eliminated the magic numbers that was adding uncertainty to the previous examples. This code is more clear, and any attempt to mix values from other bit fields will be caught by the compiler as a syntax error.

An astute reader will ask the question about performance. In the previous examples, the special function register could be accessed directly with a load/store operation. Using these wrapper functions, now there is the overhead of making a function call, which requires setting up the call stack, jumping into the function, and return back from the function call. However, there is one more C trick that is exposed in this example.

inline modifier The *inline* modifier on the functions asks the C compiler not to actually make them function calls at all. Instead of making a function call, the compiler will replace the function call in the caller with the body of the function in the callee. By using this trick, we eliminate the overhead of making the function calls. For example, the call to `devcfg0_setdebug` will not actually be a function call, but will ultimately be a direct write to the register. One of the tricky rules about inline functions is that there can be only one definition, and they can only be used in the compilation unit in which they are declared. Generally, inline functions are declared in an include file that contains include guards. This way, there is only ever one definition per compilation unit (C file) and the compiler can safely keep all of the implementations the same.

A.13.5 Conclusion

All of the methods to access a special function register described in this section are legitimate methods and have been used by one vendor or another. Every vendor is going to have a set of recommended best-practices, and they should be used as a guide to determine which method to use. Unfortunately, many of these development environments have been around for 10 or more years, and internet forums are full of examples. Maybe 10 years ago, programmers used the magic numbers approach, later they switched to the linker-mapped union / bit-fields, and now they are recommending the function call / enum approach. The best advice is to use whichever method makes the best quality code, for development, testing, and long-term maintenance.

If there is an existing project that uses one method, it probably isn't worth rewriting the whole project just to "improve" the readability of the code. If the code is

really that bad and isn't reliable, then it may be worthwhile, but that is a different discussion.

A.14 Linker

The Linker is an application that is responsible for processing all of the object files (output from assembler), archives, and library files together into a final executable image. As the last stage of the compilation process, it is the Linker that resolves names into addresses and makes decisions of how to organize memory and where to place code and data. For systems with very limited memory or customized memory maps, this can be crucial. For example, when working with custom parts, its possible that amount and location of memories is different than the built-in default. In order for code to use it, the Linker needs to be reconfigured to use (or avoid) the custom memory space. This section explores what and how the linker works, and provides an introduction to how to control its behavior.

When the C compiler is invoked on a source file, for example, `gcc test.c -o test`, there is actually a whole process that the `gcc` command follows. First, it invokes the C pre-processor to resolve directives such as `#define` and `#include`. The compiler runs through the code and writes an assembly source file. The assembler is invoked and writes an object file. During compilation, the compiler converts the C into Assembly language. Figure A.63 shows the result of a simple C program compiled on an Intel machine. The assembly code includes labels, in this case, it uses a label `main` to declare a function in lines 1-3, `.LFB1` is used by the debugger, and `printf` is used as the target of the function call on line 9.

Listing A.63: Assembly code generated by compiler

```
2      .globl  main
      .type   main, @function
main:
4      .LFB1:
      pushq   %rbp
6      movq   %rsp, %rbp
      subq   $16, %rsp
8      movl   $0, %eax
      call   printf
```

The Assembler will convert each instruction into its Machine Language encoding and will resolve addresses where it can. But some symbols such as `main` need to be referenced by other code (the program startup code needs the address of `main`, so it needs to be a global, external symbol); or `printf` which is code that is from an external library. The assembler creates an *object file*, that includes the chunks of assembled machine language code, but also contains static data (constants, literals),

uninitialized static variables, segment relocation information, and dynamic linking information.

Large projects are usually broken down into multiple source files. This helps keep code logically organized for programmers, and helps tools such as code analyzers and revision control systems. This also improves compilation times since only changed source files need to be recompiled. Each file is compiled into its object file, and written on disk. The object file is a standard format, (e.g. *DWARF* or *COFF* formats). The language used to create an object file is irrelevant. By the time it is compiled into an object file, it is machine language code with unresolved symbols. This is shown in the first part of Figure A.5 where C, C++, Fortran, and assembly files are converted into object files.

One of the great surprises for those new to C language is that the `main` function is not the first code that is ever executed when their program runs. Instead, during the linking phase, the linker is configured to include *prologue* and *epilogue* that will interface with the operating system, setup the basic I/O devices, and load any dynamic link libraries that are needed by the application. In Windows systems, these are *DLL .dll* files, and in Linux systems, these are *Shared Object .so* files. The run-time linker will resolve all symbols or terminate the program with an error. After all of the setup and initialization of the *prologue* is complete, it will execute a function call to the programs *main* method. When *main* exits the function call, the *epilogue* will run to properly clean up. For example, to build a runnable image for an GNU Linux / Intel machine, the linker will include: `crt1.o`, `crti.o`, `crtbegin.o`, any project objects; and then `crtend.o`, and `crtn.o`. Together with any system required or project dependent libraries, the linker creates an executable.

An interesting experiment can be done using the GNU C compiler (`gcc`) and binary utilities (`binutils`). Use the compiler to compile even a trivial *hello world* application: `gcc -v -o hello hello.c`. The *command line parameter* of `-v` asks the compiler to be verbose, and it will display all of its actions, including the pre-processor, compiling, assembling, and linking phases. There are some occasions where this information can be helpful for resolving strange errors.

To compile a series of smaller C files into a complete executable, one needs to

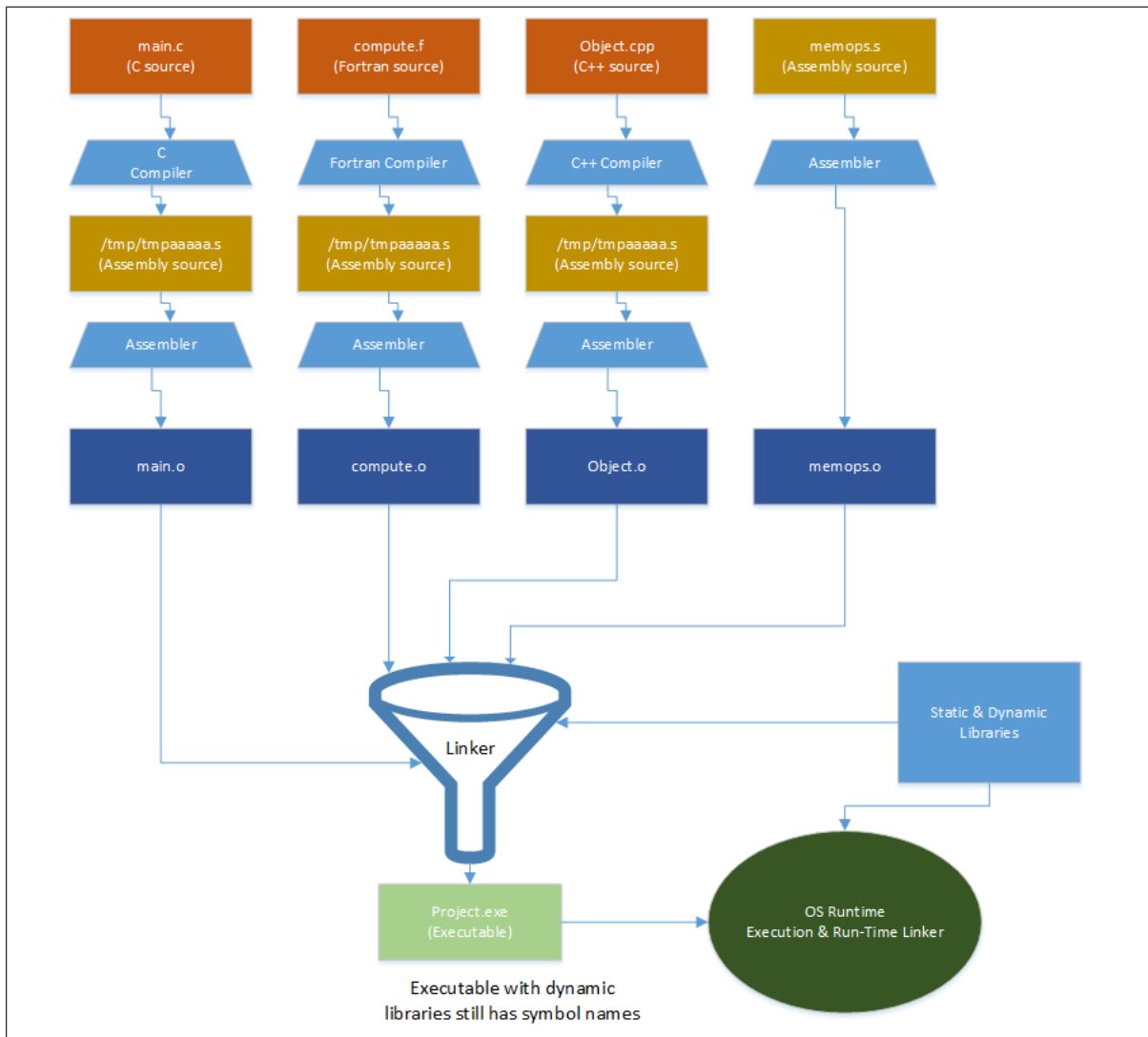


Figure A.5: Multiple source files are compiled into Assembly files, then assembled into object files, and linked with system libraries into a final executable. When the user submits a program to the operating system, the run-time executive uses the available dynamic link libraries to resolve dynamic symbols, and then the code is scheduled by the operating system.

invoke the C compiler several times. For example, if there are three files: `main.c`, `one.c`, and `two.c`, to compile each one separately into an object file, and then create the final executable, the GNU C compiler syntax would be:

```
1 gcc -c main.c
   gcc -c one.c
3  gcc -c two.c
   gcc -o executable main.o one.o two.o
```

A.14.1 Libraries

Depending on one's operating system, the default C library will include a basic set of behaviors that are needed for an executable in that environment. For example, compiling for UNIX systems will include the C library, which is a rather large set of built-in functions, including basic I/O, system calls, networking, string, and memory operations. Other libraries may be included to expand the available list of functions. For example, the math library is needed for operations such as `sin` and `sqrt`; and the pthread library is needed for creating multi-threaded applications.

Generally, the documentation will describe if and when any libraries are needed. To compile a project with a custom set of libraries, the standard GNU linker includes two command line arguments, `-l` and `-L`. The `-L` argument is used to tell the linker what directories to search for a library, and the `-l` tells the linker which libraries to include. One common mistake with the `-l` option regards how to tell the linker which file to use. The `-l` is a pattern for the linker, not a file name. In a UNIX or similar system, if the parameter is `-lm`, then the linker will look in its directories for a file named `libm.so`, and if there is no shared object, then `libm.a`.

Dynamic vs. Static Linking The difference between a `.so` file and a `.a` file is that the `.so` file is *relocatable*, meaning that the code can be loaded anywhere in memory, and all addresses are *relative*. This creates *position independent code*, and is critical for a dynamic link library which cannot have objects at the same address. The `.a` file is an archive file, which is an amalgamation of previously compiled object files, and as object files, can be linked directly into an application. When a program is compiled with an archive file, the objects in the archive are built into the program image (just

like the program's own objects) and the symbols are resolved during linkage. When the program is run, program does not need to look for the archive file, its objects are already loaded.

A program that is built without any external dependencies is said to be *static*, and a program that does depend on dynamic link libraries is said to be *dynamic*. When building code for systems, such as micro-controllers, that don't have an operating system, building dynamic does not make any sense - there isn't an operating system with a file system to load libraries. Code is normally built *static* for these systems. The command line argument to the linker, `-static` will force the executable to be static binary. Any dynamic libraries will have their relocatable code built into the final image, and there won't be any run-time dependencies from the linker.

Even on systems that do have an operating system and dynamic linking, static programs may be a good idea. Compiling a program as a static executable will remove any run-time dependencies. As surrounding operating system and applications are upgraded, new bugs and incompatibilities may be introduced into a program. For example, a program may be compiled, tested, and working fine with one version of an OS library, but an OS upgrade "fixes" a bug in the library that is incompatible with the application. As a result, the application breaks as a result of the OS upgrade, even though the application isn't changed. Compiling the application as static would fix this. In fact, static compilation is often a requirement for tested/certified code. Why not compile everything as static? The first reason is that if the application is compiled against a library that had a bug in it, an update to the library may fix the bug, and the application will work without having to change the application. The irony of updates breaking and fixing bugs (possibly both at the same time) is not lost on programmers that have to support an application over a long period of time. The other reason against using static for everything is that the static binary will be larger than the dynamic binary. The dynamic image will include only the stubs for function calls that are in libraries; while the static version will *not* include the run-time linker code, but *will* include all of the code used by an application. For example, a simple *hello world* application compiled for a Linux-Intel machine was 8 kB as a dynamic executable, but was 840 kB as a static binary.

A.14.2 Undefined References

When breaking down a project into multiple object files, each object file defines its own symbols and expects to find other symbols. When compiling one file of many, it is likely that a symbol is defined in another file (e.g. a function may be declared in one file and used in another). At the point of compiling the C file into an object file, it is not (yet) an error for a symbol to be missing in the object file; in fact, it is expected. When the linker attempts to compile a complete executable, it must resolve all symbols into addresses, and there is a symbol that is not in an object file or a library, then it becomes an *undefined reference* error, and will generate an error message similar to:

```
2 $ gcc -o main main.o two.o one.o
   two.o: In function 'b':
   two.c:(.text+0xa): undefined reference to 'aa'
4 collect2: ld returned 1 exit status
```

The linker provides some information to resolve the problem, namely that in the object file, `two.o`, there was a reference to a function named `aa` that occurred in the definition of a function named `b`. The object file doesn't report a line number, but it does report where the error occurred. The error was at `.text+0xa`, which means that the symbol was referenced by an instruction in the object code that was 10 (0xa) bytes away from the start of the code for function `b`.

These errors are quite common, and often quite easy to resolve. The common causes for an unresolved reference include:

- A typographic error in the source when making the function call, e.g. typing `prinft` when it should be `printf`
- A result of changing a function name where it was declared but not where it was used
- The function was defined in an `#ifdef`, and the conditions were not met
- An object file was not included when the linker was invoked
- A library was not included or available when the linker was invoked (e.g. `sin` function needs the math library)
- A library was revised and the new library does not provide the same functions

Listing A.64: Simple C with run-time values

```
2 static int NUM_VALUES = 8;
3 static int VALUES[8] = { 0, 1, 2, 3, 4, 5, 6, 7 };
4
5 int main(int argc, char **argv)
6 {
7     printf("Hello World\n");
8     ...
9 }
```

A.14.3 Memory Organization

The linker has the responsibility to create an executable file that contains the instructions needed by the operating system, device programmer, or boot loader to initialize memory and ensure that various program elements are placed at the proper addresses. For example, on a typical Linux machine, a program is stored as an Executable and Linker File (ELF). The ELF file provides information that is used by the operating system to create the virtual memory spaces needed by the program, including the starting address, the length, and any permissions that should be applied (e.g. read-only, executable). The OS allocates the memory, and then loads each chunk of the ELF file into its designated memory section. Each of these ranges is called a section. It is ultimately the Linker's job to determine the starting address and size for each of these sections.

Listing A.65: Assembly translation of Listing A.64

```
1 .section data
2 NUM_VALUES: dw 8
3 VALUES: dw 0, 1, 2, 3, 4, 5, 6, 7
4
5 .section rodata
6 LC1: .string "Hello World\n"
7
8 .section text
9 main:
10     ...
11     ...
```

Listing A.64 shows the beginning of a C program that defines a global value, `NUM_VALUES`, an array named `VALUES`. There is also a string, `Hello World` that appears in the C code. These three values must be stored in the executable so that they can be used when the program runs. The Assembler's mechanism for storing data is to use its data definition statements as shown in Figure A.65. What is interesting in this example is the use of the `.section` directive to the assembler. Three different sections are shown here, `.data`, `.rodata`, and `.text`. The section information will be written out to the object file and used by the linker when creating the executable. When the executable is run, it will copy these values from the executable

Listing A.66: Partial Linker Script

```

2 MEMORY
3 {
4     kseg0_program_mem      (rx)  : ORIGIN = 0x9D000000, LENGTH = 0x80000
5     kseg1_data_mem         (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x8000
6     sfrs                   : ORIGIN = 0xBF800000, LENGTH = 0x100000
7     configsfrs             : ORIGIN = 0xBFC02FF0, LENGTH = 0x10
8 }
9
10 /* *****
11  * Configuration-word sections. Map the config-pragma input sections to
12  * absolute-address output sections.
13  * ***** */
14 SECTIONS
15 {
16     .text :
17     {
18         . = ALIGN(4) ;
19     } >kseg0_program_mem
20     /* Global-namespace object initialization */
21     .init :
22     {
23         KEEP (*crti.o(.init))
24         KEEP (*crtbegin.o(.init))
25         KEEP (*crtend.o(.init))
26         KEEP (*crtn.o(.init))
27         . = ALIGN(4) ;
28     } >kseg0_program_mem
29 }

```

into the memory address where each section begins. The code that uses these symbols will use offset addressing from the beginning of each segment. This is the basis for the mechanism that allows relocatable code. Normally, the compiler, assembler, and linker take care of the care and feeding of the memory sections. But, there are times when the programmer needs to take over.

Linker Script Embedded systems are notorious for having exotic memory address ranges, that are often small, and disjointed. For example, the PIC32 processor used in this course has many different memory address spaces of various sizes and for various purposes. A *linker script* is used to control what sections are understood by the linker, what memory addresses are available, and the mapping of sections into memory.

A partial linker script for the PIC32 processor is shown in Listing A.66. Much of the actual script has been omitted for brevity, but the important aspect is the `MEMORY` and `SECTIONS` configuration values. The `MEMORY` section describes regions of the device's memory that *can* be used. This section associates a name with its configuration information which includes how it can be used, where it starts, and how long it is. For example, this script shows that there is a chunk of memory named `kseg0_program_mem` that can be used for both *reading* and *execution*, but cannot be used for writing. It starts at address `0x9D000000`, and is `0x800000` bytes long.

The next part of the configuration script is the `SECTIONS` directive. This directive allows one or more memory sections to be defined along with any special instructions for the linker. Finally, each section is mapped into a memory range. In this example, the `.text` section will be word-aligned, and is the first section to be placed into the `kseg0_program_mem` section. Its first byte will be at address `0x9D000000` (the start of the memory block). The linker will accumulate all of the objects that are marked (as in Listing A.65) for the `.text` section, and will write them into the executable file as a string of bytes. Then, it will do the same thing for the objects that are marked for the `.init` section, and will append those bytes after the `.text` section. The result is that the executable file will include the instructions to whatever will run the program to load some number of bytes into the memory address that starts at `0x9D000000`. Notice that the loader does not pay attention to *what* is being loaded, it will simply copy the raw bytes from the executable file to that memory address.

In the case of a micro-controller, the executable image written by the linker is typically read by a device called a *programmer*, which can connect to the micro-controller through a programming/debugging port, and will use the contents of the executable to directly write the executable image to the chip. In this way, the programmer is taking the place of the OS executive. Generally, the executable that is used for this process must be static with no run-time libraries; which makes some sense - the image is being written directly to the device's memory, and the device will be ready to run that code whenever it is reset.

A.14.4 Heap

There are three areas of memory that are absolutely required: the text segment, which holds the program's instructions, the stack segment, which is used for automatic variables and function calls, and the data segment that holds the program's literal values.

Programmers who are familiar with desktop programming often start looking for a fourth chunk of memory, the *heap*. The heap is an area of memory that is reserved for dynamic memory allocations through such functions as `malloc` and `free`. The availability of a heap is relatively new. Until very recently, there just wasn't enough RAM in a micro-controller to have a heap. Even now, memory sizes in an MCU are

so small as to make dynamic memory allocation is relatively bad idea. One of the primary reasons for using dynamic memory is to support unbounded growth at run-time. This behavior is insupportable for all but the smallest problems; and typically leads to project failure.

If a heap is still desired, then it must be made available through a linker script. There must be space allocated to the heap so that space can be reserved for the heap control structures. The micro-controller development environments that support a heap each have their own mechanisms for configuring the size and placement of the heap. For example, in the Microchip tools, there is a project option for including a heap at all, and if it is included to specify a size.

Programmers who insist on using `malloc` and `free`, need to adjust how they use these functions. Although it is bad practice in desktop applications to ignore the return value from `malloc`, it is unacceptable for an MCU. In a desktop machine with many gigabytes of RAM, `malloc` does not typically fail. In an MCU with a few kilobytes of memory, `malloc` should be expected to fail, and its return value should *always* be checked against `NULL`. Further, in a desktop application when `malloc` fails, the program typically displays an error message and exits. There is nothing to exit to in an MCU, so the MCU will either restart or enter an infinite loop until it is manually reset. Either way, this is not desirable behavior. Also, in a desktop machine, calling `free` on a block of memory that was previously freed (a double-free) will typically trigger a memory protection violation and a nice error message will be displayed. A double free in an MCU will usually cause undefined behavior, which is never good. Finally, a desktop application that does not free memory creates a *memory leak*. As the application continues to run, memory continues to leak, and the application uses more and more memory. This is not a problem on a system with many gigabytes of memory. On an MCU, there is often barely enough memory for the application, and certainly never enough to waste it on a memory leak. So, the programmer must be completely *clean* in their use of dynamic memory and ensure that there is never a possibility for a leak in a program that should be expected to run forever.

A.14.5 Conclusion

Programming linker scripts is like learning a whole new programming language, and certainly beyond the scope of this text. This section tried to provide a brief overview of how a program goes from source code to a final executable image; and ultimately how that executable image is made to run on an OS or target device.

There are times when it is beneficial to force a symbol, such as a function or data element to be at a particular address. Of course, it is then the programmer's responsibility to ensure that that address is valid and that nothing else is using that address. Micro-controller development environments typically provide some functionality for managing the linker script without actually having to manually edit the linker script; others do not. The basic information in this section should be enough to get started by knowing that there are such things as linker script, and to understand the basic terminology found in the linker's user manual.

Exercises

Exercise A.1 (5)(§A.1) Table A.2 shows four different bases, and an example of each base. There is something in common between each of these values. Determine what it is.

Exercise A.2 (5)(§A.1) There is an error relating to signedness in Listing A.1. Fix it.

Exercise A.3 (5)(§A.1) Show the encoding in each of the four radix values for the following decimal integers:

1. 24
2. 8191
3. 200
4. 255

Exercise A.4 (8)(§A.1) The `const` keyword qualifies the value immediately to its right. The `strcpy` function shown in Listing A.4 modifies the pointers. Rewrite the function so that it is compatible with the following method signature: `void strcpy(char * const dst, const char * const src)`.

Exercise A.5 (8)(§A.3) Prove that bitwise AND of any two different powers of two will always result in 0.

Exercise A.6 (5)(§A.3) The code in Listing A.11 will produce a list of 16 output values that follow what should be a *very* familiar series. Name that series. Rewrite this code to use exactly one integer and one for loop but produce the same output.

Exercise A.7 (5)(§A.3) Write a C language function that will, given an integer, return the value reduced to its nearest even integer. For example, given the value 3, it

will return 2, but given 8, will return 8. Your answer must use AND masking. For an extra challenge, see if you can write the code without using an if statement.

Exercise A.8 (5)(§A.3) Suppose a flag variable has the value `0xA7f3`. Which bits (identified by number) are enabled in this flag?

Exercise A.9 (5)(§A.3) Show a C language function that will, given an integer argument, determine whether bits 18, 7, and 4 are enabled and if it is, it will return 1, else return 0.

Exercise A.10 (5)(§A.3) Show what the C compiler will use after *constant folding* on `(1 << 29) | (1 << 18) | (3 << 4)`

Exercise A.11 (5)(§A.5) What is the greatest value that will fit in each of the three fields of the `rgb565_t` field of Listing A.17? What value would actually be stored if we tried to put the value 0235 into the green field?

Exercise A.12 (5)(§A.5) The memory address for the PORTC register is defined by the micro-controller's architecture manual. PORTC *always* exists at the memory address shown in Listing A.19. Modify the declaration of PORTC so that the value it is a constant pointer (the address it points to cannot change, but the contents there can).

Exercise A.13 (5)(§A.5) Binary Coded Decimal (BCD) is still a widely used method of storing values. In packed BCD, each digit of a string of numbers is encoded using 4 bits. The following table shows the encodings.

0	0b0000	5	0b0101
1	0b0001	6	0b0110
2	0b0010	7	0b0111
3	0b0011	8	0b1001
4	0b0100	9	0b1001

Show the C definition of a data-type that will store 8 BCD digits using the least

amount of memory. *Hint: it can accomplished with 4 bytes.*

Exercise A.14 (5)(§A.5) Suppose we have 6 states (`STATE0 ... STATE5`). How many bits are needed to store the states? With that number of bits, how many unique values *could* be represented? What percentage of the *possible range* is being used? Show the C data structure, called `state_t` to store these values. What would `sizeof(state_t)` return?

Exercise A.15 (5)(§A.7) Listing A.19 showed the definition of a bit-field for a special function register. There are many times when we may want to reference the individual bits of a register, or we want to get the whole value of the register as an integer. Write the C language definition for an SFR that is a union of the bit-field from the listing and the value of the register as an integer. Give an example of reading and writing to the various fields of the union.

Exercise A.16 (5)(§A.8) Listing A.32 checked the bounds on the `age` variable. What would be checked if the dereference operators were omitted?

Exercise A.17 (5)(§A.8) Write a C language function that will convert the code in Listing A.14 to a string and return it.

Exercise A.18 (5)(§A.8) Listing A.1 showed an implementation of the `swap` function that returned a copy of the given value. While working on a project, you are instructed to re-write the function to use a pointer to change the value in-place: `void swap_inplace(unsigned int *v)`. Provide a *safe* implementation of this function.

Exercise A.19 (5)(§A.8) Write a C language function called `inrange` that will take a pointer called `start`, a length (in bytes), and a pointer called `inside`. The function will return 1 if `inside` is between `start` and `start + length`, and 0 if it is outside the range.

Exercise A.20 (5)(§A.10) Write a defined symbol called `BIG2LTL` that takes a sin-

gle identifier, assumed to be an integer, and converts it from big to little endian.

Exercise A.21 (5)(§A.10) Write a defined macro that takes three arguments representing an integer, a start, and an end. The macro returns the bits of the integer in the range $\{start \dots end\}$. For full credit, this can be done as a single-line macro expansion involving two shift operations and some extra arithmetic. Hint: remember how C decides to use shift-right-logical vs. shift-right-arithmetic

Exercise A.22 (5)(§A.10) Describe the defined macro evaluation for the code shown in Listing A.52

Listing A.52: Listing for Exercise 22

```
1 #include <stdio.h>
3 #define F(x,y) { \
4     int __tmp; \
5     __tmp = x; \
6     x = y; \
7     y = __tmp; \
8 }
9
11 int main( )
12 {
13     int a = 5;
14     int b = 8;
15
16     F(a,b)
17     printf("%d %d\n", a,b);
18 }
```

Exercise A.23 (5)(§A.10) Compare and contrast the use of defined symbols, especially to replace functions in C code.

Exercise A.24 (5)(§A.12) In the In-line Assembly section, it was stated that the Assembly code will be placed directly in the Assembly file generated by the C compiler. Review Listing A.60. Line 21 does not contain any Assembly code, but it is *very* important to the correct functioning of this example. First, what is `sum2`, and how is it referenced in the Assembly code? Why is it needed in this example? Why isn't there any code there?

Exercise A.25 (10)(§A.12) Some models of the MIPS processor have a special instruction for working with the bit-fields. The `INS` instruction takes four arguments:

rt the destination register

rs the source register

start an immediate, the start bit (least)

size an immediate, the number of bits

The instruction:

INS	\$t0, \$t1, 4, 5
-----	------------------

Will copy the bits 4-8 (the 5-bits starting at 4) from \$t1 to bits 4-8 in \$t0, leaving all of the other bits unchanged.

But, not all machines have this feature. There is a pre-processor defined symbol, `HAS_EXTENDED_MIPS` that will be defined if the instruction is present, and will be undefined otherwise.

Write a C function: `uint32 copy_bits(uint32_t dest, uint32_t src)` that will copy bits 5-12 from the src the dest, and return the new value. Your code must use in-line assembly if the instruction is available, or normal C code otherwise, you will need to write this C code.

Exercise A.26 (15)(§A.12) The previous exercise asked you to write an Assembly wrapper around the `INS` instruction. This was less effective because the Assembly code *requires* the position and size to be *literals*. This can be solved with the pre-processor. Make a `#define` symbol `COPY_BITS(DST, SRC, SIZE, POS)` that uses the `INS` instruction or not as before, but works with `SIZE` and `POS` bits.

Bibliography

- [1] Constant folding. URL http://en.wikipedia.org/w/index.php?title=Constant_folding&oldid=609868991. Page Version ID: 609868991.
- [2] Intel MCS-51. URL http://en.wikipedia.org/w/index.php?title=Intel_MCS-51&oldid=617021042. Page Version ID: 617021042.
- [3] Streaming SIMD Extensions. URL http://en.wikipedia.org/w/index.php?title=Streaming_SIMD_Extensions&oldid=612663437. Page Version ID: 612663437.
- [4] Edsger W. Dijkstra. Letters to the Editor: Go to Statement Considered Harmful. 11(3):147–148. ISSN 0001-0782. doi: 10.1145/362929.362947. URL <http://doi.acm.org/10.1145/362929.362947>.
- [5] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 5 edition edition. ISBN 978-0-12-383872-8.
- [6] Yves-Marie Morgan. Entry to 22nd IOCCC. URL <http://www.ioccc.org/2013/morgan2/hint.html>.
- [7] S. Sandeep. GCC Inline Assembly Howto. URL <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>.

Index

Analog-to-Digital Controller, 54

Architecture

Address Space, 148, 194

Arithmetic, 124

Bypassing, 124

Caching, 150

Control Unit, 123

Fetch-Decode-Execute, 123

Harvard, 35, 36

Load-Store, 126

Memory Regions, 151

Memory-Mapped I/O, 194

MMU, 148

Pipeline, 123

Register-Memory, 126

Registers, 126, 127

Stalls, 124

TLB, 148

Von Neumann, 35

binary coded decimal, 33

bit-banging, 54

Bus

Dual-Role Controller, 120

master, 118

slave, 118

bus steering, 120

byte, 33

C

In-Line Assembly, 181

Pragma, 162

capacitatively couple, 77

Clock, 89

Clocks

LPRC, 155

Primary, 155

RC, 154

Secondary, 155

Divider, 100

Drift, 93

Jitter, 92

Multiplier, 100

Overshoot and Undershoot, 92

PLL, 101

Skew, 93

Slew, 90

Type

Logic, 99

RC, 94

Resonator, 95

clock domains, 121

Clocks

Divider, 157

- Multiplier, 157
- PLL, 156
- Selection Mux, 158
- versus Power, 159
- code density, 46
- combinational logic, 29
- comparator, 55
- cycles per instruction, 46
- Direct Memory Access, 120
- display driver, 33
- DMA, *see also* Direct Memory Access
- dot-product, 125
- Execute In Place, 49
- Execute-In-Place, 36
- Finite State Machine, 295
- General Purpose I/O, 193
- glue logic, 30
- GPIO, *see* General Purpose I/O 193
- i8051, 39
- instruction sets, 117
- Interrupt, 279
 - FSM, 295
 - Interrupt Vector Table, 299
 - ISR, 281, 289
 - Nested, 286
 - Shadow Registers, 300
- Interrupt Controller, 41
- Interrupt Request, 41
- interrupt service routine, 41
- Jellybean Logic, 28
- logic gate, 28
- look-up-table, 33
- Low-Power Mode, 42
- magnetic field, 380
- Memory Management Unit, 71
- memory mapped I/O, 120
- Micro-Controller
 - 8051, 126
 - i8051, 39
 - MIPS, *see also* MIPS32
 - PIC32, *see* PIC32
 - TI MSP430, 55
 - TMS1000, 30
- Micro-controller
 - TMS1000, 36
- MIPS32, 115, 123
 - Architecture, 123
 - Arithmetic Instructions, 138
 - Bit Instructions, 138
 - Branch Instructions, 142
 - Calling Conventions, 127
 - Cause Register, 134
 - Co-Processor0, 129
 - CP0, 129, 300
 - HI, 128
 - INS and EXT, 138
 - Instructions, 136
 - Interrupt, 299
 - LO, 128
 - Load Store Instructions, 145

- Logic Instructions, 139
- Machine Control Instructions, 143
- MDU, 124
- MIPS16e, 123
- Multiplier, 124
- PC, 128
- Registers, 126, 127
- Shadow Registers, 300
- Special Registers, 128
- SRSCtl, 301
- Trap Instructions, 143
- Moore's Law, 47
- motor, 353
 - armature, 380
 - brush, 380
 - characteristic torque, 356
 - commutator, 380
 - housing, 380
 - linear velocity, 358
 - speed constant, 358
 - speed vs. torque, 361
 - stator, 380
 - torque, 354
 - torque formula, 354
 - two-pole, 380
 - windings, 380
- nibble, 33
- parasitics, 77
- PIC32, 115, 118, Bus121
 - Block Diagram, 119
 - Bus Matrix, 121, 122
- Clock
 - Fail-Safe Monitor, 158
- Clock Manager, 122
- Clocks, 121, 122, 148, 153, 155, 156, 158, 159, 164, 173, 174, 188–190
 - Inputs, 153
- Configuration, 161
- Configuration Options, 163
- Control Unit, 123
- Core, 118
- Core Timer, 129
- Errors
 - Cannot Program, 174
 - Clock, 171
 - Configuration, 171
 - Handlers, 182
 - ICD Connection, 173
 - Reboots, 175
- Feature Comparison, 118
- Flash Memory, 121
- ICD, 118
- Interrupt, 299
- Intrinsics, 146
- Memory, 148
 - Address Space, 149
- Memory Map, 151, 152
- MMU, 148
- Multiplier, 124
- Multiply-Accumulate, 125
- PBCLK, 121, 122, 148, 155, 156, 158, 159, 164, 173, 188–190

- Peripheral Bridge, 121
- Physical / Virtual Mapping, 149
- PIC32MX1xx, 117, 118
- PIC32MX3xx, 118
- PIC32MX4xx, 118
- PIC32MX7xx, 117, 118
- PIC32MZ, 117
- Pragma, 162
- Pre-Fetch Module, 121
- SYSCLK, 121, 122, 148, 156, 158, 159,
164, 173, 174, 188–190
- register, 37
- sequential logic, 29
- seven-segment display, 33
- signal integrity, 77
- Silicon Graphics
 - Onyx, 116
- thru-hole, 77
- TI MSP430, 55
- TMS 10000, 30
- Universal Asynchronous Receiver/Trans-
mitter, 41
- Very-Large Scale Integrated, 47
- VLSI, 47