5.4 SUPPORTED DATA TYPES AND VARIABLES

5.4.1 Identifiers

A C variable identifier (the following is also true for function identifiers) is a sequence of letters and digits, where the underscore character "_" counts as a letter. Identifiers cannot start with a digit. Although they can start with an underscore, such identifiers are reserved for the compiler's use and should not be defined by your programs. Such is not the case for assembly domain identifiers, which often begin with an underscore, see **Section 5.12.3.1 "Equivalent Assembly Symbols"**.

Identifiers are case sensitive, so `main` is different to `Main`.

Not every character is significant in an identifier. The maximum number of significant characters can be set using an option, see **Section 4.8.8 "-N: Identifier Length"**. If two identifiers differ only after the maximum number of significant characters, then the compiler will consider them to be the same symbol.

5.4.2 Integer Data Types

The MPLAB XC8 compiler supports integer data types with 1, 2, 3 and 4 byte sizes as well as a single bit type. Table 5-1 shows the data types and their corresponding size and arithmetic type. The default type for each type is underlined.

TABLE 5-1: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
`bit`	1	Unsigned integer
`signed char`	8	Signed integer
`unsigned char`	8	Unsigned integer
`signed short`	16	Signed integer
`unsigned short`	16	Unsigned integer
`signed int`	16	Signed integer
`unsigned int`	16	Unsigned integer
`signed short long`	24	Signed integer
`unsigned short long`	24	Unsigned integer
`signed long`	32	Signed integer
`unsigned long`	32	Unsigned integer
`signed long long`	32	Signed integer
`unsigned long long`	32	Unsigned integer

The `bit` and `short long` types are non-standard types available in this implementation. The `long long` types are C99 Standard types, but this implementation limits their size to only 32 bits.

All integer values are represented in little endian format with the Least Significant bit (LSb) at the lower address.

If no signedness is specified in the type, then the type will be `signed` except for the `char` types which are always `unsigned`. The `bit` type is always unsigned and the concept of a signed bit is meaningless.

Signed values are stored as a two's complement integer value.

The range of values capable of being held by these types is summarized in Table 5-2 The symbols in this table are preprocessor macros which are available after including `<limits.h>` in your source code.

As the size of data types are not fully specified by the ANSI Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

The macros associated with the `short long` type are non-standard macros available in this implementation; those associated with the `long long` types are defined by the C99 Standard.

TABLE 5-2: RANGES OF INTEGER TYPE VALUES

Symbol	Meaning	Value
CHAR_BIT	bits per `char`	8
CHAR_MAX	max. value of a `char`	127
CHAR_MIN	min. value of a `char`	-128
SCHAR_MAX	max. value of a `signed char`	127
SCHAR_MIN	min. value of a `signed char`	-128
UCHAR_MAX	max. value of an `unsigned char`	255
SHRT_MAX	max. value of a `short`	32767
SHRT_MIN	min. value of a `short`	-32768
USHRT_MAX	max. value of an `unsigned short`	65535
INT_MAX	max. value of an `int`	32767
INT_MIN	min. value of a `int`	-32768
UINT_MAX	max. value of an `unsigned int`	65535
SHRTLONG_MAX	max. value of a `short long`	8388607
SHRTLONG_MIN	min. value of a `short long`	-8388608
USHRTLONG_MAX	max. value of an `unsigned short long`	16777215
LONG_MAX	max. value of a `long`	2147483647
LONG_MIN	min. value of a `long`	-2147483648
ULONG_MAX	max. value of an `unsigned long`	4294967295
LLONG_MAX	max. value of a `long long`	2147483647
LLONG_MIN	min. value of a `long long`	-2147483648
ULLONG_MAX	max. value of an `unsigned long long`	4294967295

Macros are also available in `<stdint.h>` which define values associated with fixed-width types.

When specifying a `signed` or `unsigned` `short int`, `short long int`, `long int` or `long long int` type, the keyword `int` can be omitted. Thus a variable declared as `short` will contain a `signed short int` and a variable declared as `unsigned short` will contain an `unsigned short int`.

It is a common misconception that the C `char` types are intended purely for ASCII character manipulation. However, the C language makes no guarantee that the default character representation is even ASCII. (This implementation does use ASCII as the character representation.)

The `char` types are the smallest of the multi-bit integer sizes, and behave in all respects like integers. The reason for the name "char" is historical and does not mean that `char` can only be used to represent characters. It is possible to freely mix `char` values with values of other types in C expressions. With the MPLAB XC8 C Compiler, the `char` types are used for a number of purposes – as 8-bit integers, as storage for ASCII characters, and for access to I/O locations.

5.4.2.1 BIT DATA TYPES AND VARIABLES

The MPLAB XC8 C Compiler supports `bit` integral types which can hold the values 0 or 1. Single `bit` variables can be declared using the keyword `bit` (or `__bit`), for example:

```
bit init_flag;
```

You can also use the These variables cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function. For example:

```
int func(void) {
    static bit flame_on;
    // ...
}
```

A function can return a `bit` object by using the `bit` keyword in the function's prototype in the usual way. The 1 or 0 value will be returned in the carry flag in the STATUS register.

The `bit` variables behave in most respects like normal `unsigned char` variables, but they can only contain the values 0 and 1, and therefore provide a convenient and efficient method of storing flags. Eight bit objects are packed into each byte of memory storage, so they don't consume large amounts of internal RAM.

Operations on `bit` objects are performed using the single bit instructions (`bsf` and `bcf`) wherever possible, thus the generated code to access `bit` objects is very efficient.

It is not possible to declare a pointer to `bit` types or assign the address of a `bit` object to any pointer. Nor is it possible to statically initialize `bit` variables so they must be assigned any non-zero starting value (i.e., 1) in the code itself. Bit objects will be cleared on startup, unless the bit is qualified `persistent`.

When assigning a larger integral type to a `bit` variable, only the LSb is used. For example, if the `bit` variable `bitvar` was assigned as in the following:

```
int data = 0x54;
bit bitvar;
bitvar = data;
```

it will be cleared by the assignment since the LSb of `data` is zero. This sets the `bit` type apart from the C99 Standard `__Bool`, which is a boolean type, not a 1-bit wide integer. The `__Bool` type is not supported on the MPLAB XC8 compiler. If you want to set a bit variable to be 0 or 1 depending on whether the larger integral type is zero (false) or non-zero (true), use the form:

```
bitvar = (data != 0);
```

The psects in which `bit` objects are allocated storage are declared using the `bit PSECT` directive flag, see **Section 6.4.9.3 "PSECT"**. All addresses assigned to bit objects and psects will be bit addresses. For absolute bit variables (see **Section 5.5.4 "Absolute Variables"**), the address specified in code must be a bit address. Take care when comparing these addresses to byte addresses used by all other variables.

If the `xc8` flag `--STRICT` is used, the `bit` keyword becomes unavailable, but you can use the `__bit` keyword.

5.4.3 Floating-Point Data Types

The MPLAB XC8 compiler supports 24- and 32-bit floating-point types. Floating point is implemented using either a IEEE 754 32-bit format, or a modified (truncated) 24-bit form of this. Table 5-3 shows the data types and their corresponding size and arithmetic type.

TABLE 5-3: FLOATING-POINT DATA TYPES

Type	Size (bits)	Arithmetic Type
`float`	24 or 32	Real
`double`	24 or 32	Real
`long double`	same as `double`	Real

For both `float` and `double` values, the 24-bit format is the default. The options `--FLOAT=24` and `--DOUBLE=24` can also be used to specify this explicitly. The 32-bit format is used for `double` values if the `--DOUBLE=32` option is used and for `float` values if `--FLOAT=32` is used.

Variables can be declared using the `float` and `double` keywords, respectively, to hold values of these types. Floating-point types are always signed and the `unsigned` keyword is illegal when specifying a floating-point type. Types declared as `long double` will use the same format as types declared as `double`. All floating-point values are represented in little endian format with the LSb at the lower address.

This format is described in Table 5-4, where:

- Sign is the sign bit which indicates if the number is positive or negative
- The exponent is 8 bits which is stored as excess 127 (i.e., an exponent of 0 is stored as 127).
- Mantissa is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign} \times 2^{(exponent-127)} \times 1.$ mantissa.

TABLE 5-4: FLOATING-POINT FORMATS

Format	Sign	Biased exponent	Mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
modified IEEE 754 24-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

Here are some examples of the IEEE 754 32-bit formats shown in Table 5-5. Note that the Most Significant Bit (MSb) of the mantissa column (i.e., the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

TABLE 5-5: FLOATING-POINT FORMAT EXAMPLE IEEE 754

Format	Number	Biased exponent	1.mantissa	Decimal
32-bit	7DA6B69Bh	11111011b	1.0100110101101101 0011011b	2.77000e+37
		(251)	(1.302447676659)	—
24-bit	42123Ah	10000100b	1.001001000111010b	36.557
		(132)	(1.142395019531)	—

Use the following process to manually calculate the 32-bit example in Table 5-5.