Chapter I2
Using Turbo C

Revised: Jan. 14, 2011

I2.1 ABOUT TURBO AND BORLAND C /C++

The discussions of MPX-PC in this manual generally assume that a compiler in the "Turbo C" family will be used for development. These compilers were produced by Borland International. Although it may be possible to develop the project using other C compilers, and the Turbo C compilers are old, there are at least two important reasons for the choice of Turbo C:

1. Turbo C provides low-level mechanisms for hardware access, making it unnecessary to resort to assembly language.

2. An appropriate version of Turbo C is available online for free.

There are many versions of the compiler we call Turbo C. In particular:

- Older versions were designed for C alone; newer versions support also C++.
- Some versions run under MS-DOS and produce programs for MS-DOS; other versions run under Windows and produce programs for Windows.
- The older and simpler versions were called Turbo C or Turbo C/C++; newer versions are called Borland C++.

In this manual, unless otherwise stated, we will continue to use the generic name "Turbo C" to refer to all versions.

This manual will assume the use of Turbo C/C++ version 2.01. Although this compiler has a very primitive IDE compared to such modern tools as Eclipse or Visual Basic, it is available and will do the job for MPX. If you have a different version of the compiler, be *sure* that it will generate 16-bit (MSDOS) executables.

I2.2 WHY C?

MPX-PC is designed for development using the C language as defined by the official standard published by the American National Standards Institute (ANSI), designated X3.158-1989. This is commonly known as ANSI C. There are a number of reasons for this choice:

1. There is a well-defined standard. The language is mature enough that the standard is widely accepted.

2. ANSI C compilers are available (and inexpensive) for almost any type of computing system. Thus, versions of MPX for other systems can use a consistent language.

3. C is an effective *System Implementation Language*. It includes features designed for efficiency and low-level control even at the cost of some safety and abstraction features. This is important for operating systems programming.

Languages which could be considered as reasonable alternatives to C include Ada, C++. and Java. Ada is well standardized and includes both good abstraction and important system implementation features, but is not widely known and compilers are often unavailable. C++ includes all features of ANSI C, since it is a superset of C. However, the object-oriented features of C++ lead to greater abstraction and less efficient implementation, which makes it less suitable as a system implementation language. A similar statement applies to Java, which has even more serious performance issues.

I2.3 THE INTEGRATED DEVELOPMENT ENVIRONMENT

This section provides a quick overview of the capabilities of the Turbo C software packages, and discusses some aspects that are of specific importance for MPX. It is not our purpose to provide a detailed description of all of the features of Turbo C; these details can be found in the documentation supplied for your particular version.

The task of producing an executable PC program using Turbo C ultimately includes the same processing steps that occur with almost any type of program:

1. Develop the program in the form of a C-language source file, or collection of source files, using a suitable editor;

2. Process each source file with a compiler, producing a set of object files;

3. Process the object files with a linker to produce the final executable file.

In the PC environment, C source files are distinguished by the name extension `.C`; object files have the extension `.OBJ`, and relocatable executable files, the type we will deal with, have the extension `.EXE`.

This sequence, of course, describes only the technical processing steps in program development; we have not mentioned the steps in analysis, design, maintenance, etc. that are all part of the total software engineering process.

In many computing environments, the steps of editing, compiling, and linking are performed by separate programs, each invoked by a command at the system interface. Most Turbo C versions support this traditional approach to compiling. For this purpose two programs are provided: a command-driven compiler, typically called `tcc`, and a linker, typically called `tlink`. To use the traditional approach, a programmer first develops source files using any desired editor (one possible choice is the editor included in the Turbo C IDE, discussed below). These source files are then processed by `tcc` to produce object files. Finally, `tlink` is invoked to produce the executable file. Any desired options and parameters must be specified on the `tcc` and `tlink` command lines. Full details of this process are given in the Turbo C documentation.

Turbo C, however, also provides a (relatively) more powerful and interactive alternative to command-oriented program development; this alternative is called the **Integrated Development Environment,** or **IDE.** If you are familiar with IDEs such as Visual Studio or Eclipse, you may be disappointed in the limitations of the Turbo C IDE. However, it does have important advantages over command line compiling. Because of its relatively attractive features, we will assume in the rest of this discussion that you will be using the IDE.

The Turbo C IDE is a unified, interactive environment that presents a menu-driven Graphical User Interface (GUI) to the user. The exact form of this GUI varies for different Turbo C versions, but the principles are the same. Working within this environment, the user can edit multiple source files; compile; link; run the resulting program; and monitor execution using a source-level interactive debugger.

One of the most powerful features of the Turbo C IDE is its ability to organize all of the files related to your program into an entity called a **project**. This facility is especially valuable when your program involves a number of source files, as will certainly be the case for MPX. Because Turbo C tracks all related files and knows what forms exist (source, object, executable) and when they were created, it can automatically reprocess as necessary. This forms the basis for a powerful "make" command which performs all necessary recompilation and linking, and only that, before execution. The IDE also understands included header files; if any file referenced by an `#include` statement is changed, all other files that include it will be recompiled.

Turbo C maintains the information describing the contents and status of a project in a **project file** which has the extension `.PRJ`. The format of these files has varied considerably over different versions, and in general one version of Turbo C cannot read a project file created by a different version. In the earliest versions, project files were text files that could be created and modified by a standard editor. Later versions have used non-text formats that can only be created or modified by the IDE.

I2.4 TURBO C CONFIGURATION

Turbo C allows the user to specify a large number of options that control its overall operation. These options determine everything from the type of descriptive map produced by the linker to the special ways the editor may interpret a TAB key. In the IDE the options are selected

-

from the *Options* menu, which will have various submenus. Many of them may also be specified on the command line using `tcc`. The exact set of options available varies across different versions, but the options important for MPX are available in all versions we have examined.

Turbo C saves the currently-selected options in a configuration file, which usually has the standard name `TCCONFIG.TC`. By default this file resides in the Turbo C directory, and options selected during one session affect all future sessions. It is also possible to place a configuration file in a local directory, which affects only projects run from that directory. In general, configuration files produced by one version cannot be used by another.

Borland C++ supports a staggering assortment of options. These are either recorded in configuration files, or in *Style Sheets* associated with individual output files. Users of this compiler should note that we are concerned only with options for generating 16-bit DOS applications.

Some of the Turbo C options affect the behavior of the compiler or linker. It is essential for MPX that several of these options be set in a particular way. The reason for most of these has to do with details of the PC architecture, to be discussed in Chapters I3 and I4. For now, we will just list the important settings without explanation. These are as follows:

MEMORY MODEL: *large*.

STANDARD STACK FRAME: *on*.

TEST STACK OVERFLOW: *off*.

ASSUME SS EQUALS DS: *off* (or *never*).

TURBO C (or C++) KEYWORDS: *selected*.

CASE SENSITIVE LINK: *off*.

The choice made for most other options is not critical to MPX. Of course, you should always select options appropriate to your environment; do not select code generation for a floating-point coprocessor if you do not have one! In most cases, it is best to rely on the default option settings unless you thoroughly understand the effect of your changes.

I2.5 TURBO C HEADER FILES

Turbo C provides all of the usual header files specified by the ANSI C standard. In addition, there are a number of system-specific header files defined for various purposes in the PC environment, such as low-level access to graphics facilities, I/O routines, etc. All of these header files are fully described in the Turbo C documentation.

A few of these special header files are automatically `#included` by `MPX_SUPT.H` or `MPX_SUPT.C`. The only system-specific header file you should ever `#include` is `dos.h`. This file should be referenced only by modules that actually make use of the low-level faciltities described in Chapter I4.

-

I2.6 NOTES ON THE C LANGUAGE

The discussions in this manual assume that you know the ANSI C language well enough to write simple programs. If you do not, there is a wide selection of texts available for this language. It is important to choose a text that covers ANSI C rather than earlier, unstandardized versions; such a text will have a publication date no earlier than 1990.

The remainder of this chapter provides some hints that may aid you in the most effective use of C in the MPX project.

Pointers and Structures

The implementation of any operating system relies heavily on data structures (control blocks) that represent processes, I/O operations, memory allocation, etc. C allows one to define structures in a very straightforward way. It is also possible to define a data type using the `typedef` statement. When this is done, the type name can be used in place of the C-supplied types like `int` and `char`.

In most cases, there are multiple control blocks of the same type. A given execution of an OS procedure will work with one particular control block representing the I/O, process, etc. being serviced. Referencing a particular control block is done via pointer variables. C allows the declaration of a variable as a pointer to a structure. The pointer then can refer to members of a data structure using the notation *ptr->member*. The following example is typical in the MPX-PC environment:

```
/* define symbolic name ready */
#define READY  1

/* define pcb structure */
typedef struct pcb_s {
    struct pcb *next_pcb;
    int state;
    int priority;
    char name[8];
    int *stack_ptr;
    } pcb;

/* Declare pcb structures */
pcb pcb0;
pcb pcb1;
pcb pcb2;
pcb pcb3;

/* define a pointer to a pcb */
pcb *ptr;
```

The structure name is `pcb`. It consists of five elements:

`next_pcb`, - a pointer to the next pcb in a linked list
`state` - an integer variable
`priority` - an integer variable
`name` - a character string
`stack_ptr` - a pointer to an integer

The following program statements demonstrate the use of pointers in addressing pcb control blocks:

```
/* set ptr to address of pcb2 */
     ptr = &pcb2;

/* set next_pcb pointer of pcb2 to address of pcb3 */
     ptr->next_pcb = &pcb3;

/* copy name into pcb2 */
     strcpy(ptr->name,"process2");

/* set state in pcb2 */
     ptr->state = READY;

/* set priority in pcb2 */
     ptr->priority = 70;
```

By simply using the notation ptr->member we can refer to any member of the structure using a pointer to a structure. Recall that in C, the & in front of a variable name refers to its address, e.g., `&pcb2` above refers to the address of pcb2. Recall also that a character string's address can be referred to using either its name, or using the notation `&name[0]` (remember the first element of a character string (array) has subscript 0). Hence for the example above, `buffer` and `&buffer[0]` both refer to the address of the character string (array) buffer.

Recall also, that if we wish to refer to the contents of a variable pointed to by a pointer, we use the notation **ptr*. Hence, if we have the following declaration:

```
int *number;
```

`number` refers to the address of an integer variable, while `*number` refers to the contents of the integer variable. If we wish to set an address into the pointer `number` we use

```
number = &count;
```

If we want to then set a value into the variable pointed to by `number` (in this case `count`), we use

```
*number = 8;
```

Sooner or later we all get caught by that inadvertent use of **pointer* when we should have used *pointer* and vice versa.

In the above case, we can use `pcb` as a data type. While the declaration differs, the references to members of the `pcb` structure remain in the form *ptr->member*.

In some cases only one data structure may be required. In this case we can refer to the members of the structure using the notation *structname.member*. This is illustrated in the following example:

```
/* define device control block for printer */
     struct {
          int busy;
          char *buffer_add;
          char *next_char;
          int length;
          } prtdcb;


     char buffer[80];
     .
     .
     prtdcb.busy = 0;
     prtdcb.buffer_add = buffer;
     prtdcb.next_char = &buffer[0];
     prtdcb.length = 80;
```

Here we can refer to any element using the notation: *structure.member*. We point out that if you have a pointer to a structure, you may refer to the member of the structure with either the form *ptr->structure.member* or the form *ptr->member*, i.e., you may omit the structure name from the reference. Hence, if `ptr` points to the structure `prtdcb` the following all refer to the member `length` of the structure `prtdcb`:

```
     prtdcb.length = 0;
     ptr->prtdcb.length = 0;
     ptr->length = 0;
```

The style of programming will differ among programmers, but you should be consistent in the way you refer to elements of a structure when using pointers.

-

Pointers and Allocation

When using pointers, you must be careful that each pointer refers to a variable that is indeed *allocated*. A common error (and one which sooner or later will get you) is to declare a pointer to a structure, but never declare the structure, as shown below:

```
/* Example of Error in Allocation */
/* Only a pointer to the structure is being allocated */
struct {
     int busy;
     char *buffer_add;
     char *next_char;
     int length;
     } *prtdcb;
```

If we refer to `prtdcb->length`, without any other declaration of a `prtdcb`, then in Turbo C, `prtdcb` takes on the value 0 and the data structure is located at location 0 of the current data segment (see next chapter), i.e., we have a "wild pointer". Null pointers can show up in (early) testing of the MPX-PC operating system. It is wise to use *defensive programming* in which you validate pointers before you use them to be sure they are never 0.

Pointers to Functions

In C it is possible to create a pointer to a function, and then to call functions indirectly using this pointer. A pointer may point in turn to a number of different functions, provided that they all have the same form for their prototype (that is, the return value type and the number and types of parameters are the same). This facility allows creation of a table of functions, for example, so that calls can select their destination by index from this table.

Suppose that a collection of functions can all be represented by the prototype

```
int func(int a, char b);
```

Then we may declare a pointer to functions of this type by writing

```
int (*fp) (int, char);
```

Assignments may be made to this pointer by statements such as

```
fp = func1;
```

if `func1` is a function in the collection. Finally, if the above assignment has been made, then the following statements are equivalent:

```
val = func1(3, 'a');
val = (*fp) (3,'a');
```

This facility may be particularly useful in managing the command functions for Module R1.

Symbolic Names

We recommend that you define symbolic names for numeric values used throughout your program. The use of numbers that represent device addresses, device status, process status, etc. in the code of Turbo C programs should be avoided. It is difficult to know just what these numbers mean within the program. Assigning symbolic names to these numbers, defining them with the `#define` statement, and referencing them in a C program using the symbolic name is considered good programming practice in the MPX project.

This is perhaps best illustrated with an example. The following shows code that uses absolute numbers:

```
pcb->state = 1;
.
.
pcb->state = 2;
```

Just what is the meaning of the number 1 above? Comments might help us understand the above statements more easily, but consider:

```
#define  READY   1  /* process ready state */
#define  BLOCKED 2  /* process blocked state */
   .
   .
     pcb->state = READY;
   .
   .
     pcb->state = BLOCKED;
```

The use of such symbolic values makes the code far easier to understand and more importantly, simplifies maintenance. Common style conventions for C recommend that symbolic constants of this type use all capital letters for their names.

I2.7 HARDWARE SPECIFIC EXTENSIONS

The elements of Turbo C described so far are sufficient for programming Required Modules R1 and R2. Turbo C also has a number of extensions that are not part of ANSI C but are extremely useful for system programming on the PC architecture. To understand these extensions, we must first learn some details of the PC architecture itself. We will go under the hood of the IBM-PC and its processor, the Intel 8086, in the next chapter. Following that, we will see how Turbo C allows the programmer to fully control this architecture without the need for programming in assembly language.