FTL Modula-2

Fast Interactive Modula-2 Compiler

Z80 User Manual

Published exclusively in Europe by HiSoft

First printing August1987

ISBN 0 948517 07 7

Set using an Apple Macintosh™ and Laserwriter™ with Microsoft Word™.

Table of Contents

Z80 User Guide

FTL Modula-2
Z80 User Manual

1 Introduction

This is the user guide for the Z80 CP/M compiler. It reflects the state of the compiler at version 1.24. If you have a later version, consult the README.NOW file for details of more recent additions. In this manual, you will find detailed descriptions of the installation of the compiler on your computer, of the standard modules that come with this particular version of the compiler, and of the restrictions and extensions to the compiler in this version.

There is a separate user manual for each version of the compiler. The reference manual is common to all versions. To find information about the Modula-2 language (as implemented by FTL), refer to the reference manual. For details of using it on a particular type of machine, consult the user manual for that version.

2 Hardware Requirements

To run this compiler, you will require a Z80 processor running CP/M. A 58k CP/M system is desirable, with a TPA that extends up to D000. If your TPA is less than this, you are likely to be harassed by the `out of memory` error message.

To determine if your TPA is large enough, run the program MEMCHECK. Once you have run this program, you can delete it from your Modula-2 system disk. This program errs on the conservative side, so that, even if it tells you that you do not have enough memory, it may still be possible to run the compiler. You may need to use the compiler flag `/S` which reduces the amount of memory used by the compiler.

Secondly, you will almost certainly need at least two disk drives. It should be possible to run on a single large capacity floppy disk drive - we have used it on a 1 Megabyte double density, double sided eight inch drive. You only need one floppy drive if you have a hard disk or a substantial RAM disk.

The compiler will not run under CP/M86 or CP/M68K, unless your system has a dual processor and can also run Z80 (note - not 8080) code. We hope to have versions for the machines on which these run in the future. An MSDOS version of the compiler is available from HiSoft now. A 68000 version for the Atari ST should be available soon. Other 68000 versions will follow.

To run the editor you will need a terminal which can perform some basic editing functions. You will need a terminal which can perform insert and delete line functions and some form of highlighting, inverse video, or underlining. It is also nice if your terminal can use graphics characters to draw boxes.

If your terminal is unable to delete and insert lines, there is a *dumb* editor for use on dumb terminals. This will work provided that your terminal can position the cursor, but is slower than the standard editor because it must re-draw the screen more often.

If you have a memory mapped video, you can probably modify one of the versions of the `ScreenIO` module supplied with the optional Editor/Toolkit disk to produce a memory mapped version of the editor for your system.

3 Installation

3.1 Amstrad Machines

If you don't have an Amstrad CPC or PCW please skip to **Section 3.2. FTL Modula-2** is a sophisticated piece of software and the cost of its power is that installation isn't as simple as with a BASIC interpreter or a simple Pascal compiler. An hour spent now (hopefully it wouldn't take that long) will soon pay itself back in the lack of frustration as you learn how to use the system to its full rather than wasting your time swapping disks and looking through this manual to work out which keys to press. Above all don't be put off by the fact that you don't understand the precise reasons for setting your disks up as described; after a short time familiarizing yourself with the system the later sections should make sense and we hope you will appreciate the reasoning behind our suggestions.

3.1.1 Getting started on an Amstrad PCW 9512

Naturally it is easier to run **FTL Modula-2** on the PCW9512 than the earlier Amstrad machines with the possible exception of the PCW8512.

1. To make a backup of your distribution disks format a disk in CF2DD format using `DISCKIT` and remove this from drive A; this will be your backup disc.

2. Make sure you are logged into drive A and then for each side of the master disks type

```
WP A: M: -Q
```

This will copy all of the files onto the ramdisk.

Insert your backup disk and type:

```
M:WP M: A: -Q
```

This will copy all the files on this side of the master disk onto your backup.

Now type WD M:

Then, when prompted, type A for all. This removes all the files from the ramdisk so that you have plenty of room for the next side. Repeat this process for all four sides.

Mark your backup disk FTL Modula-2 Backup and, to be safe, make a copy of this entire disk using DISCKIT. This new disk will be your work disk. You can now put the distribution disks and your original backup away safely.

You can now delete the files MEMCHECK.COM, MEDUMB.COM, SETTERM.COM, TERMINAL.DAT and USQ.COM from your work disk as these are not required.

The method of working that we suggest initially is to have all the parts of the system that you need together with the .SYM files output by the compiler on drive M and your source files and .REL files on a real disk in drive A. To set this up we need to run the SETSEARC program whilst logged into drive A. This will ask a number of questions about disk drives, answer @ [ENTER] to all of these. The program will then ask if you want listing on or to change disks during links ; reply N[ENTER] to these.

We have supplied a submit file called PROF8512.SUB which copies the files that you normally need to M (the same file works for both the 8512 and 9512) but to run this we need the CP/M SUBMIT utility, so copy this to your work disk.

Now you can type PROF8512.S from the A> prompt and the files will be copied to drive M. The other files on your work disk are not always needed (they are mainly the source of the libraries and the assembler) so you can delete these if you wish.

The next step is to log on to drive M using M:. From now on we will remain logged into this drive. To compile the LIST.MOD example program type

```
M2 A:LIST.MOD
```

Then, after the compiler has finished,

```
ML A:LIST
```

This will create the file LIST.COM on drive M: and you can run this by typing

```
LIST A:LIST.MOD CON:/O
```

which will list the program on the console.

When you next want to load **FTL Modula-2**, boot normally. It is a good idea to use SETKEYS KEYS.WP as the editor uses many of the same keys as WordStar; so the cursor keys will move the cursor for example. Then log into drive A and type PROF512.S. Finally log onto drive M. You can now try out the editor and start writing your own programs, using for example:

```
ME B:TEST.MOD
```

but we recommend reading the Language Reference Manual and the sections on the Editor (**Section 5**), Compiling and Linking (**Section 4**) and the library modules (**Section 9**) first.

If you wish you can copy the .EMS file from your Amstrad master disc to your Modula-2 disc and rename PROF8512.SUB to PROFILE.SUB and the files will be copied to the ramdisk automatically on boot.

3.1.2 Getting started on an Amstrad PCW 8512

Naturally it is easier to run **FTL Modula-2** on the PCW8512 than the earlier Amstrad machines.

To make a backup of your distribution disks format a disk in CF2DD format using DISCKIT and leave this disk in drive B. Now for each of the 4 sides of the distribution disks type

```
wp a: b: -q
```

and this will make a backup on just one disk. Mark this disk FTL Modula-2 Backup and, to be safe, make a copy of this entire disk using DISCKIT. This new disk will be your work disk. You can now put the distribution disks and your original backup away safely.

You can now delete the files MEMCHECK.COM, MEDUMB.COM, SETTERM.COM, TERMINAL.DAT and USQ.COM from your work disk as these are not required.

The method of working that we suggest initially is to have all the parts of the system that you need together with the .SYM files output by the compiler on drive M and your source files and .REL files on a real disk in drive B. To set this up we need to run the SETSEARC program whilst logged into drive B. This will ask a number of questions about disk drives, answer @ [ENTER] to all of these. The program will then ask if you want listing on or to change disks during links ; reply N[ENTER] to these.

We have supplied a submit file called PROF8512.SUB which copies the files that you normally need to M:, but to run this we need the CP/M SUBMIT utility so copy this to your work disk.

Now you can type PROF8512.S from the B> prompt and the files will be copied to drive M. The other files on your work disk are not always needed (they are mainly the source of the libraries and the assembler) so you can delete these if you wish.

The next step is to log on to drive M using M:. From now on we will remain logged into this drive. To compile the LIST.MOD example program type

```
M2 B:LIST.MOD
```

Then, after the compiler has finished,

```
ML B:LIST
```

This will create the file LIST.COM on drive M: and you can run this by typing

```
LIST B:LIST.MOD CON:/O
```

which will list the program on the console.

When you next want to load **FTL Modula-2**, boot normally. It is a good idea to use SETKEYS KEYS.WP as the editor uses many of the same keys as WordStar; so the cursor keys will move the cursor for example. Then log into drive B and type PROF512.S. Finally log onto drive M:. You can now try out the editor and start writing your own programs, using for example:

```
ME B:TEST.MOD
```

but we recommend reading the Language Reference Manual and the sections on the Editor (**Section 5**), Compiling and Linking (**Section 4**) and the library modules (**Section 9**) first.

3.1.3 Getting started on an Amstrad PCW 8256

If you haven't expanded your machine's memory to 512k think about it. It should cost you £30 or less, the compiler, linker, libraries and editor will all load almost instantly and for all but the most substantial projects (such as the Editor as supplied in the Editor Toolkit) you shouldn't need to do any disk swapping. Upgrading your RAM is probably more useful than buying a second disk drive because of the faster response. If your Joyce is still under warranty it is still worth upgrading the RAM as the machines are generally very reliable and the time you save is probably worth the cost of a maintenance contract.

If you have one disk drive and expanded memory:

1. First format both sides of three disks using `DISCKIT` and at least one side of a further disk. Two of these disks will be backups of the master disks, the third will be your Modula-2 system disk and the fourth for storing your programs on.

2. Make sure you are logged into drive A and then for each side of the master disks type

```
WP A: M: -Q
```

and this will copy all of the files onto the ramdisk.

Insert the relevant side of your backup disks and type:

```
M:WP M: A: -Q
```

This will copy all the files onto your backup.

Now type `WD M:`

and then when prompted type `A` for all. This removes all the files from the ramdisk so that you have plenty of room for the next side. Repeat this process for all four sides.

3. Now copy both sides of the compiler and linker disk to the ramdisk using `WP A: M: -Q` and copy the definition module compiler from the utilities side of disk 2 to the ramdisk using `WP A:MD.COM M: -Q`. and log onto drive M:.

4. Run the `SETSEARC` utility. Answer the questions as follows:

```
Enter compiler search list :a@
Enter device to receive .SYM files :@
Enter loader search list :a@
Enter device to receive .COM files :@
Do you want to list source :N
Do you want to change disks :N
```

User input is underlined.

5. Now insert your new Modula-2 system disk into the drive and type:

```
WP M: A:
```

you will then be prompted to copy the various files on the disk. Type `Y` to the following and `N` to the rest:

`SD.COM WD.COM WP.COM M2.COM ERRMSG.DAT SYMFILES.LBR M2OVL.OVR ME.COM CONTROL.DAT` and `ML.COM`

6. Now turn your work disk over and copy `MODULA2.LBR` and `MD.COM` in a similar manner.

7. Now copy `LIST.MOD` from M: to your new programs disk.

8. Check that everything is ok; reboot the system and insert Side 1 of your Modula-2 system disk and type

```
WP A: M: -Q
```

Log on to drive M:; turn the disk over and type

`WP A: M: -Q` again.

Now take out your Modula-2 system disk and insert you programs disk. Type

```
M2 A:LIST.MOD
```

This will compile the list program.

9. Now type

```
ML A:LIST
```

and this will create the file LIST.COM on drive M:

10. You can run this by typing

```
LIST A:LIST.MOD CON:/O
```

and this will list the program on the console.

When writing your programs, keep them on the programs disk in drive A and, after loading CP/M, copy all the files from your Modula-2 system disk as described above. It is a good idea to use SETKEYS KEYS.WP as the editor uses many of the same keys as WordStar; so the cursor keys will move the cursor for example. You can now try out the editor and start writing your own programs, using for example:

```
ME A:TEST.MOD
```

but we recommend reading at least the Language Reference Manual and the sections on the Editor **(Section 5)**, Compiling and Linking **(Section 4)** and the library modules **(Section 9)** first.

If you have only 256k and one disk drive:

1. First format both sides of three disks using DISCKIT and at least one side of a further disk. Two of these disks will be backups of the master disks, the third will be your Modula-2 system disk and the fourth for storing your programs on.

2. Making sure you are logged into drive A and then for each side of the master disks type

```
WP A: M: -Q
```

and this will copy most of the files onto the ramdisk. The ones that it has not will be followed by Disc I/O Error. Don't worry; this just means that the ramdisk is full. Make a note of the files that haven't been copied. Make sure that WP.COM and WD.COM have been copied successfully. If they haven't delete another file and copy it using

```
WP A:WP.COM M: -Q or
WP A:WD.COM M: -Q respectively.
```

Now log onto drive M: , insert the relevant side of your backup disks and type:

```
WP M: A: -Q
```

This will copy all the files that you have transferred so far. Now type

```
WD M:
```

and then, when prompted, type Y to every file except WD.COM and WP.COM. This deletes nearly all the files on drive M so there is plenty of room for the rest of the disk.

Now type

```
WP A: M:
```

and type N to the files that you have already copied and Y to those that you haven't. Next transfer them to your backup as before and repeat this for all four sides.

We recommend working with your source and .REL files on ramdisk and the Modula-2 system in the disk drive. You should be careful to back up your source programs to real disk. Unfortunately there is not enough room on your ramdisk to hold the whole system; this is why we recommend that you upgrade to 512k.

3. Now make another copy both of sides of the compiler and linker disk this will be your Modula-2 system disk.

4. Delete SETTERM.COM, TERMINAL.DAT and the README.NOW file from side 1 (the compiler side) of your new Modula-2 System Disk.

5. Copy MD.COM from the Utilities side of disk 2 on to drive M: and from there on to side 1 (the compiler side) of your new Modula-2 System disk

6. Run the SETSEARC utility from A:. Answer the questions as follows:

```
Enter compiler search list :ma
Do you really have a drive m ? y
Enter device to receive .SYM files :m
Enter loader search list :ma
Enter device to receive .COM files :m
```

```
Do you want to list source :N
Do you want to change disks :N
```

User input is underlined.

7 Delete SETSEARC.COM from this side of the disk as it is no longer required.

8. Now turn your Modula-2 system disk over and run SETSEARC again and answer the questions as before.

8. Delete SETSEARC.COM as it is no longer required.

9. Now copy LIST.MOD from the source side of disk 2 to drive M: and from there onto your new programs disk.

10. Put the compiler side of your work disk in the drive A , type

```
M2 M:LIST.MOD
```

This has compiled the list program.

11. Now turn over the compiler/linker disk in the drive and type

```
ML M:LIST
```

and this will create the file LIST.COM on drive M:

12. You can run this by typing

```
M:LIST M:LIST.MOD CON:/O
```

and this will list the program on the console.

When writing your programs keep them on the programs disk and turn the disk in the drive over when you switch from compiling to linking.

3.1.4 Getting started on an Amstrad CPC 6128

A RAM disk is a worth while investment. For about £80 you will get the advantage of almost instant loading of the compiler, linker and libraries.

1. Use DISCKIT3 to copy both sides of both disks; these are your backup disks.

2. Now make another copy of both sides of the compiler and linker disk. This will be your drive A work disk with the main Modula-2 system on it.

3. Format another disk using DISCKIT3. This will be the disk you store your programs on.

4. Put the compiler side of your work disk in drive A and run SETTERM. Enter the number for the Amstrad CPC (19 at the time of writing). Then answer N when asked if you wish to test it and then 0 to exit to CP/M.

5. Delete SETTERM.COM, TERMINAL.DAT and the README.NOW file from side 1 (the compiler side) .

6. Copy MD.COM from the utilities side of Disk 2 to the compiler side of your work disk.

7. Run the SETSEARC utility. Answer the questions as follows:

```
Enter compiler search list :ba
Enter device to receive .SYM files :b
Enter loader search list :ba
Enter device to receive .COM files :b
Do you want to list source :N
Do you want to change disks :N
```

User input is underlined.

8. Delete SETSEARC.COM as it is no longer required.

9. Now turn your work disk over and run SETSEARC again and answer the questions as before.

10. Delete SETSEARC.COM as it is no longer required

11. Now copy LIST.MOD from the compiler side of your Modula 2 system disk to your new programs disk.

12. Put the compiler side of your work disk in drive A and your programs disk in drive B and whilst logged into drive A type

```
M2 B:LIST.MOD
```

This has compiled the list program.

13. Now turn over the compiler/linker disk in drive A and type

```
ML B:LIST
```

This will create the file `LIST.COM` on drive B:

14. You can run this by typing

```
B:LIST B:LIST.MOD CON:/O
```

and this will list the program on the console.

When writing your programs keep your programs on the disk in drive B and turn the disk in drive A over when you switch from compiling to linking.

If you have a CPC 6128 with only the in-built disk drive:

Buy a disk drive! Although it is possible to compile very small programs with only one disk drive, it is not possible to compile and link a program that uses a large number of the library modules (such as the `LIST.MOD` example program) without changing disks about 50 times.

3.2 Getting Started Quickly on a non-Amstrad Machine

Before you do anything else, make copies of the distribution disks (put write disable tabs on them if they haven't already got them). Then, put the distribution disks in a safe place and work from the copies. You should never use the distribution disks for anything but making copies.

The system will usually come on two disks. We assume in this section that you are creating a two disk system. If you are really short of space, the fuller notes in the next subsection should be consulted.

Format two disks. If you have a choice of disk capacities, use the highest capacities available. Put a copy of your operating system on one disk and label this disk `Disk A - Modula-2 System Disk`. Label the other `Disk B - Modula-2 Work Disk`.

Because of the large amount of data and programs on the disks and the small size of some formats, you will find that some of your programs have been placed in libraries and possibly squeezed so that we can fit everything onto the master disks. The following instructions will tell you how to get these files out of the libraries.

The files you absolutely must have in order to have a usable system are as follows:

i) The compilers and the linker. These are called M2.COM, MD.COM and ML.COM. You also need the file M2OVL.OVR which contains overlays for the linker and the compiler. Normally, all these files will be on your A disk but the overlay file can be put on your B disk instead.

The file ERRMSG.DAT should be copied to your A disk. This disk file contains the error messages used by the compiler in a compressed form. If you have small disks, leave this step for last!

ii) The editor. Select one of the supplied editors. ME.COM is the normal editor. It may be a memory mapped editor if you have a system for which we have a memory mapped editor. MEDUMB.COM is a version of the editor for users of terminal with inadequate escape sequences for controlling the screen. The next point deals more fully with the selection of editors.

You can use the compiler and the linker without the editor, just as you would use a normal stand-alone compiler, but that is making life difficult for yourself.

Put the editor on the A disk.

iii) The next step is to create the file CONTROL.DAT on your A disk. To do this, copy the file TERMINAL.DAT and the program SETTERM.COM to your A disk and run the program.

This step is not needed if you are using a memory mapped version of the editor, such as is supplied with the Osborne O1 or Microbee versions of the system. In this case, the program ME.COM will be already set up for your machine.

The SETTERM program allows you to select a terminal type from a number of predefined terminal types. It also allows you to define a new terminal type. If you need to do that, you need to read the next section of the manual.

For now, we will assume that your terminal is on the list and that you can select it just by typing its number. You can test the definition if you like. To exit, just enter return when asked to select a terminal type for the second time.

You can also configure the editor to use your arrow and function keys. These are configured while using the editor, rather than being set up in a separate program and are covered in the section on the editor.

You can now delete the SETTERM.COM program and the TERMINAL.DAT file.

Some terminals are unable to perform some of the operations *Delete Line* , *Insert Line* or *Delete to End of Line.* These are noted in the configuration as *dumb* terminals. If your terminal is one of these, you will have to use the dumb editor MEDUMB.COM. Delete the normal editor and rename MEDUMB.COM to ME.COM.

iv) Now copy the file MODULA2.LBR to your B disk. (Actually, as we shall see in a moment, it can be placed on either disk). This file contains the relocatable binaries for all the standard modules supplied with the system.

v) Finally, copy the file SYMFILES.LBR to either drive.

If the library files are too big to fit on any one disk, you can extract the files using the provided library utility (MLU) and spread the extracted files across disks. SETSEARC can then be used to make the compiler search those disks for files.

vi) By default, the compiler assumes that you have two disk drives called A and B. Whenever it needs a file, it will search drive A first and then drive B for the file.

If you have a system with more or less than two drives, or a system on which the drives are not called A and B, or if you want to search B first, you will need to run the SETSEARC program. See the next section for details of SETSEARC.

If you have a large disk, such as a hard disk, or a RAM disk, you may be able to fit all the files from disks A and B onto that large disk. In this case, use SETSEARC to set all the search lists to be @, which represents the current disk.

Note that the linker (ML) always searches all the disks in the search list for a file. When you receive your system, many of the standard files are contained in a library file (MODULA2.LBR). This library is not searched until the entire disk search list has been scanned. For this reason, keep your linker search list as short as possible.

vii) Finally, depending upon the amount of space you have left, you may wish to place some of the sources for the standard modules on your disks. The definition modules are in the library DEFFILES.LBR. The implementation modules are in MODFILES.LBR while the assembly language modules, are in ASMFILES.LBR with the opcodes file for the assembler.

3.3 Compiling and Running Your First Program

You should now be ready to compile and link a program. The program HITHERE is a suitable candidate since it is very short.

To compile, link and execute HITHERE, copy the file HITHERE.MOD from the distribution media to your B disk. Then do the following:

```
M2 b:HITHERE.MOD
ML B:HITHERE/D
HITHERE
```

This should simply print `Hi there` on your terminal.

If you have purchased the Editor/Toolkit, another good test for the compiler is to recompile and link the editor. If you can do this, you have little to worry about.

Some of the editor modules import a lot of other modules. If you are short of memory, modules which import lots of other modules will give trouble, since the symbol table for each definition module must be loaded. Even though definition modules are compacted to take the minimum possible space, the symbol table file for a large definition module takes up a significant portion of the symbol table space.

If you have only small disks, you will need to set up a special disk for re-compiling the editor with just the editor sources on it and you may have to alter the file RECMPED.SUB to allow you to change disks during the compilation.

3.4 SETSEARC in Detail

The program SETSEARC.COM patches MD.COM, M2.COM and ML.COM for your system. With SETSEARC, you can do the following:

i) Change the search list for .SYM files. This includes searches for the SYMFILES.LBR library file as well. By default, .SYM files are looked for on the B drive followed by the A drive. (The defaults are the settings in the compiler when it arrives and before you run SETSEARC)

The .SYM files are symbol table files. They are output by the definition module compiler. They contain the details of all symbols defined in the definition module. These files are read by the compilers whenever the corresponding file is imported.

ii) Change the search list for .REL files searched for by the linker. By default, the linker looks on drive B and then drive A.

The .REL files are the relocatable files produced by the compilers. These files are read by the linker to produce an executable program. The linker will automatically search for and link in any .REL files it requires as the result of an import until all modules which have been imported by any module have been loaded.

iii) Change the disk onto which .SYM files are written by MD.COM. By default, they are written to the logged in disk. It is best if this disk is the same as the first disk in the list given for .SYM files.

iv) Change the default disk to receive .COM files produced by the linker. By default, they are written to the logged in disk.

v) Turn the listing off. By default, all text compiled is listed to the terminal. Turning listing off makes the compiler run faster. As you cannot change the listing option when you call the compiler from the editor, turning the listing off is recommended.

vi) Enable the *change disks* option. If this is selected (it is deselected by default), and the linker has not found a module, it will prompt you to change the disk and continue.

This is useful if you have insufficient space for all your .REL files on a single disk. If you do have enough space, don't use it - it gives you one more thing to worry about when you are missing a module.

The search lists are lists of disk drive identifiers, such as A, B, C etc. In addition, the character @ can be used to denote the currently logged in disk (that is, the logged in disk when the compiler is run - not when SETSEARC is run).

You should enter the list in the order in which the disks are to be searched. Note that the compiler will run fastest if the file is always on the first disk searched.

Run the program after you have copied the files to your work disk. The files patched are MD.COM, M2.COM and ML.COM but all three do not have to be present - the program will patch those it can find.

The program will prompt for the required information. For example:

```
SETSEARC

Enter compiler search list :bc@
Enter device to receive .SYM files :@
Enter loader search list :bc@
Enter device to receive .COM files :a
Do you want to list source :N
Do you want to change disks :N
Patching MD.COM
Patching M2.COM
Patching ML.COM
Done
```

If you use a hard disk or a large RAM disk, then it is probably best to set all the search lists to be just `@`, and then to always log on to the RAM disk before using the system.

If you enter a drive designator other than `A`, `B` or `@`, the program will ask you to confirm that you have a drive with that designation with the question `Do you really have a drive x?`. Enter `yes` to accept the designator or `no` to re-enter the search list.

3.5 SETTERM in Detail

When you receive the system, you will probably have to set it up for your particular console. To do this, copy the files `TERMINAL.DAT` and `SETTERM.COM` to your system disk, then run the program `SETTERM`.

This will not be necessary if you have received a memory mapped version of the editor. If this is the case, you will not need a `CONTROL.DAT` file on the logged in disk and the editor should work straight away.

When you load `SETTERM`, you will be presented with a list of terminals which have been pre-defined for use with the editor. If yours is on the list, select it and exit from `SETTERM` by pressing return when you are asked to select a terminal type for a second time. It is up to you whether you choose to test the installed terminal type. It is easiest to ignore that possibility for the moment until you have tried the editor.

If you directory your logged in disk, you will now see a file called `CONTROL.DAT` there. This file contains the control codes used by your terminal for various functions.

If you cannot find your terminal on the terminal list, first see if you can find an equivalent terminal. If you can find one which is at least close, you can edit that definition to produce a new definition for your terminal.

As well as using the terminal definition file with the editor, you can use it with your own programs, through the module `ScreenIO`, which is supplied with the compiler, but the source to the implementation module is part of the Editor/Toolkit. For this reason, it is a good idea to complete as much of the table as you can.

The editor, however, only uses a few of the functions in the terminal configuration file. These are as follows:

```
Delete Line     Graphic Boxes
Insert Line     Clear Screen
Delete to EOL   Cursor Position (Go to x,y)
Bold            Half Intensity
```

You can manage without some of these; *Half intensity* is only required for cosmetic purposes. You can replace *bold* by some other means of highlighting (underline is a good choice). The *graphic boxes* are used around menus and can be managed without, though with some detraction from the aesthetics of use of the editor.

Unfortunately, some terminals cannot even do these. If your terminal cannot perform any of the first three functions (*delete line*, *insert line* and *delete to end of line*), you can use the dumb editor (MEDUMB.COM). This editor tends to be slow, since scrolling operations require the whole screen to be rewritten. Another possibility, if you have bought the Editor/Toolkit, is to create a memory mapped version of the editor. To create a memory mapped version of the editor, modify the module SCREENIO.MBE for your computer and re-compile the editor, substituting SCREENIO.MBE for SCREENIO.MOD in the file RECMPED.SUB.

Providing that your video memory map is in the normal memory used by CP/M, and is not in another bank, it should only be necessary to change a few constants in SCREENIO.MBE to produce a memory mapped version for your computer.

Once SETTERM has been used, you can delete both it and the file TERMINAL.DAT from your work disks. If you have created a new terminal definition, save the TERMINAL.DAT file before deleting it as the new definition is saved to this file as well as being output to the CONTROL.DAT file.

On some terminals, when a character is written into the last column of a line, the screen scrolls. This means that the edit text gets displayed with blank lines in between lines of text. This can be particularly troublesome with the dumb editor, because it fills the rest of a partially full line with blanks to simulate *erase to end of line.* If this is a problem, or if your terminal scrolls up a line when you write to the last line of the screen, reduce your screen width by one.

3.6 The Files on the Distribution Disk

Because of the lack of space on the disks and the large amount of data we send you - including the source code to all of the standard modules - you may find that the files on your disks have been squeezed and or packed into libraries. If this is the case, you will receive a separate sheet of instructions showing how to unpack and unsqueeze the files.

There are several types of file on the disks.

i) The files with the extensions .COM

M2.COM	The [implementation] module compiler
MD.COM	The definition module compiler
ML.COM	The linker
ME.COM	The editor
MEDUMB.COM	The editor for dumb terminals
ASM.COM	The assembler.
SETTERM.COM	The terminal configuration program
SETSEARC.COM	The search list set up program
MLU.COM	The library file editor
M2OVL.OVR	The overlay file for the compilers and linker.
WP.COM WD.COM SD.COM	The HiSoft 1k Utilities

ii) The files with extension .SYM

These are the symbol definition files for the definition modules. One is produced whenever a definition module is compiled. They must be present whenever you compile a program which imports modules. They need not be on the A drive: the compiler will search for them. The SETSEARC program allows you to change the list of disks that the compiler will search for .SYM files.

These files are in the library SYMFILES.LBR. There is no need to extract them from this library as the compiler can use them directly from the library.

iii) The files with extension .REL

These are the relocatable binaries for the supplied source files. They can be on any disk. A .REL file is created or extended whenever a module is compiled. The linker links .REL files to produce an executable program.

These files are not in the same format as Microsoft .REL files.

iv) The files with extensions .MOD, .DEF and .ASM

These are the definition and implementation modules for the supplied modules. Note that there is no source for SYSTEM or for LOADER. These are not really modules. SYSTEM.REL is in fact the run time support for the compiler; it is only slightly related to the definition file represented by SYSTEM.SYM. It is quite impossible to regenerate SYSTEM.SYM with a Modula-2 program.

These files are not needed unless you want to recompile or modify a supplied module. On the other hand, it is convenient to keep the .DEF files on your working disks so that you can refer to them from within the editor when writing another module.

The .MOD files are in the library file MODFILES.LBR. The .DEF files are in the library file DEFFILES.LBR. The .ASM files are in the library file ASMFILES.LBR.

v) The .DAT files. These are some data files used by the system.

CONTROL.DAT is the terminal definition file for your machine. It may not be present on the distribution disk since it is created by SETTERM.

TERMINAL.DAT is the file of available terminal definitions. If you add a new one, we would be most pleased to receive a copy of it. If you receive a memory mapped version of the editor, this file may not be present. Once you have selected the terminal driver you wish to use, this file (and the program SETTERM.COM) can be deleted from your work disk.

EDITSTAT.DAT ought not be present on your distribution disk. It will be created by the editor the first time you use it on any disk (it is always created on the logged in disk). It keeps a list of the names of all files edited and the position in the file so that when the editor is reloaded, you are placed back where you left off.

MACROS.DAT will also be created by the editor the first time you use it. It is used to store the definitions of macros and special keys.

ERRMSG.DAT contains the error messages for the compiler. It must be on the logged in disk when you run a compile. Usually, it will be on the same disk as the compiler.

OPSASM.DAT is a file of opcodes read in by the assembler. It must be on the logged in disk when the assembler is run, so it is normally kept on the same disk as the assembler.

vi) The Library Files

MODULA2.LBR is an LU file of relocatable modules for the standard modules. The files in this library are all .REL files. The linker can link directly out of this file, but will link a file that *stands alone* on the disk in preference to a file of the same name in the library. This avoids having to update MODULA2.LBR every time you recompile a file that is contained therein.

Similarly, the standard .SYM files have been placed in a library called SYMFILES.LBR. Once again, the compiler will use a free-standing .SYM file in preference to one that is in the library.

There will probably be several other library files on the master disks.

MODFILES.LBR	contains the sources for the supplied modules.
DEFFILES.LBR	contains the sources for the definition modules.
ASMFILES.LBR	contains both the sources for the modules that are written in assembler.

Provided that you have at least 180k on each of two disk drives, you will be able to use the compiler successfully. You will not have a lot of space available on the A drive but your B drive should have about 50k available. Make sure that you delete SETSEARC.COM, SETTERM.COM, MEMCHECK.COM and TERMINAL.DAT so as to make available the maximum amount of space.

You can put the .REL and .SYM files, or the library files containing them; MODULA2.LBR and SYMFILES.LBR respectively, on any disk that is convenient. It is best to keep all files with a given extension on a single disk, so that you can use SETSEARC to set the compilers to look at that disk first. Otherwise, time will be wasted in an extra disk directory search.

If your disks are smaller than 180k, you are going to have to organize things a little. It is best to keep the two compilers (M2 and MD) on a disk with the editor if possible. With the attendant .DAT files, this requires around 118k. You will probably also be able to add the linker (ML) to give you a total of 156k, which leaves you enough room for your favourite utilities.

The assembler, which is only used occasionally, and the auxiliary programs can be kept on another disk.

If you have a single drive system, you should still be able to use the product provided that the drive is at least 400k. In this case, you will probably want to set up one disk for editing and compiling and a second disk for the linker and your .REL files.

Note that you never need any of the source files on line while using the compiler: all the information required to compile and link modules is contained in the .SYM and .REL files. However, having the .DEF files on line is very nice if you have the space, as you can then bring up a definition module in a window of the editor when you need to see how to call a particular routine.

We do not recommend attempting to run the compiler if your disks are less than 180k, which is the capacity you get on single sided, forty track double density.

If you have 100k disks, then it is most certainly long past time for an upgrade.

Please note that the next page is page 29!

4 Compiling and Linking

This section describes the use of the compiler and linker.

These programs can be used as stand-alone programs, just as would be done with a conventional compiler. In addition, however, you can call the compiler from inside the editor and you can return to the editor either at the end of the compilation or when an error is detected. In the latter case, the editor will position to the place of the error.

There are a large number of compiler and linker flags. None of these are required for successful compilation. The most important flags are the /D flag in the linker and the /R flag in the compiler.

4.1 Compiling Programs

You can compile programs either through the editor or in stand-alone mode. This section describes the stand-alone mode. To compile from within the editor, type ^O (Control and O) while editing and the rest should be fairly obvious. See the section on the editor for a full description.

To compile an implementation module, or a module without a definition module, use the M2 command:

```
M2 b:MyMod.MOD
```

You must give the extension of the file to be compiled.

If you want to call the relocatable something different (this is not normally advised), give the required file name as a second parameter. You can also redirect the relocatable to a different disk by giving the drive designator for the required disk:

```
M2 b:MyMod.MOD YourMod.REL
M2 b:MyMod.MOD a:
```

By default, the relocatable is placed on the same disk as the source (even if you give a file name but no drive designator!).

To compile a definition module, use the command MD.

```
MD b:MyMod.DEF
```

The second parameter can be used in the same way as in M2. It still affects the .REL file, not the .SYM file.

Whenever you recompile a definition module, you must recompile the implementation module as well. However, you may recompile the implementation module as often as you like without recompiling the definition module. The reason for this difference is that the .REL file for the implementation part is a appended to the definition part. Whenever the implementation part is re-compiled, the previously appended information is overwritten. Recompiling the definition module causes the .REL code for the implementation part to be lost.

Also, if you recompile a definition module, any module which imports the recompiled module must be recompiled as well. This is because the .SYM file contains linkage information which is copied by modules which import the module, and recompiling the definition module may change the linkage information.

When you compile the implementation module, the .REL file produced by the definition module compilation must be on the disk to which you are writing the .REL file for the implementation part, since the code is appended to that file. Several flags can be used on a compilation:

/S This flag causes the compiler to use short (256 byte) buffers instead of the normal 1024 byte buffers. As a result, you can compile larger modules at the expense of compilation speed.

/L Toggle listing of the source file on the terminal. You can also turn listing on and off in the file itself with (*$L+*) and (*$L-*). The initial value can be set with the SETSEARC program.

/T This flag causes extra information to be output to the .REL file to allow tracing of program execution, and to allow the printing by the linker of individual procedure load addresses. See the loader flags for more details.

/P This flag is similar to the /T flag, except that only the procedure trace information is output. The statement trace information is suppressed. This results in smaller executable files while still retaining some trace information.

/E This flag is used by the editor to inform the compiler to load the editor at the end of the compilation. This flag is not normally used directly.

/R This flag enables range checking. The compiler will detect, usually at run time but sometimes when the code is compiled, assignments to subranges of values which are not in range. It will also detect arithmetic overflows and subscript range errors.

/U This flag does the same thing as the /R flag except that extra checking is included. This increases the size of the code but will pick up out-of-range errors caused by the value being assigned not being a valid value for its type.

For example:

```
TYPE Colour=(Red,Green,Blue);
VAR c:Colour;
BEGIN
c:=Colour(4);
```

This error will not be picked up with the /R flag because the left hand side and the right hand side are both of type `Colour`. The /R flag assumes that if the value being assigned is of a type which cannot cause a range error then no range error can occur. The /U flag does not make this assumption. The /U flag will catch some errors, typically caused by undefined variables, that the /R flag will not catch.

You should give any flags at the end of the command line, after any file names. For example:

```
M2 b:editcont.mod/s
```

4.1.1 Pseudo-Comments

Some of the flags that can be used on the command line can also be used as *pseudo-comments* in the text of the file being compiled. The flags are enclose in comments and are introduced with a $ - for example `(*$L+*)`. Note that the flag consists of the $ character followed by the character for the flag and then either a + or a - to enable or disable the option respectively.

There must be no spaces in this sequence. You can include spaces (or spaces followed by a comment) after the + or -. The flags that can be used in this way are L, T, R, U and P.

In addition, there is an A pseudo-comment. This pseudo-comment is described in the reference manual (**Section 3.3** of the reference manual).

4.2 Linking Programs

To run the linker, use the command ML followed by the name of the main module for the program.

```
ML b:MyMod
ML b:MyMod/d
```

This simple form of the linker command is all you need to perform most links. The second form (using the D flag), separates the data from the code and so produces a smaller .COM file. For this reason, using this flag is recommended.

The .COM file produced by the link will be output to the disk selected with SETSEARC. The default output disk is the currently logged in disk. Any modules that must be loaded to complete the link will be loaded automatically. This is a big advantage over most C compilers, and those Pascal compilers which support separate compilation, since those compilers require you to specify every module that is to be loaded, or to keep your modules in a library.

Linker commands can be entered on the command line or they can be entered after receiving an asterisk (*) prompt from the linker. This facility should only rarely be required. For example, you may need it if you are using CORG in an assembly language module and you want to pack as much code as possible below the CORG address to avoid wasting space. The CORG pseudo is described in the section on the assembler.

You can also use the long form of the command to load up modules which would otherwise not be included in a link. Doing this would not make sense in any other programming language, but it is possible to have any number of main programs in Modula-2. For example:

```
ML b:LowData,b:MyMod
```

This command would explicitly place LowData first in memory.

If a command is too long to fit on a single line, it may be terminated by a comma. The linker will then ask for another line of commands, prompting you with an asterisk. The comma must be a comma that would occur normally in the command if it had not been split across lines. This comma convention can be used on the command line as well as in response to an asterisk.

The complete form of the command is:

```
ML com file name =rel file,rel file,…
```

For example:

```
ML test=b:me,b:newstorage
```

If the com file name is omitted, it defaults to the name of the first .REL file (with the extension changed to .COM).

The .REL files are loaded in the order given. Any modules imported by the modules named in the linker command are not loaded until after all the named modules have been loaded.

The order of the module names in the command line will not affect the order of execution of the main program parts (see **Section 4.2.2**). There is no way to change the order of execution of main program parts.

In other linkers, the relocatable files contain *entry points*, each of which has an explicit name. This means that other linkers allow you to replace a module by another module with a different file name but with the same entry point names. This is not possible in Modula-2. The entry points are simply numbers. If you want to replace a module with another module, you must rename the .REL files.

On the other hand, this means that you never have the problem, which is common with other languages, of finding several entry points in different modules all have the same name.

Once all the explicitly named modules have been loaded, the linker searches for any other modules which have been imported by any loaded module. This process continues until all imported modules have been loaded or diagnosed as missing. Only the .REL file needs to be available to the linker; the .SYM files, and the source files are not used.

The .REL files for required modules may either be on one of the disks in the search list set up with SETSEARC or they may be in a library called MODULA2.LBR. If a .REL file exists as a separate file on one of the disks and it is in the library file as well, the free standing version will be loaded. This avoids the problem of having to update your library every time you change a module.

The libraries are created with the utility MLU.

The name used to import a file is used as the file name for its .REL file, so you must ensure that these names match. For this reason, you should always make the file name of any module the same as the module name.

For example, if you had a module called FRED:

```
IMPLEMENTATION MODULE FRED;
```

It should be kept on a file called FRED.MOD. The .REL file must be called FRED.REL, or it will not be found by the linker.

Of course, because of CP/M's naming conventions, only the first eight characters are significant and the name must be converted to upper case. The loader and the compilers always convert to upper case before searching for a .REL or a .SYM file. You can use upper or lower case in the module header and in import statements.

For example, if you import the module `Storage`, then the linker will look for the file `STORAGE.REL`. `Conversions` becomes `CONVERSI.REL`.

Recall that the `SETSEARC` program can be used to alter the list of disks to be searched for a `.REL` file.

4.2.1 Magic Numbers

When you compile a definition module, the compiler generates a *magic number* which is stored in both the `.SYM` file and the `.REL` file. Each module which imports the given module records this magic number in its own `.REL` file. If you now recompile the first definition module, the magic number will be generated a second time and, unless you are unlucky (less than one chance in 256), the number will no longer match the magic number recorded in the importing files.

When you link with the files in this state, the linker will report that the first module has been recompiled since the modules which import it. This message is a warning message only, but you ignore it at your peril: the linkages to the re-compiled module from those which import it may be incorrect, and may not even be made at all!

This is only a problem if you recompile the definition module. You can recompile the implementation module as often as you like.

To overcome the problem, recompile the modules which import the original module. If it is imported by an implementation module, you need only recompile the implementation module. If it is imported by a definition module, you will have to recompile both the definition module and its associated implementation module and you will also have to recompile any modules which import this module in turn.

If this sounds arduous, it's probably because it is! To keep life simple, we have provided a program `Preceden` which will generate a table of module dependencies from a set of modules, and an associated program `BuildSub` which will generate a submit file to recompile all the modules required to be recompiled after recompilation of a given module, using the tables generated by `Preceden`.

You will have to compile and link `Preceden` and `BuildSub` before you can use them. They are described later in this manual.

Occasionally, you may get the `recompiled since` message even after recompiling all the importing modules. This occurs if the version of the `.SYM` file does not match the version of the `.REL` file for the original file.

4.2.2 Order of Execution of Main Program Parts

In Modula-2, every module can have a main program part. This differs from most other languages which only permit one main program part in any executable file.

The order in which these main program parts are executed can be important. The main program part may set up initial values for variables in the module. If a procedure in the module were called before the main program part had been executed, it could be accessing undefined variables.

For example, the main program part of the module `Storage` sets up a pointer to the start of the heap. If `NEW` were called before the main program part of `Storage`, then the pointer returned could point anywhere at all.

To make sure this does not happen, if a module imports a second module, the main program part of the second module (if any) will be executed before the main program part of the first module.

So, for example, if `TreeBuilder` imports `Storage`, then the main program part of `Storage` is executed before the main program part of `TreeBuilder`.

If a cycle exists (for example, `a` imports `b` which imports `c` which imports `a`), one of the modules is selected arbitrarily, and a diagnostic is printed to tell you which one it was. This diagnostic will be printed even if only one of the modules in the cycle has a main program part.

4.3 Linker Output

Whenever you perform a link, the linker outputs some information to your terminal.

For each module, the name of the module and the address at which it starts are printed. If the `/R` linker flag is selected (see below), each module is followed by a list of procedures in the module, with their addresses. All addresses are in hexadecimal.

If you get messages such as `bad code in file`, they refer to the most recently listed module.

After all modules have been loaded, the linker links together the main program parts. You will be given a list of module names in the order that their main program parts will be executed. With each module name is the address of the start of the main program part.

During the linking of the main program parts, you may get the message `Circular references`. This means that two or more modules import each other (For example, module `a` imports module `b` which imports module `c` which imports module `a`, thus forming a loop). In this case, the linker picks one of the modules at random and displays a message saying which one has been selected.

You may also see a message `e recompiled since d` This means that `d` imports `e` and the definition module of `e` has been recompiled since `d` was compiled. You should recompile `d` and re-link. If the definition modules of `d` imports `e`, then you must recompile both the definition module and the implementation module for `d`. If the import is in the implementation module only, then you need only recompile the implementation module.

At the end of the linker listing is a set of four numbers. These are:

`Data Size`	Bytes of uninitialized data in the program.
`Code Size`	Bytes of code in program.
`Data in Code`	Bytes of initialized data (embedded in code).
`Top Address`	Top address used. Address for start of heap.

4.4 Linker Options

There are several flags you can use with the linker. If you use a flag, it must be given after the first file name in the command, or, at the latest, after the first .REL file. For example:

```
ML Fred/d,George
ML George=Fred/d,George
ML George/d=Fred,George
```

But not:

```
ML Fred,George/d
```

If more than one flag is used, each flag should have its own /:

```
ML Fred/D/R
```

The flags are:

/D This is the most frequently used flag. Normally, (without the flag), the linker mixes up your data and executable code. This results in a .COM file which is larger than it need be, but it does ensure that all your variables have initial values (zeros in fact), which can avoid rude surprises when you run a program you thought was working in a new context.

Entering just /D causes the static data to be allocated downwards from the top of memory, starting just beneath your operating system. This option is very convenient when you are developing a program and you do not know how large the program will be. However, you should not use it for programs you are going to send to others, because if their transient program area is smaller than yours, your program will overwrite their operating system!

Also, if you use a debugger, it will usually relocate itself to the top of memory, which is just where your data is. As a result, your program will overwrite your debugger.

Under CP/M Plus you will also have the same problems if you try to run your program from within a SUBMIT file, because SUBMIT loads an RSX at the top of memory.

To overcome this problem, use the /D:xxxx option. Here, xxxx is a hexadecimal value for the address from which (and working upwards) the data is to be loaded.

When you link with the /D option (without the :xxxx), the linker will tell you the last address used. This is the last address used by the code. Make a note of this address and use it as (xxxx) when you relink the program for distribution. It is a good idea to leave a little space for expansion of the code so that when you make some changes to the program, you will not have to recalculate the address for xxxx.

To summarize:

No /D Flag - Code and data mixed together, .COM file rather larger than required. Very safe.

/D Flag - Data in high memory. Stack starts immediately below data. Data is allocated from the top of memory down. Not good for debugging.

/D:xxxx - Data is allocated upwards from the given address. The stack starts from the top of memory. The heap starts from the end of the data and works up.

This flag only affects static data (that is, data allocated at the module level). Dynamic data; data allocated in procedures, or in modules nested inside procedures, is always allocated on the stack.

The compiler has an extension which allows you to give initial values to variables. These variables are always part of the .COM file. They are never affected by the /D flag. Also, of course, variables which use the absolute address construct are not moved by the /D flag.

For example:

```
ML b:me/d
```

The data will be just below the top of the TPA.

```
ML b:me/d:8000
```

The data works up from 8000 hex, towards FFFF hex.

/F Full FOR loop control. Normally, for a FOR without a BY clause, or with a BY constant of plus or minus one, the compiler generates fast code using DEC and INC instructions. If you have FOR loops which contain more than 32767 iterations total (for example, `FOR i:=0 TO 60000 DO` ,where i is a CARDINAL), you should specify this flag. It will produce slower code, but the large FOR loops will work correctly. If you use large FOR loops and fail to use this flag, the loop will only give one iteration.

/L This is a prelink option. With this option, you can produce a file from a group of modules which is basically the same as a .COM file except that not all the required modules have been loaded. The output file contains enough tables to allow the linking of other modules later. If you are writing a large program which uses many modules (either your own or standard modules), you can prelink all the modules you are not currently working on. This will reduce the time taken for a link, since the linker just needs to copy the prelinked modules, instead of having to go through the process of linking them a second time.

The file produced by a link with this option has the extension .REL. You can only use one prelinked module in any link, since the code produced cannot be relocated. It must also be the first module mentioned. For example, suppose that your main module `Test` imports the modules `Terminal` and `SmallIO` and you want to prelink these modules. Enter the command:

```
ML misc/l=terminal,smallio
```

This will produce a prelinked file containing these modules, plus any modules that are imported by them. (If you want to omit a module that is imported by one of the linked modules, you will have to temporarily change its name so that the linker will not find it (don't forget to change the MODULA2.LBR file either)). The files that are explicitly prelinked may not be in the library.

When you want to link to produce an executable program, just type:

```
ML test=misc,test
```

In this example, we are assuming that you keep your .REL files on disk B, and that you want the executable file to be TEST.COM rather than MISC.COM.

Do not use the /D flag or any of the ROM-able code flags when linking using a pre-linked module. Rather, you specify the desired flag values when you create the pre-linked code file.

/O The omit checking code flag. If you have compiled a module with either the /R or the /U flag, the linked code will normally contain range checking code. This flag causes the executable file to be created without any checking code.

/P The separate displays option.

The linker normally allocates one display for the entire program. A display is an area of memory which is used to access variables which are local to procedures between the current procedure and the global level. For example, if you have a procedure nested inside another procedure:

```
PROCEDURE Outer;
VAR b:BOOLEAN;
PROCEDURE Inner;
...
```

Then, when a reference to variable b is made from within Inner, the display is consulted to determine where in memory b is.

If you are running a program which uses multiple processes, and you only have one display, then the display may be altered while a process is inactive. For example, if the procedure Inner contained a transfer to another process, then by the time b was re-started, the display could have changed, and the access to b would access some other part of memory, more or less at random.

This flag gives you one display per module. This allows you to have a process per module (which is a common arrangement) without having to worry about the display.

4.4.1 Options for Debugging

There are two options which assist with debugging. If you wish to use them, you must use the `/T` option when you compile the modules which are to be debugged. The linker will accept a mixture of modules compiled with and without the `/T` compiler option, so you can restrict debugging output to the modules you are interested in.

The linker flags are:

`/T` Generate trace code. If you use the simple form of this flag (`/T`), the linker generates an `RST 6` instruction at the beginning of each procedure and at the beginning of each statement. You can change the number of the `RST` instruction used by using the form `/T:n`, where `n` is the number of the interrupt to use. Normally, you should restrict yourself to using 6 or 7. If you use a debugger, it will use one of these. You should then use the other one for tracing. Hence, if your debugger uses `RST 6` to trap instructions, you should use the flag `/T:7`. Unfortunately some machines, such as the Amstrad range, use `RST 7` for interrupts so there is only one spare restart.

The module `DEBUG` of the release system can be used as a trace routine. You may want to modify this module for your own purposes. The source of the module contains all the details of how it works. You must import `Debug` into any module that is to be traced, to ensure that the main program part of `Debug` is executed first.

If you do change the interrupt used for tracing, for example, by using `/T:7`, you need to change the constant `IntNumber` in the `DEBUG` module.

`/R` List addresses of procedures. For modules compiled with trace mode on, using either the `/T` or the `/P` compiler flag, the names of the procedures in the module and their addresses in memory, will be displayed.

4.4.2 Options for ROM-able Code

The linker contains three options to support the production of ROM-able code. These flags allow you to position the code, the stack, and to set a HALT jump address other than zero.

`/J:xxxx` The address given by xxxx is the address to jump to at the end of the program, or when a call is made to the procedure HALT. By default, this address is zero. The primary reason for this flag is to support the generation of programs which run stand-alone, rather than under CP/M.

`/C:xxxx` Code starts at the given address, rather than at 100h. This flag can be used to position code which will be burnt into ROM. For example, if the ROM starts at address 1000H, you could use the flag as:

```
ML b:MyProg/c:1000
```

Note that programs that have been linked with a base of other than 100h cannot be run as CP/M programs. That is, if M1000 has been linked with the command

```
ML M1000/c:1000
```

then attempting to run M1000 from the A> prompt in CP/M will not work. If you do attempt to run them as commands from CP/M, they will be loaded at 100H but all the addresses in the file will be set as if the code was loaded at the address given by the flag.

`/S:xxxx` Stack start address. By default, the stack starts from the bottom of the operating system, which is the top (high address) end of your TPA. This is found (when the linked program is run), by picking up the value at address 6. This flag allows you to set the start address for the stack. The stack will work downwards from this address. The address given (the xxxx) is not part of the stack.

5 The Editor

The editor is loaded in one of three ways:

i) You can load the editor explicitly, use the command ME. The command may be followed by the names of up to three files to be edited.

```
ME MyMod.MOD MyMod.DEF
```

If no file names are given the program comes up with the menu described in the next section.

ii) When an error is detected during a compilation, the compiler will give you the message `Space to continue, i to ignore and continue, e to edit.` Typing `e` will cause the editor to be loaded with the file that was being compiled as the edit file. The cursor will be positioned to the point of the first error and the error message will appear on the last line of your screen. To see subsequent errors see **Section 5.3**.

iii) If a program is compiled with the `/E` option, the editor will be loaded at the completion of a (successful) compilation. This is how the editor gets control back after calling the compiler.

5.1 The Main Edit Menu

The editor contains a menu from which you can select filing and compiling options. If you start the editor without giving any file names, this menu is displayed.

You can invoke the menu at any time by typing Control O (^O) (o for options). The menu looks like this:

Open File	Comp/Exit
Save File	Quit File
Close File	Quit Window
Split File	Exit Edit
Compile	Reset Disk

You can use the cursor keys (^X, ^S, ^D and ^E) to select an option. Alternatively, typing the first letter of an option selects the option. If there is more than one option that starts with the given letter, typing the letter selects the next one. Hence, to select `Split File` you have to type `S` twice.

Typing Escape (the key marked `Esc`, or else Control and opening square bracket ([) if your keyboard lacks the `Esc` key), will abandon option selection and return to editing. Typing return will cause the currently selected option to be executed.

The menu options are:

`open file`	open a file for editing. A new window is created with the file in it. You can have at most three windows.
`save file`	save a file. The file is not closed so you can continue editing it.
`close file`	save a file and finish editing that file.
`split file`	split a file between two windows so that separate parts of the file can be viewed and edited together.
`compile`	compile a program.
`comp/exit`	compile and then return to CP/M.
`quit file`	abandon a file.
`quit window`	abandon the current window. If this is the only window in which a file is visible, the file is abandoned as well.
`exit`	exit the editor, saving altered files.
`reset disks`	make all disks read/write. This is very useful if you are using CP/M 2.2 and cannot save an edited file because a disk is full. Simply change the disk, reset the disks and then save the file again.

The editor allows you to have up to three files open simultaneously, although with three windows open, the windows become unpleasantly small. (there is no way to adjust the sizes of the windows)

The `save file`, `compile`, `comp/exit`, `split file`, `quit window` and `quit file` commands always reference the file in the current window. When `compile` or `comp/exit` is selected, the editor will call either `M2`, or `MD` depending upon the extension of the file being edited. The extension must be either `.MOD` or `.DEF`; otherwise, using either of these options is equivalent to a `save file`.

The assembler is not integrated with the editor. You can only run the editor from the CP/M command line, and you cannot get into the editor from the assembler when an error occurs.

5.2 Basic Editing Commands

The editor is a full screen editor with a command set based on WordStar™ but with a couple of ideas taken from Emacs, an editor developed by Richard Stallman, namely, the repeat command facility and the macro facility. These latter two facilities make the editor very powerful indeed.

While the command structure of WordStar has been often abused (and probably as equally often praised), it has the advantage of being familiar to a large number of people, especially micro-computer users.

The current version of the editor is limited to editing files that will fit into memory. Producing a *virtual* version of the editor should not be difficult, as all the required changes should be restricted to the module `MakeEdit` from the Editor/Toolkit.

In any case, if you don't like the command structure, you can always change it, as the sources of the editor are available. The key assignments can be found in the module `Keyboard`. (This is a lovely excuse to have, as it can cover all conceivable problems.)

If your terminal has arrow keys or function keys which transmit more than one character (for example, left arrow might transmit the characters `<ESC>[A`) to the computer, you can define the arrow and function keys to be commands for use by the editor. How to do this is described in the section on creating macros.

In the following descriptions, the caret (^) is used to denote that the control key should be depressed in combination with the character following.

As many of the commands are the same as for WordStar, it is appropriate to start with a list of the differences.

i) The ^KX, ^KS and ^KD commands which, under WordStar, save the file and exit, save the file and continue editing and close the file respectively, are replaced by ^O followed by the appropriate menu option.

ii) The key ^B is used to change windows.

iii) In find commands, lower case matches either upper or lower case. Upper case only matches upper case.

The basic commands which are implemented are:

The cursor control commands:

^S move one character left (^H does likewise)

^D move one character right

^E move one line up

^X move on character down

Note how these keys form a diamond on the left hand side of the keyboard.

The first two of these can be combined with ^Q to produce the following:

^Q^S move to left of screen

^Q^D move to end of current line

When you position past the right hand side of the screen, the screen will be scrolled horizontally. This allows you to edit lines which are longer than the width of your screen.

The forward/backwards by a word commands:

^A	move backwards by a word
^F	move forward by a word

In this editor, a word is a sequence of alphanumerics, or a sequence of non-alphanumerics. Hence, these commands work better than they do in WordStar.

The delete commands:

^G	deletes the character under the cursor
^T	deletes the next symbol (word)
^Y	deletes the current line
DEL	deletes the preceding character

Some of these can be combined with ^Q.

^Q^Y	deletes from the cursor to the end of line
^Q^G	turns off the ringing of the bell if you strike a key which has no function.

The screen scroll commands:

^C	moves forward by one window.
^R	moves back by one window.

By *one window*, we mean that the cursor moves up or down the file by a number of lines equal to the number of lines in the window. Hence, if you have a normal 24 line screen and only one window, a move of about 22 lines is made. If you have three windows, a move of only 7 lines is made.

^Z	scrolls up one line
^W	scrolls down one line
^Q^R	returns to the beginning of file
^Q^C	goes to the end of the file

The insert mode change command:

^V — toggles the insert mode. At the beginning of an editor session, you are in insert mode. This means that any printing character you enter will be inserted into the current file, and the character the cursor was positioned to will move up one place. If you toggle into overwrite mode, when you enter a printing character, the character under the cursor is replace.

The search and replace commands:

^Q^F — causes a search for a new string. The string is remembered for use with ^L. To search backwards, precede the command with <ESC>-1. This is described further later, in the discussion of command repetition.

^Q^A — causes a search for a string, followed by replacement of the string when found.

^L — causes a search for the next string, as previously set by ^Q^F or ^Q^A. If the preceding command was ^Q^A, then the requested substitution is repeated. The <ESC>-1 construct can be used with ^L as well.

In the search commands, lower case characters entered as part of the search string will match either upper or lower case characters in the text. Upper case characters will only match upper case characters in the text.

Other commands:

^O — brings up the options display, as described in an earlier section.

^N — splits the current line at the cursor position but, unlike carriage return, does not move the cursor to the following line.

^B — Changes to the next window.

5.3 Block Moves and Labels

^K followed by a numeric digit (0 to 9) sets a label at the cursor. At present, there is no visible indication of the presence of the label. To return to the position of the label from another point in the file, type ^Q followed by the digit.

Labels are associated with files not windows. Hence, if you split a file across two windows, a label defined in one window can be *gone to* in the other window as well. However, you cannot *goto* a label in another file. That is, the ^Q digit command can never change windows.

There are two special labels: B and K. Type ^KB to define the beginning of a block of text. Use ^KK to define the end of the block. You can now use several commands which reference the block:

^KV	moves the block to the current cursor location.
^KC	copies the block to the current cursor location.
^KY	deletes the block.

You can use the ^KV and ^KC commands to move text between windows. To do this, define the block of text, use ^B to go to the destination window, position the cursor to the desired position for the block. Enter ^K followed by enough ^Bs to get back to the window which contains the block to be moved. Now type V or C.

Note that, when doing a *between window* block move or copy, you use ^B to position to the window from which the block is coming. This means that you cannot use ^K^B to define the start of a block. You must release the control key before typing the B. If you find this annoying, you can change the window change character. ^P is, we think, the only key on the keyboard that does not currently have a command attached.

These commands can be used in place of the ^KR and ^KW commands of WordStar, which read a file into the current edit file, and write the select block to a file, respectively. In fact, because you can select the text to be copied in from a file, and because you can join several block together in a file, the commands are rather more powerful than those of WordStar.

There are also a set of labels that can be set up when you have errors when compiling. If you use the 'space' option to continue compiling then the compiler remembers the position of the error. If you want to ignore a particular error press `i` and it won't be added to the table. You can keep up to ten error messages this way.

When you enter the editor the first error message is displayed and the cursor positioned accordingly. You can then use the following commands to get to other errors.

`^QEn` where n is the number of the error takes you to the nth error. The first one is `^QE0` and the last `^QE9` (if you saved 10 errors)

`^QEN` goes to the next error

`^QEP` goes to the previous error

`^QEC` goes to the current error

Note that the error positions can become inaccurate if you change your source; if this happens just recompile.

5.4 Command Repetition

All commands can be repeated by entering escape followed by the number of times the command is to be repeated and the command to perform. For example, `<ESC>10^G` deletes ten characters, `<ESC>70*` enters 70 asterisks.

Hence, to alter 10 occurrences of a string to another string, use the `^QA` command to alter the first occurrence, and then use `<ESC>9^L` to alter the others. Or, if you are confident, you can precede the `^QA` by `<ESC>10`.

The character `^J` (line feed) can be used instead of escape. You may also need to use it if you define your function keys for use with the editor, and they transmit sequences like `<ESC>4` since then entering `<ESC>4` explicitly would have exactly the same effect as hitting the function key.

The find commands: ^QA, ^QF and ^L allow the use of a negative count to denote a backwards search. These are the only commands which recognize a negative count. All other commands treat a negative count as if it were a positive count with the same absolute value. Once you have set the direction of the search, the direction remains fixed until you enter another count, so you can search through a file backwards by typing ESC -1^QF and then using ^L for the following searches.

5.5 Macros

The editor allows the definition and use of simple keystroke macros. You can define a macro in two ways:

i) Explicit definition.

Enter <ESC>D. A window will pop up saying character to define:. Enter the alphabetic character to associate with the macro.

The window will change to Macro:. Enter the text of the macro, terminated by carriage return. You may use any characters in the macro except the characters used to edit a command line: ^S, ^H, ^X, DEL and carriage return. If you want to enter these characters, you will have to use learn mode.

ii) Learn mode.

Enter <ESC>^L. You will be asked for the character to be defined. The editor then returns to normal editing. Perform the edit sequence that you want the editor to learn, then type <ESC>^L a second time. There is a limit of 60 characters on the length of the definition.

To invoke a macro, enter <ESC>x where x is the alphabetic character that you have defined. For most characters, you can also type <ESC>^x, but this cannot be used with D, L or P, since these have other uses.

To list the current macro definitions, enter <ESC>^P.

5.6 Key Definitions

To define an arrow key or a function key which transmits a multiple character sequence, proceed as if you were defining a macro, using either learn mode or command mode. However, when the prompt `Enter character to define:` appears, type the escape key. the prompt will change to `Enter key to define`. Now press the key you want to define, followed by carriage return. You can now associate a macro with the key just as if this were a normal macro definition.

The macro definitions are saved on the file `MACROS.DAT` between edits.

5.7 The Position File

The editor maintains a file `EDITSTAT.DAT` on the logged in disk, which will usually be the disk containing the editor. This file gives the position of the cursor when a file is saved. A separate entry is kept for each file name. Quitting from a file does not change the position entry for that file.

This file allows the editor to re-enter any file at the position at which it was saved. If you delete `EDITSTAT.DAT`, or a new file is edited for the first time, the editor will position to the beginning of the file. Once you get used to this facility, you will never want to return to an editor which always positions at the beginning of the file.

5.8 Stopping Macros

During the execution of a repeated command, or of a macro, you can stop the execution by typing `^U`. In addition, typing `ESC` will continue the execution of the macro or repeated command, but will stop the screen being updated after each change. This speeds up the execution.

6 The Library Manager

The library manager is called MLU. This module provides the most useful facilities found in a utility like the public domain utility LU plus a few facilities to make it easier to manage your libraries.

To use the library manager, simply enter the command MLU without any parameters. You will now be prompted for the name of the library that you want to manage. Enter the full name of the library including the extension. For example, MODULA2.LBR.

If you want to create a new library, enter the name of the new library. MLU will first confirm that you want to create the file and will then ask How many slots?. The number of slots is the maximum number of files that can be placed in the library. A typical answer would be 32. The value is always rounded to a multiple of four.

There are now a number of commands that you can use.

A Add a file to the library. If the file already exists in the library, the old copy is deleted. Follow the command letter by the full name of the file to be added. For example

```
A INOUT.REL.
```

D Delete a file from the library. Follow the D by the name of the file. For example

```
D INOUT.REL.
```

E Extract a file from the library. Write a copy of the file to the logged in disk. The file is not deleted from the library. Follow the command letter by the full name of the file to be extracted. For example

```
E INOUT.REL.
```

F Close the library file and exit. You should always exit with this command rather than ^C as if you use ^C, the index will not be updated.

L List the names of the files in the library. Each file name is listed together with the length of the file in bytes.

R Reorganize the library. The existing file is made into a .BAK file, for example MODULA2.LBR becomes MODULA2.BAK, and all the files are copied into a new library with the same name as the old library (e.g. MODULA2.LBR). When files are deleted from a library, the space that was used by the file is not recovered. Reorganizing the library produces a new copy without the wasted space.

You can also use the reorganize command to change the number of slots in the library.

U Update the library. The command is followed by a list of disk drives to be scanned. For example U AB. The given drives are scanned for any files with the same name as any files in the library. These files are then copied into the library. This gives you a quick way of updating a library with newly compiled files.

6.1 Compiling Library Files

The compiler cannot directly update files that are in a library. Hence, if you want to recompile an implementation module for a module that has been placed in a library, you must first extract the .REL file from the library, re-compile and then put the file back into the library. Here is a typical example:

```
MLU
Name of library to update:MODULA2.LBR
Command: E INOUT.REL
Command: F
M2 INOUT.MOD
Compilation Complete
MLU
Name of library to update:MODULA2.LBR
Command: A INOUT.REL
Command: R
How many slots? 32
Command: F
```

7 Assembling Programs

When we first introduced **FTL Modula-2**, we did not expect that anybody would have much use for an assembler. It seems, however, that assembler is like Rock 'n Roll - it will never die. We have therefore expanded these notes as well as improving the assembler.

The assembler uses Z80 mnemonics as defined in 'An Introduction to Microcomputers Volume II (Some Real Products)' by Adam Osborne. These are shown in the following table. Most of the standard mnemonics are available. Many instructions which access the accumulator (A) can be written with or without the accumulator - for example, either `ADD A,H` or `ADD H`. The only exceptions to the usual mnemonics are:

1 `IN A,(N)` must be written as `IN A,N`.

2 `OUT (N),A` must be written as `OUT N,A`.

3 `EX AF,AF` must be written as `EX AF`.

4 `RST` instructions must be written `RST 0` through `RST 7`, not `RST 0` through `RST 38h`.

Each line consists of a label field, an opcode field, an operand field, and a comment field. All fields are optional. There may be several labels on a line.

A label must either be followed by a colon, or must start in the first two column positions in a line. It may do both.

For example:

```
; this line contains just a comment
loop: ld a,(hl) ;this is a comment
      nop
      cp   a,0
      jc nz,end
      inc  hl
      jp   loop
```

The field following the last label is the opcode (`ld` in the example above). It must start after the first two character positions (or it may be preceded by a single tab character). If the opcode field is omitted, the operand field must also be omitted.

An assembly language statement cannot span a line break.

Identifiers may be in either upper or lower case. They are always converted to upper case.

The operands, if any, follow the opcode. Individual operands are separated by commas.

The comment field may start anywhere on a line. It is preceded by a semi-colon (`;`). The remainder of the line is then treated as a comment.

To run the assembler, use the command `ASM`.

```
ASM b:CPM.ASM b:CPM.REL
```

Both parameters must be given in full. The file `OPSASM.DAT` *must* be on the logged in disk, otherwise the assembler will hang.

The assembler always produces a listing file on the console during pass 2 unless the `LISTS` pseudo is used. It is possible for the assembler to loop if an operand is particularly badly formed. The `LIST1` pseudo can be used to list the source on pass one so that the offending line can be found.

An assembly language module takes the place of an implementation module. To interface to other modules, written in assembler or in Modula-2, you must write a definition module. The definition module looks exactly like the definition module for a module written in Modula-2.

By default, an assembly language implementation module completely overwrites the `.REL` file created by the definition module, whereas a normal implementation module appends to the definition module `.REL` file. The `MODULE` pseudo, described below, allows you to append to the definition module compiler's `.REL` file.

The interface to an assembly language routine is insecure because the assembler has no way of checking that you have in fact implemented the procedures that you have declared in the definition module.

To export procedures, you define them in the definition module in the usual way and use the LABEL pseudo to define them in the assembly language module. If you do not use the MODULE pseudo, the definitions must be in the same order in each module, since the assembler assigns label values in the order the label pseudos are encountered; there is no attempt to match a label field with a name from the definition module.

If you are using the MODULE pseudo, then the LABEL pseudo takes as parameter the name of the procedure being defined:

```
label reboot    ;reboot procedure
```

7.1 Expressions

Expressions are used in the operand field of statements. Each opcode will have one or two expressions. The expressions on a single line are separated by commas. The assembler accepts all the usual expression syntax.

The registers are:

```
AF BC DE HL IX IY
```

(these are two bytes each)

```
A B C D E H L
```

(these are one byte each. B, D and H are the top halves of BC, DE and HL respectively. C, E and L are the bottom halves.

Expression may contain the usual relational and boolean operators:

```
=,<,<=,<>,>,>=,EQ,LT,LE,LQ,GT,GE,GQ,NE,NOT,AND,OR,XOR
```

and the operators:

HIGH	Value is high byte value
LOW	Value is low byte value
SHL	Shift the left operand left by the number of bits in the right operand.
SHR	Shift the left operand right by the number of bits in the right operand.

Every expression must be of a form that is acceptable the the opcode on the line. These forms are called addressing modes. Not every instruction can accept all addressing modes. If you use an addressing mode that is not acceptable, you will get the error message `Invalid Operands for Opcode`. This should not be confused with the error message `Undefined Opcode` which is given when the Opcode itself is not recognized.

The use of parenthesis in expressions has a special meaning. They denote that the contents of the value of a label rather than the value itself is to be used.

For example,

```
LD A,temp
```

loads `A` with the value `temp`.

```
LD A,(temp)
```

loads `A` with the contents of the address `temp`.

7.2 The Instructions

Here is a complete list of the machine instruction mnemonics accepted by the assembler.

In the tables, the following symbols are used:

ADDR	A 16 bit address.
BIT	A value in the range 0 to 7.
DISP	A value in the range -128 to 127.
DATA	A 1 byte value.
PP	Any of BC, DE, IY, SP
PR	Any of BC, DE, HL, AF.
RP	Any of BC, DE, HL, SP.
RR	Any of BC, DE, IY, SP
REG	Any of A, B, C, D, E, H, L.

```
ADC (HL)
ADC (IX+DISP)
ADC (IY+DISP)
ADC A,(HL)
ADC A,(IX+DISP)
ADC A,(IY+DISP)
ADC A,DATA
ADC A,REG
ADC DATA
ADC HL,RP
ADC REG
ADD (HL)
ADD (IX+DISP)
ADD (IY+DISP)
ADD A,(HL)
ADD A,(IX+DISP)
ADD A,(IY+DISP)
ADD A,DATA
ADD A,REG
ADD DATA
ADD HL,RP
ADD IX,PP
ADD IY,RR
ADD REG
AND (HL)
AND (IX+DISP)
AND (IY+DISP)
AND A,(HL)
AND A,(IX+DISP)
AND A,(IY+DISP)
AND A,DATA
AND A,REG
AND DATA
AND REG
BIT BIT,(HL)
BIT BIT,(IX+DISP)
BIT BIT,(IY+DISP)
BIT BIT,REG
CALL ADDR
CALL C,ADDR
CALL M,ADDR
CALL NC,ADDR
CALL NZ,ADDR
CALL P,ADDR
CALL PE,ADDR
CALL PO,ADDR
CALL Z,ADDR
CCF
CP (HL)
INC (IY+DISP)
INC IX
INC IY
INC REG
INC RP
IND
INDR
INI
INIR
JP (HL)
JP (IX)
JP (IY)
JP ADDR
JP C,ADDR
JP M,ADDR
JP NC,ADDR
JP NZ,ADDR
JP P,ADDR
JP PE,ADDR
JP PO,ADDR
JP Z,ADDR
JR C,DISP
JR DISP
JR NC,DISP
JR NZ,DISP
JR Z,DISP
LD (ADDR),A
LD (ADDR),BC
LD (ADDR),DE
LD (ADDR),HL
LD (ADDR),IX
LD (ADDR),IY
LD (ADDR),SP
LD (BC),A
LD (DE),A
LD (HL),DATA
LD (HL),REG
LD (IX+DISP),DATA
LD (IX+DISP),REG
LD (IY+DISP),DATA
LD (IY+DISP),REG
LD A,(ADDR)
LD A,(BC)
LD A,(DE)
LD A,I
LD A,R
LD HL,(ADDR)
LD I,A
LD IX,(ADDR)
PUSH IX
PUSH IY
PUSH PR
RES BIT,(HL)
RES BIT,(IX+DISP)
RES BIT,(IY+DISP)
RES BIT,REG
RET
RET C
RET M
RET N
RET NC
RET NZ
RET P
RET PE
RET PO
RET Z
RETI
RETN
RL (HL)
RL (IX+DISP)
RL (IY+DISP)
RL REG
RLA
RLC (HL)
RLC (IX+DISP)
RLC (IY+DISP)
RLC REG
RLCA
RLD
RR (HL)
RR (IX+DISP)
RR (IY+DISP)
RR REG
RRA
RRC (HL)
RRC (IX+DISP)
RRC (IY+DISP)
RRC REG
RRCA
RRD
RST BIT
SBC (HL)
SBC (IX+DISP)
SBC (IY+DISP)
SBC A,(HL)
SBC A,(IX+DISP)
SBC A,(IY+DISP)
SBC A,DATA
```

```
CP (IX+DISP)
CP (IY+DISP)
CP A,(HL)
CP A,(IX+DISP)
CP A,(IY+DISP)
CP A,DATA
CP A,REG
CP DATA
CP REG
CPD
CPDR
CPI
CPIR
CPL
DAA
DEC (HL)
DEC (IX+DISP)
DEC (IY+DISP)
DEC IX
DEC IY
DEC REG
DEC RP
DI
DJNZ DISP
EI
EX (SP),HL
EX (SP),IX
EX (SP),IY
EX AF
EX DE,HL
EX HL,(SP)
EX IX,(SP)
EX IY,(SP)
EXX
HALT
IM 0
IM 1
IM 2
IN A,DATA
IN REG,(C)
INC (HL)
INC (IX+DISP)
LD IX,ADDR
LD IY,(ADDR)
LD IY,ADDR
LD R,A
LD REG,(HL)
LD REG,(IX+DISP)
LD REG,(IY+DISP)
LD REG,DATA
LD REG,REG
LD RP,(ADDR)
LD RP,ADDR
LD SP,HL
LD SP,IX
LD SP,IY
LDD
LDDR
LDI
LDIR
NEG
NOP
OR (HL)
OR (IX+DISP)
OR (IY+DISP)
OR A,(HL)
OR A,REG
OR DATA
OR REG
OTDR
OTIR
OUT (C),REG
OUT DATA,A
OUTD
OUTD
OUTDR
OUTDR
OUTI
OUTI
OUTIR
OUTIR
POP IX
POP IY
POP PR
SBC A,DATA
SBC A,REG
SBC DATA
SBC HL,RP
SBC REG
SCF
SET BIT,(HL)
SET BIT,(IX+DISP)
SET BIT,(IY+DISP)
SET BIT,REG
SLA (HL)
SLA (IX+DISP)
SLA (IY+DISP)
SLA REG
SRA (HL)
SRA (IX+DISP)
SRA (IY+DISP)
SRA REG
SRL (HL)
SRL (IX+DISP)
SRL (IY+DISP)
SRL REG
SUB (HL)
SUB (IX+DISP)
SUB (IY+DISP)
SUB A,(HL)
SUB A,(IX+DISP)
SUB A,(IY+DISP)
SUB A,DATA
SUB A,REG
SUB DATA
SUB REG
XOR (HL)
XOR (IX+DISP)
XOR (IY+DISP)
XOR A,(HL)
XOR A,(IX+DISP)
XOR A,(IY+DISP)
XOR A,DATA
XOR A,REG
XOR DATA
```

Notice that the assembler will accept many instructions with either an implicit or explicit accumulator. For example `ADD A,L` or `ADD L`. The original assembler only accepted the implicit form.

7.3 The Pseudos

The assembler supports most of the common pseudos, except for those associated with macros (e.g. MACRO, IRP, ECHO etc). The SET pseudo is called NOW to avoid confusion with the SET opcode.

The pseudos are:

CORG — Conditional ORG. The code must be loaded above the address given in the operand field. Useful for machines which use bank switching. Note that the linker will never use any space which is left unused because of an ORG or a CORG to a higher address.

CSECT — Returns to the code segment. You can have multiple DSECTs and CSECTs. They will be joined together by the assembler.

DB — Define bytes. The operand field contains one or more values to be placed in bytes. A character string may be used as an operand.

DS — Declare space. The operand field gives the number of bytes required. The bytes are not initialized. (Actually, at present they will all contain zero)

DSECT — Defines the start of one or more bytes of uninitialized data values. That is, you can only use DS statements in a DSECT. Data allocated in a DSECT will be allocated with the uninitialized data from the Modula-2 modules, instead of being a part of the COM file. This allows you to produce ROM-able assembly language modules.

DW — Define words. The operand field contains one or more values to be placed in words. Recall that on the Z80, a word value is stored with the least significant byte first. For example, if the operand is de, the order of the bytes is ed. When used as part of a DW statement, de is an integer number, not a string. Strings can only be used with the DB statement.

END — Terminates the assembly module. Must be present.

EQU — Gives the associated label the value of the expression in the operand field. The label may not be re-defined.

IMPORT Import symbols from other modules The operand field contains the name of the module being imported identifier must be given for each imported identifier, and the case must be correct. For example:

```
IMPORT Terminal,WriteString
```

However, inside the assembler, only the first eight characters are kept and those are converted to upper case. This is also true of the MODULE statement, which means that, when you write the definition module, you have to make sure that all the identifiers are unique in their first eight characters after capitalization. Also, make sure you do not use identifiers like BC because these are already defined as registers.

You can import types, variables, constants and procedures. When you import a type, what you get is a constant with the size of variables of the type in bytes as its value.

If all the symbols will not fit on a single (80 character) line, end the line with a comma that would have been required anyway and then continue on the next line.

LABEL Defines a procedure. The order must be the same as the order of the definitions in the definition module.

LIST1 Produce a listing on pass one.

LISTS List only lines containing errors.

MAIN Marks the start of an initialization part for the module. This will be executed as a main program part. The code must exit with a return (RET) instruction.

MODULE Defines the module for which this is an implementation part. The operand field must contain the name of the module. Only the first eight characters count. If used, this pseudo must be the first statement in the module.

When this pseudo is used, the code for the assembly language part of the module is appended to the code for the definition part, rather than overwriting it. Also, the symbols defined in the definition part are imported as if they had been imported with an IMPORT pseudo.

NOW — Gives the associated label the value of the expression in the operand field. The label may be given other values with subsequent NOW statements.

ORG — Set the origin counter. Care must be taken with this instruction since it may cause the linker to overwrite code already loaded (in which case a warning is given), or it may cause a gap to be left in the code. Normally, it should not be used.

Do not use the ORG statement to define labels which are outside the transient program area. Use EQU instead. If you do use ORG in this way, the link editor may behave strangely, typically producing a huge (64k) COM file.

7.4 Parameter Passing Conventions

Parameters are passed on the stack. There are two ways of passing parameters; *by value* and *by reference*. *By reference* is used for VAR parameters (as in `PROCEDURE Thing(VAR i:INTEGER);`).

For a value parameter, the variable is copied onto the stack. Single byte values are pushed as two bytes, with the value being in the lower byte. All other parameters require the same number of bytes as there length; a five byte value uses five bytes from the stack.

For a reference parameter, only the address of the parameter is pushed onto the stack. This always requires two bytes.

Open array parameters are handled differently. An open array parameter always requires exactly three words (six bytes) of stack space. The first word (highest in memory) contains the address of the parameter, even for value parameters. The next word contains the high bound for the array (as accessible with the HIGH function. The final word (lowest in memory) contains the total size, in bytes, of the value.

Parameters are pushed onto the stack from left to right so the first parameter in the procedure definition is in the highest position on the stack.

If the procedure is a function, space for the returned result is allocated on the stack before any of the parameters are pushed.

It is the called programs responsibility to remove the parameters from the stack before returning. In addition, the called program must save and restore the value of the IX register. All other registers may be altered.

7.5 Limitations

There are several limitations in the use of the assembler. Firstly, you cannot add an offset to an imported symbol. This is because the address of the imported symbol is really an entry number, so adding an offset would change the entry number.

Secondly, it is possible to get `Doubly defined identifier` messages when the assembler should be giving `Phase Error` messages. This happens when the position of an instruction changes between pass 1 and pass 2. This can be caused by undefined labels in the code.

Thirdly, the assembler is unable to access `.SYM` files from the `SYMFILES.LBR` library. The `.SYM` files must be stand-alone files.

8 The Utility Programs

A number of general purpose and Modula-2 utility programs have been included on the distribution disks for your convenience. The HiSoft 1k utilities provide alternatives to some of the CP/M programs and the Modula-2 utilities give further examples of the use of Modula-2 and of the supplied modules. You will have to compile the Modula-2 programs before using them.

8.1 The LIST Program

On your distribution media, you will find a program `LIST` which can be used to list out files. It will list ordinary ASCII or WordStar files and will accept a wild card parameter giving the names of the files to list. For example:

```
LIST b:*.DOC
```

When you first install the system, it is worthwhile to list out all the supplied definition modules for future reference. Compile and link this program and run it. The required statements look something like:

```
M2 b:LIST.MOD
ML b:LIST
LIST b:*.DEF
```

You may need to use a different drive designator, or change disks between statements. You may need to modify the list program for your printer. Some printers will not print tabs. Other printers are even more weird. For example, we have one customer with a printer which does not do a form feed until a carriage return is received, and then it prints the line and then does the form feed! He had to add a carriage return after every form feed. We resisted making the `LIST` program *all singing and all dancing* since it would have ended up being too complicated for you to hack. As it is, hacking `LIST` is a useful introduction to the language.

If, when you compile and link this and the following modules, they produce `.COM` files which seem very large, you may be forgetting to use the `/D` flag on the link edit. Forgetting this flag will cause the data to be allocated as part of the `.COM` file. See the section on linker flags for details of the `/D` flag.

8.2 The Precedence Programs

On your distribution disks, you will also find two programs in source which simplify the re-compilation of dependent modules when you recompile a definition module.

The first of these programs, `Preceden`, scans all the files given as parameters to build a table of dependencies. Compile and link this program, then run it as follows:

```
Preceden b:*.def b:*.mod
```

The parameters are the (wild card) names of files to be scanned.

The program creates a file called `Preceden.dat` will be created with a copy of the table. The table will also be output to your terminal (so you can use Control P on the command line to have it printed to your printer as well).

The second program is `BuildSub`. To run this program, use the command:

```
BuildSub Temp.sub name1 name2..[flag
```

`Temp.sub` is the name of a submit file to be created. The file will contain commands to recompile the various modules which import any of the modules in the list of names. You can then use `submit` or `supersub` to execute the command file.

A flag may be used with the `BuildSub` command: If you terminate the command line with `[I`, the modules given in the list will be included in the submit file.

For example:

```
BuildSub temp.sub Files Terminal [i
```

The `BuildSub` program expects to find the file `PRECEDEN.DAT` on the logged in disk. The file `temp.sub` can then be executed with the standard CP/M program `submit`.

8.3 The HiSoft 1k Utilities

We have provided some useful (and small!) file utilities to make file management and conversion more straightforward.

8.3.1 WP

WP.COM copies files from one disc to another. It is invoked by typing its name followed by its parameters at the CP/M prompt. The general form of the command line is

```
wp <source afn> <destination afn> [-q][-b]
```

The items specified as <source afn> and <destination afn> above are standard CP/M *ambiguous file specifications*, with optionally a drive name at the front. An ambiguous file specification is a filename which can match more than one file; this is done using *wildcards*, which are described in great detail in your CP/M documentation. WP extends the definition slightly in line with CP/M's built-in DIR command, such that a drive name alone (such as B:) is equivalent to *.* on the specified drive. If this item is left out altogether, it is taken as *.* on the current (default) drive.

The items specified by [-q] and [-b] are an optional and will be described later.

Typical WP invocations, then, would be

```
wp a: m: [RETURN]
```

which copies all files on drive A onto drive M,

```
wp m: [RETURN]
```

which copies all files on the default drive to drive M and

```
wp b:*.com a:*.bak [RETURN]
```

which copies all files on drive B with an extension of .COM to drive A with an extension of .BAK. If the source and destination files are the same, then WP prints an error message and returns to CP/M.

When a valid command line has been typed, WP collects the names of the matching files and displays each one in turn, followed by a prompt:

```
Copy (Y/N/A/Q/P/B/W)?
```

You may type Y to copy this file, B to copy the file making a *backup* (any existing destination file is renamed to have extension .BAK), W to copy the file *without* a backup, N not to copy this file, P to go back to the *previous* selection, A to copy *this and all subsequent matching* files or Q to quit now without copying this or subsequent files.

If -Q is present as the *last* item on the command line, WP does not prompt and copies each matching file without asking.

If -B is present as the final item on the command line then Y and A will automatically make backups and to copy without a backup you must use W.

8.3.2 WD

WD.COM deletes files from a disc. It is invoked by typing its name followed by its parameters at the CP/M prompt. The general form of the command line is

```
wd <afn>
```

The item specified as <afn> above is a standard CP/M *ambiguous file specification*, with optionally a drive name at the front. An ambiguous file specification is a filename which can match more than one file; this is done using *wildcards*, which are described in great detail in your CP/M documentation. WD extends the definition slightly in line with CP/M's built-in DIR command, such that a drive name alone (such as B:) is equivalent to *.* on the specified drive. If this item is left out altogether, it is taken as *.* on the current (default) drive.

Typical WD invocations, then, would be

```
wd a: [RETURN]
```

which deletes all files on drive A and

```
wd b:*.com [RETURN]
```

which deletes all files on drive B with an extension of .COM.

When a valid command line has been typed, WD collects the names of the matching files and displays each one in turn, followed by a prompt:

```
Delete (Y/N/A/Q)?
```

You may type Y to delete this file, N not to delete this file, A to delete *this and all subsequent matching* files or Q to quit now without deleting this or subsequent files.

8.3.3 SD

SD.COM is a utility to display a detailed directory listing and the disc free space. It takes exactly the same parameter types as CP/M's built-in DIR command: an *ambiguous file specification*, a drive name or no parameter at all. The files matching the given specification are listed on the screen along with their vital statistics. These include the length in CP/M *records* (128-byte units) and the size of the file rounded up to the nearest 1k boundary. If a file it set to *Read-Only*, an R is printed by its name; if a file is set to *System*, an S appears. Both can appear together for the same file.

The final part of the display is the number of bytes free on the disc, in 1k units.

9 The Standard Modules

In this section, we shall attempt to tell you enough about the standard modules to allow you to write programs. The ultimate definition of the standard modules can be found in the definition files supplied with the compiler. Occasionally, additions are made to these modules, so you should read the definition modules to determine the current status of each module.

If you have the disk space, it is useful to keep the definition modules for the standard modules on your work disk so that you can display them in a window when you are unsure of the use of any procedure.

The use of modules instead of inbuilt facilities is both Modula- 2's great strength and its great weakness.

It is its great weakness because there is not yet a standard for what modules should be present. In this implementation, we have chosen to follow the guide given by Niklaus Wirth in his book 'Programming in Modula-2'.

It is Modula-2's great strength because, providing that you receive the sources of the modules, as is the case with our compiler, you can at worst take your modules with you when you port a program to another machine. We are quite happy for you to do this when you port programs you have written using **FTL Modula-2**, though we ask that you do not distribute those modules without our permission except to other purchasers of **FTL Modula-2**.

This means that you can never be held hostage to a particular implementation of Modula-2. Although the greatest source of variation between implementations of Modula-2 is in the Input-Output modules, Input-Output is the greatest area of variation in compilers for other languages too (COBOL is a prime example of this, as anyone who has ever ported a large COBOL system will be glad to confirm). At least in Modula-2, it's under your control.

Also, if the supplied modules do not quite meet your requirements, you can always alter them to suit. If you do this, however, call the module by a new name. This will avoid confusion when you share your programs with other users.

9.1 The Standard Modules are Ordinary Modules (Almost)

You may change and re-compile almost any of the standard modules. The only modules which you must not recompile are the `SYSTEM` module, since the `.SYM` file for this module cannot be generated using a Modula-2 definition module, and the `LOADER` module, since this module is a dummy module used by `Storage` to mark the top of the load module (you can, of course use `LOADER` yourself, it is not special to `Storage`).

Some of the modules provided are a standard part of Modula-2. Some off these, such as `SYSTEM` and `Files`, are low level modules which provide a clean interface to the underlying operating system.

Because these modules are related to the operating system, they are not totally portable. However, the definitions in this compiler are sufficiently close to the original PDP-11 definitions that you should find little trouble in converting from that compiler to this. If you are using both machines, get the M23 compiler for the PDP-11, since it contains the latest changes to the language.

In fact, the filing system of RT-11 on the PDP-11 is more primitive than that of CP/M. As a result, the low level modules in this compiler are more powerful than the original low level modules of the PDP-11.

The *higher level* modules, such as `Streams` and `InOut` are relatively machine independent.

We have noted any deviations of the modules from those presented in Wirth's book in the source code.

9.2 A Quick Tour

Here is a quick guide to the standard modules, by function:

Input-Output

The module `Terminal` supports terminal input-output. `ScreenIO` can be used to draw structured screens.The Modula-2 source to this module is part of the Editor Toolkit. The `Terminal` module treats the terminal as a teletype.

The module `Files` can be used to read and write disk files. If directly implements the BDOS functions of CP/M, so that you must write whole sectors. It provides routines to open, close, create, delete, rename, read and write files.

The module `Streams` is a layer on top of `Files` which supports Input-Output of characters and records, rather than fixed length sectors.

Formatted Input-Output

The modules `InOut` and `SmallIO` provide input-output for formatted information. For example, these routines allow you to write integers as text, rather than in their internal representation. `InOut` uses `Streams` to do the actual input-output. `SmallIO` uses the `Terminal` module for this.

There is a module `RealInOut` which performs formatted input-output of real numbers.

String Manipulation

The `Strings` module provides facilities to manipulate standard (zero byte terminated) Modula-2 strings.

Heap Management

Heap management is used to allocate or deallocate space when you do a `NEW` or a `DISPOSE`. This is provided by the module `Storage`.

Command Line Handling

The `Command` module allows you to pick up the command line. The module will parse it into fields such as file names and options.

Directory Search

The `GetFiles` module allows you to search disk directories for file names specified with a wild card (such as `*.MOD`).

Multi-tasking

The `Processes` module provides the facilities to run multiple processes. Some of the facilities in the Processes module are normally part of `SYSTEM`. We have separated them out from that module because there is nothing in the implementation that cannot be done in FTL Modula-2 with the help of the assembler and so you may as well have the source to hack.

Bit twiddling

There are several modules that allow you to perform low level operations. `Conversions` allows you to convert between various sorts of data. `FastMove` allows you to move blocks of memory. `IntLogic` allows you to perform logical operations in integers.

Direct CP/M Access

The modules `CPM` and `CPMBIOS` allow you to directly access CP/M's BDOS and BIOS. The modules contain data structures for some of the internals of the operating system.

Maths Routines

The `Maths` module provides a number of standard transcendental functions. This module is sometimes called `MathLib0`. The `SOLVE` module solves sets of equations Gaussian Elimination. We included this since in `BASIC`, to invert a matrix you simply write `MAT INV A=B`, and we reasoned that a similar operation should be available in Modula-2.

Others

The `Sort` module provides a general in-memory Quicksort. The `Chain` and `SetUpCall` modules allow you to call other programs. The `Debug` module is used in conjunction with the inbuilt compiler facilities to trace programs.

9.3 Terminal Input-Output: Terminal

Terminal Input-Output is handled with the module *Terminal*.

The `Terminal` module treats the screen as a *glass teletype*. That is, characters written by the module appear on the screen as if being presented on a printing terminal (except possibly for overstrikes!)

The module provides routines to read and write single characters and strings (`Read`, `Write`, `ReadString`, `WriteString`). The procedure `WriteLn` terminates a line; performing a carriage return/line feed on the terminal.

Under CP/M, there are three ways in which characters can be read from the terminal. All three are supported by `Terminal`. The three methods are:

i) Single character input (CP/M function 1)

This method of input is used by the routine `Read` if buffered input is not active. The routine will wait until a character is typed at the terminal. When reading in this mode, no input editing functions are available. However, if the user types `^C`, your program will be terminated by the operating system. The only way to read `^C` is with the `BusyRead` routine described in method iii.

ii) Buffered input. (CP/M function 10)

This method of input is used if the routine `ReadBuffer` is used. Once `ReadBuffer` has been used, subsequent calls to `Read` will read from the buffer instead of reading another character. When the buffer empties, method one is reverted to. When this method is used, the user can use all the normal (command line) editing facilities. For this reason, this method of input is to be preferred. To use this form of input, you should call `ReadBuffer` at the start of each line of input. The formatted text input-output modules (`InOut`, `SmallIO` etc) do this automatically.

`ReadBuffer` takes one boolean parameter. If this parameter is `TRUE`, the current contents of the input buffer are discarded and a new line is read.

iii) Conditional input (CP/M function 11)

This method is used by the `BusyRead` routine. It returns zero if no character has been typed and the character typed otherwise. You should use this routine if you wish to retain control if no character has been typed, or if you want to ensure that every character typed at the console is returned to the program. In the other methods, some characters (such as `^P` and `^S`) may be chewed by the operating system while `^C` will terminate the program.

For example, the editor uses `BusyRead` to read the edit keystrokes.

Note that, as well as doing output to the terminal through Terminal, you can use `Streams` and `InOut`. Those modules support file redirection, whereas `Terminal` does not. However, `Terminal` is more efficient (and smaller).

There is a problem with switching between `BusyRead` and `Read` for character input. When you switch, the last character read by `BusyRead` may be read a second time. This is a problem in CP/M, rather than in this compiler. To overcome the problem, call `ClearCharBuffer` after you finish using `BusyRead` and before calling `Read`. Remember that `Read` is used by the formatted input-output modules as well.

There are several variables in the `Terminal` definition module. The variable `CharRoutine` allows you to *capture* the output of characters to the terminal. This is used by the memory mapped version of the `ScreenIO` module so that it can maintain the current screen position. To use `CharRoutine`, assign the address of a procedure that is to do the character output to `CharRoutine` and set the boolean variable `CharEnabled` to `TRUE`. For example:

```
FROM Terminal IMPORT CharRoutine,CharEnabled;
      ...
PROCEDURE Writeit(ch:CHAR);
      ...
BEGIN (*main program*)
      CharRoutine:=Writeit;
      CharEnabled:=TRUE
      ...
```

BlinkRoutine is called (if enabled by setting BlinkEnabled to TRUE) every time BusyRead is called. It is primarily used to control blinking of the cursor in the memory mapped ScreenIO module, but you can no doubt think of other uses for it.

In addition to the routines provided by Terminal, output to the screen can be performed through the module ScreenIO. This module allows character attributes to be set as well as supporting cursor positioning and other screen control operations.

This module is used by the full screen editor. The main program part of ScreenIO reads in the terminal configuration file defined by the program SETTERM.

Note that the source for ScreenIO is part of the Editor Toolkit.

The ScreenIO module provides facilities to perform a number of screen related operations. The procedure ScreenControl can perform takes a parameter an element of an enumeration Edits. This allows you to perform operations such as clear screen, draw graphics boxes and select display attributes. The procedure Gotoxy is used to position the cursor.

9.4 Low level File Input-Output: Files

This module is based on the Files module of RT-11, but with some changes to reflect the changed operation system. One thing that has not changed, however, is the names of the entry points in the modules, which are taken from the system call names used under RT-11.

Use the routine Lookup to open an existing file. Use Create to delete any existing file and create a new one. The Close routine should always be used to close a file which has been written to, even if the file has not been extended. It can also be used for input files without problems.

It is a good idea to close input files so as to make your programs as portable as possible. For example, under Unix, failing to close input files could cause you to run out of file units, which would cause your program to abort.

Data is read from a file with the routines ReadBlock and SeqReadBlock and written with the routine WriteBlock and SeqWriteBlock. You must always read or write a multiple of 128 bytes. If you attempt to read less than 128 bytes, then 128 bytes will be read anyway. This may prove embarrassing if the code is something like:

```
ReadBlock(f,ADR(Rec),0,SIZE(Rec),reply);
```

and Rec is not a multiple of 128 bytes, since the following variable will be over-written. To overcome this, you must pad out the variable to the appropriate size. The functions SIZE and TSIZE (available from system) will assist in this. The module Streams, described next, should be used if you want to read or write exact numbers of bytes which are not multiples of 128.

Note the use of the function ADR, which gives the address of a variable. This function is required frequently in calls to procedures in Files and also Streams, as these modules use parameters of type ADDRESS. This must be imported from the module SYSTEM:

```
FROM SYSTEM IMPORT ADR;
```

The ReadBlock and WriteBlock routines expect to receive a block number as one of the parameters.

The routines SeqReadBlock and SeqWriteBlock read and write the next sequential block of a file. These routines are not in the original implementation of Modula-2. The RT-11 system has no concept of sequential input-output to disk files.

Other routines in the Files modules are Delete, which deletes a file by file name, Rename, which changes the name of a file, and SetBlock, which repositions a file to a given 128 byte sector.

9.5 Byte Oriented Input-Output: Streams

If you want to read variable numbers of bytes from a file, the Streams module should be used. To use the streams module, you must open a file and then connect it to a stream with a Connect call.

Here is an extended fragment of a module which reads a file and rewrites it using Streams.

```
FROM Files IMPORT FILE,Lookup,Create;
FROM Terminal IMPORT WriteString,WriteLn;
FROM Streams IMPORT Connect,STREAM,Disconnect,WriteChar,
                    ReadChar,EOS,Direction;      ...
VAR   InFile,OutFile:FILE;
      InStream,OutStream:STREAM;
      reply:INTEGER;
      ch:CHAR;
      ... (* open files, connect to streams*)
      Lookup(InFile,'INPUT.DAT',reply);
      IF reply<>0 THEN
           WriteString('INPUT not found');
           WriteLn;
           HALT;
           END;
      Connect(InStream,InFile,input);
      Create(OutFile,'OUTPUT.DAT',reply);
      IF reply<>0 THEN
           WriteString('Disk full');
           WriteLn;
           HALT;
           END;
      Connect(OutStream,OutFile,output);
      (*transfer characters until EOS *)
      WHILE NOT EOS(InStream) DO
           ReadChar(InStream,ch);
      WriteChar(OutStream,ch);
           END;
      (*Disconnect streams, close files*)
      Disconnect(InStream,TRUE);
      Disconnect(OutStream,TRUE);
```

Note that the second parameter to `Disconnect` is `TRUE`. This causes the files to be closed automatically by `Streams`. In the case of input files, it does not matter, under CP/M, if you close the file or not, though you should make a practice of doing so since it can matter under other operating systems, such as MSDOS.

In the case of output files, the file must be closed to ensure that the directory information on disk is updated and that the buffers are flushed.

Programmers who have been using the PDP-11 compiler should note that the third parameter in the `Connect` call is different from the PDP-11 compiler. In this compiler, the third parameter gives the direction (input or output). In the PDP-11 compiler it was a boolean variable which distinguished byte streams from word streams. We used an enumeration so that failure to convert a call from PDP-11 form to this form would give an error at compilation time. Otherwise, you would have to find the error by testing.

You can read and write bytes and words using the procedures `ReadChar`, `WriteChar`, `ReadWord`, `WriteWord`. You can access the current position and set the position of a file using the routines `GetPos` and `SetPos`.

There are two routines to sense the end of an input stream. `EOS` returns `TRUE` if the stream is at either the physical end of file or if it is at a logical end of file, as marked by the end of file character `1ax`. The `PhysicalEOS` routine returns `TRUE` only of the actual end of file has been encountered. Thus, you should use `EOS` for ASCII files and `PhysicalEOS` for binary files.

There are also two routines `ReadRec` and `WriteRec`, which perform reads and writes of arbitrary numbers of characters on a stream. These are not in the standard.

`Streams` has another advantage over `Files`: the connected file need not be directed to a disk file; it can be any of the standard CP/M devices, including the console. This promotes device independence in Modula-2 programs.

To direct output to a device, open a file with the device name as the file name (for example `PUN:`), and then connect it to a stream. The colon must be given. The Files module knows enough about devices to be able to open a file for a device, but it cannot actually perform the input-output to the device. That must be done with `Streams`.

For an example of the use of these routines, look at the code for the `LIST` module. This module also shows how to parse the command line and how to resolve wild card file names.

9.6 Formatted Input-Output: InOut, RealInOut, SmallIO

The modules `InOut` and `RealInOut` work with the module `Streams` to format numbers, and to read numbers, converting them to real, integer or cardinal forms.

These modules use a default input and a default output stream. Routines are provided to change the current streams for input and output.

When a program is loaded, default streams are connected to the terminal. This means that you can use `InOut` and `RealInOut` to write formatted output to the terminal without having to worry about opening files, attaching them to streams and then changing the streams used by `InOut`.

The module `SmallIO` can be used in place of `InOut` if all you want to do is write numbers to the terminal. Because `SmallIO` calls `Terminal` rather than `Streams`, the resulting programs are smaller but you cannot perform file redirection with `SmallIO`.

`InOut` allows you to stack input and output streams (there is a stack for each direction). This allows you to output data to a temporary stream and then switch back to a default stream, without knowing what the default stream is. These facilities are an extension to the standard module. In the released version, the stack is only two levels deep, but you can increase it yourself by changing the array size in the implementation module and recompiling.

For most purposes, two levels will suffice. The first level is always taken by the default connections to the terminal. The second level is available for your own streams.

Let us assume that `OutStream` has been connected to a stream as shown in the example in the previous section. To write some formatted numbers to this stream, the code might look like the following:

```
(*switch from default output stream to OutStream*)
SwitchOutStream(OutStream);
(*write cardinal x in 4 places*)
WriteCard(x,4);
(*write an end of line*)
WriteLn;
(*return to default output*)
PopOutStream;
```

All these procedures must be imported from `InOut` - including `WriteLn`. Recall that, to avoid problems with conflicts with the procedure of the same name in `Terminal`, you can say `IMPORT InOut` and then qualify all references to procedures from `InOut` - `InOut.WriteLn`, `InOut.PopOutStream` etc.

Other procedures in the `InOut` module are `ReadString` which reads a string of non-blank characters, and `ReadLine`, which reads to the end of the line.

`ReadString` skips input until a non-blank, non-tab character is found. It returns all characters up to the next blank or tab character, or up to the end of the line or until the `ARRAY OF CHAR` passed as a parameter is full. The returned string will be zero byte terminated if it shorter than the variable into which it is read.

`ReadLine` reads all characters up to an end of line. If and end of line is encountered before the string is full, the string is blank filled (this makes it easier to read data that is laid out in fixed positions). If the string fills before the end of line is encountered, the rest of the line is skipped.

There are two constants exported from `InOut`; `EOL` and `EosCH`. `EOL` is a character returned by the `InOut` procedure `Read` when it encounters the end of a line. (Do not confuse this procedure with the procedure of the same name exported from the module `Terminal`)

Similarly, calling the `InOut` procedure `Write` with this constant will output and end of line. This makes the handling of ends of lines independent of the actual representation of end of line on a particular machine.

The constant `EosCH` is the character to return when a read is performed and the stream is at end of file.

There are also several variables exported from `InOut`.

The variable Done is set to TRUE or FALSE to reflect the success or failure of certain operations.

In OpenInput and OpenOutput it returns FALSE if the user has abandoned attempting to open a file by not entering a file name.

In Read, ReadString, ReadLine and SkipEOL, it returns FALSE if end of file is detected before the operation is complete. In ReadCard and ReadInt, it returns TRUE if a number is found.

The variable termCH returns the character following a field returned by ReadString, ReadCard or ReadInt. In the case of these last two, it is possible for the character to be numeric! This occurs if including the terminating character as part of the number would cause the number to overflow. Hence, to check for a being number too large on input use:

```
IF termCH IN CharSet{'0'..'9'} THEN
      WriteString(' Number too large');
      END;
```

The variable AlwaysBuffer is normally set to FALSE. If you set it to TRUE, any call to the Read routine in InOut that is directed to a stream that is connected to the console will cause a buffered read of the console. For most applications, you should probably set it to TRUE. In buffered mode, the user can use all the normal CP/M editing keys but none of the characters type is returned to your program until a return is typed. This is why the default is FALSE - it is probably more compatible with other compilers.

Reads from the terminal using ReadString, ReadLine, ReadCard and ReadInt are always buffered.

The variables eolch and ignorech allow you to set the character that is to mark the ends of lines and a character that is to be ignored entirely. By default, carriage return is the line terminating character while line feed is the ignored character. This corresponds to the normal convention on CP/M. Some programs use line feed to terminate lines of text. To process files produced by those programs, you need only assign line feed to eolch.

9.7 Memory Allocation: Storage

The standard module `Storage` allows you to allocate blocks of memory from the heap.

You will recall that the routines `NEW` and `DISPOSE` are mapped to calls to `ALLOCATE` and `DEALLOCATE` by the compiler. These routines must be imported into any module using `NEW` or `DISPOSE`. They need not be imported from the supplied module `Storage`, you can replace that by your own if desired. The name need not even be `Storage`.

In addition to the routines `ALLOCATE` and `DEALLOCATE`, the routine `RELEASE` frees all memory above the address passed as a parameter. This is useful when the heap is used as a stack (that is, the last object allocated is always the first to be deallocated), since it saves having to deallocate every pointer explicitly. However, care should be taken when using `RELEASE` that no pointers are left dangling into the released storage.

There is also a function `FreeSpace` which returns the amount of space available between the top of the heap and the bottom of the stack. This is not quite the same thing as the amount of free space, as the deallocation of space may leave holes in the heap which can be re-allocated. These holes are not included in the returned value. Some other implementations call this function `MEMAVAIL`.

The implementation module for `Storage` imports a variable from the module `LOADER`. This variable is always loaded at the top of the load module. Hence, its address serves as the address for the start of the heap.

Note that the heap works its way upwards from this address, while the stack works its way downwards from the top of the TPA. (The address for the stack is picked up from absolute address 6 when the program starts). When they meet in the middle, your program should fall over.

In the current version of the compiler, heap/stack conflicts are checked for whenever heap space is allocated, but not when a procedure is called. As a result, if you are worried that the stack may overflow the heap (as against the heap overflowing the stack), you should include an explicit check for the condition yourself.

In most programs, there are only a few routines which are called with the stack at its greatest extent, and the checking need only be performed in these routines. This is only likely to be a problem if your routines are recursive.

There is no garbage collection in Modula-2. To do garbage collection, it would be necessary to know where every pointer to every heap object is. When you dispose a heap object, a hole is created in the heap. If this hole is at the top of the heap, the heap is cut back. If it is next to an existing hole, the holes are merged. When a call to NEW is made, the list of holes is searched first. The first hole (if any) which is large enough is used, and any remaining space is retained on the list of holes (the free list).

This simple scheme works well in practice providing that the objects you are allocating are of a small number of distinct sizes. If you are allocating objects of different sizes, it is possible for a hole created by a large object to be allocated to a slightly smaller object. This can result in a small hole which is not big enough for any object, and then that memory is effectively lost until the object on one side or the other is disposed. Obviously, this never happens if all objects are the same size.

9.8 Command Line Processing: Command

The module `Command` can be used to parse the command line used to invoke a program.

In CP/M, a buffer at absolute address 80H contains the text of the command line used to start a program, with the name of the program removed.

The byte at address 80H contains the count of the number of bytes in the string starting at 81H. All characters are converted by CP/M to upper case.

The module `Command` will break this line up into fields. Each field will be identified as a name, a string or an option.

For example, in the command line:

```
Search *.doc 'Mardi' [u
```

`Search` is the name of the program to be run. It will not be returned as it is removed by the CP/M command line processor. `*.doc` is the first parameter returned, and is a name. `Mardi` is the second parameter returned, and is a string. The third parameter (`u`) is an option. Note that the quotes are removed from the string and that the option marker ([or /) is removed from the option.

The module returns a list of parameters in an array of type `Parameter`. Each element of the array contains a variable `Chars` which is the text of the parameter. This string is zero byte terminated so that you can use it with `WriteString` etc.

Here is an example piece of code using the module `Command`. The code reads two file names from the command line and opens the first as an input stream and the second as an output stream.

```
FROM Command IMPORT Parameter,ParClass, GetParams;
FROM Files IMPORT Lookup,Create,FileName,FILE;
FROM Streams IMPORT STREAM,Connect,Disconnect,Direction;
FROM Terminal IMPORT WriteString,WriteLn;
      ...
VAR   Param:ARRAY [1.2] OF Parameter;
      Count:INTEGER;
      reply:INTEGER;
      InFile,OutFile:FILE;
      InStream,OutStream:STREAM;
      ...
      GetParams(Param,Count);
      IF Count<>2 THEN Usage END;
      Lookup(InFile, FileName(Param[1]^.Chars),reply);
      IF reply<>0 THEN
           WriteString(Param[1]^.Chars);
           WriteString(' not found');
           WriteLn;
           HALT;
           END;
      Connect(InStream,InFile,input);
      Create(OutFile,FileName(Param[2]^.Chars),reply);
      IF reply<>0 THEN
           WriteString('directory full');
           WriteLn;
           HALT;
           END;
      Connect(OutStream,OutFile,output);
```

The procedure `VeryQuick` from the `QuickStr` module takes as parameters two streams. The first is opened as an input stream, the second an output stream. If file names are given on the command line, the streams are connected to these files. Otherwise, they are connected to the console.

The procedure `OpenStreams` supports more than two streams and can also return a list of options. This routine can open files or streams and allows default file names to be given. see the definition module for `OpenStreams` for details.

9.9 Directory Search: GetFiles

If a parameter is a file name, and a wild card is allowed, the module `GetFiles` can be used to return a list of all the matching file names. If no wild card is contained within a parameter, `GetFiles` always returns the given parameter formatted as a file name, even though it may not exist on disk. This will work for device names too.

You can use `GetFiles` to produce disk directories of any disk. For example:

```
FROM Files IMPORT FileName;
FROM GetFiles IMPORT GetNames;
FROM Terminal IMPORT WriteString,WriteLn;
VAR   Names:ARRAY[1..64] OF FileName;
      i,j,Count:INTEGER;
BEGIN
      GetNames('*.*',Names,Count);
      FOR i:=1 TO Count-3 BY 4 DO
            FOR j:=i TO i+3 DO
                  WriteString(Names[j]);
                  WriteString(' ');
                  END;
            WriteLn;
            END;
      FOR i:=i TO Count DO
            WriteString(Names[j]);
            WriteString(' ');
            END;
      WriteLn;
```

This fragment lists all the files from the logged in disk in four columns.

9.10 Sorting Data: Sort

The module `Sort` can be used to sort any type of data. The module implements a Quicksort algorithm. While this requires a little more effort than a 'quick and dirty' bubble sort that you could write yourself, the improved running speed will be worth it for all but the smallest sorts. To use the `Sort` module, you must provide a key comparison routine. This takes as parameters the addresses of two records to compare and returns `TRUE` if they are out of order. Its header must be of the form:

```
PROCEDURE Compar(p,q:ADDRESS):BOOLEAN;
```

It is often desirable to sort the names returned by `GetFiles`. Here is some code that will do this.

First, we need a comparison routine:

```
TYPE PFileName=POINTER TO FileName;
     ..
PROCEDURE Compar(a,b:ADDRESS):BOOLEAN;
VAR p,q:PFileName;
BEGIN
      p:=PFileName(a);
      q:=PFileName(b);
      RETURN p^>q^;
      END Compar;
```

Note that, in the example, out of order means that the first value is greater than the second value, so this comparison routine will cause the values to be sorted in ascending order. Changing the relation to be `p^<q^` would cause the values to be sorted in descending order. Note also, that we are using the ability of **FTL Modula-2** to compare strings, which is an extension to the language.

The call to the sort routine (using the names from the example in the previous section) would be:

```
SortRecords(ADR(Names),Count,SIZE(Names[1]),Compar);
```

We pass to the `SortRecords` routine the address of the data to be sorted, the number of records in the data, the size of an individual record (`SIZE(Names[1])`) and the name of the comparison routine. The `Sort` module requires that all the records are the same length. If this is not the case, you can sort an array of pointers to the records, with the comparison routine performing the appropriate dereference to access the actual data. The code would look something like this:

```
TYPE   PData=POINTER TO Data;
       DataArray=ARRAY[1.100] OF PData;
       PDataArray=POINTER TO PData;
VAR    DataPointers:ARRAY[1.100] OF PData;
       ..
PROCEDURE Compar(a,b:ADDRESS):BOOLEAN;
VAR    p,q:PDataArray;
BEGIN
       p:=PDataArray(a);
       q:=PDataArray(b);
       RETURN p^^.Key<q^^.Key
       END Compar;
```

The two arrows are required in `p^^` and `q^^` because the address passed by the sort routine is the address of the entry in `DataPointers`. The first dereference (`p^`) has as its value the pointer to the actual data. The second dereference (`p^^`) is required to get the actual data.

It is sometimes desirable, when sorting a file which will not fit in memory, to create a file of keys and to sort that before re-assembling the file. Each record in this key file will contain just the key and a pointer to the record on disk. This technique is most useful when the records are large compared to the amount of free memory and the keys are small. This method is known as sorting on detached keys.

The `List` module contains an example of the use of `Sort`.

9.11 Converting Between Data Types: Conversions

The module `Conversions` contains a number of routines to convert from one data type to another. Routines are available to split a word into its constituent bytes, and to join them together again, to convert cardinals and integers to strings.

`CardToString` and `IntToString` convert cardinals and integers to strings. The parameters are the value to be converted, the base of the conversion (base must not be greater than 16), the string to receive the result, and a variable which receives the number of characters returned in the string.

The functions `LowByte` and `HighByte` return the low order and high order bytes from a word. Under CP/M, a the most significant byte of a word is second. That is, a value such as 256 will have 0 in the first byte and 1 in the second byte. Hence, the function `LowByte` returns the first byte from a word, while `HighByte` returns the high order byte.

The routine `MakeWord` takes two bytes and combines them to produce a word.

9.12 Calling Another Program: Chain, SetUpCall

The module `Chain` can be used to transfer to another program. Once you have transferred to that program, the image of your current program in memory is destroyed, so if you want to return later, the most you can do is restart the current program from the beginning.

The procedure `LoadAndExecute`, in `Chain`, requires an unopened file control block (fcb) for the file to be loaded as parameter. This can be created with the routine `ConvertFileName` from the module `Files`.

The file control block (fcb) is a type which is exported from the module `CPM`. This is described in the section on that module.

Some small programs exit by performing a return (ret) instruction. To accommodate these programs, the LoadAndExecute procedure places a word of zero on the stack before transferring control to the new program. This will cause a warm boot when the program finishes.

If the file to which you are attempting to chain does not exist, the LoadAndExecute procedure exits to the operating system.

You can share variable between programs by using the /D:xxxx linker flag and ensuring that the modules which are to share data are all loaded in the same order and at the beginning of the program. You must, of course, use un-initialized variables, and you must use the same value for xxxx in both programs. This will ensure that the data is loaded at identical addresses. Another way to share data is to use absolute variables but this is fraught with difficulties such as avoiding the operating system and your own programs.

9.13 Some Low Level Modules: FastMove, IntLogic

There are a couple of low level modules (written in assembler) which access some machine instructions to perform operations which would otherwise be slow.

These modules are IntLogic, which performs logical operations on integer operands, and FastMove which allows you to move blocks of memory using the fast block move instructions LDIR (move down) and LDDR (move up).

The Moveup procedure should be used when an overlapping move is being performed which moves data higher in memory. The Movedown procedure should be used when the data is being moved down in memory. If the two areas do not overlap, it does not matter which procedure you use. For example:

```
FROM SYSTEM IMPORT ADR;
FROM FastMove IMPORT Moveup,Movedown;
VAR   a:ARRAY[1.300] OF CHAR;
      ...
      Moveup(ADR(a[1]),ADR(a[10]),20);
      (* moves a[1] to a[10], a[2] to a[11] etc*)
      ...
      Movedown(ADR(a[10]),ADR(a[1]),20);
      (* moves back down*)
```

After the call to Moveup in the above example, a[1] through a[9] remain unchanged. a[10] has the same value as a[1] and a[20] has the value that was in a[11]. If Movedown had been used for the move by mistake, then a[20] would have ended up with the same value as a[2], since the move would start from the bottom of the array and work up, instead of starting at the top and working down.

Note that the compiler always uses a block move instruction to move blocks of data, so you only need to make explicit calls to this module if the nature of the data to be moved is such that it cannot all be moved by a single assignment statement.

For example, if the size of the area to be moved is not know until the program is run, then you would have to use these move routines.

The routines Searchup and Searchdown in FastMove will search upwards or downwards in memory for a string of bytes.

The routine Swap from FastMove will swap two areas of memory without using any extra memory.

All of the operations performed by routines in IntLogic can also be performed by re-typing an integer as a BITSET, performing the required operation and then re-typing it back.

For example,

```
i:=INTEGER(BITSET(i)*BITSET(3));
```

Does an AND of the variable i with 3. However, this is ungainly.

9.14 The Module SYSTEM

The module SYSTEM is available in every Modula-2 implementation, but its contents will vary from implementation to implementation.

There are some special types and functions which can be imported from the module SYSTEM which break the normal typing rules. These are the types WORD, BYTE, and ADDRESS, and the functions ADR, SIZE and TSIZE.

The type ADDRESS is defined to be of type POINTER TO BYTE. However, it is compatible with all pointer types. That is you can assign a pointer value of any type to a variable of type ADDRESS. and vice-versa. For example:

```
FROM SYSTEM IMPORT ADDRESS;
TYPE pInt=POINTER TO INTEGER;
VAR   pi:pInt;
      pa:ADDRESS;
      ...
PROCEDURE a(VAR p:pInt);
      ...
 pa:=pi;(*assign to ADDRESS*)
 pi:=pa;(*assign from ADDRESS*)
 a(pa); (*use address as parameter*)
```

All of these constructs are valid. Compare this with what would happen if, for example, a variable of type POINTER TO INTEGER was assigned to a variable of type POINTER TO CARDINAL. In this latter case, an incompatible types error would be produced when the program was compiled.

To emphasize this point, if you were to declare ADDRESS yourself:

```
ADDRESS=POINTER TO BYTE;
```

The type would not be the same as that imported from SYSTEM, since it would not be compatible with pointers to other types.

The ADDRESS type is used when you want to be able to deal with blocks of memory without reference to the internal structure. Hence, it is used by Storage when allocating heap space and it is used by Files when reading and writing blocks from disk files. The function ADR, described below, is often used to create pointers for use with ADDRESS parameters.

If you are porting programs from the original PDP-11 compiler, note that the original RT-11 compiler treats ADDRESS as a pointer to WORD. The change is acceptable because the module SYSTEM is used for things which vary from machine to machine.

The type BYTE is compatible with any one byte value, such as a character or a boolean value. BYTE is a one byte cardinal. Hence, it has values in the range 0..255. You can also use BYTE when declaring a cardinal subrange to cause the resulting type to use only one byte of storage:

```
ONCEBYTE=BYTE[0..1];
```

If BYTE was not used here, the resulting type would occupy two bytes, as it would be a subrange of CARDINAL or INTEGER.

The BYTE type is an extension provided by this compiler. It may not be available on other compilers.

Also, if you declare a parameter to be of type ARRAY OF BYTE, it is compatible with all actual parameters. This allows you to pass arbitrary blocks of memory (for example, record structures) to a routine.

Some other compilers may use ARRAY OF WORD for this purpose. That would cause great problems with objects with odd numbers of bytes.

The type WORD is compatible with any two byte value, including pointers. Also, it is assignment compatible with any one byte value. When you assign a one byte value to a variable of type WORD, the more significant byte is set to zero.

The function ADR returns the address of a variable. The result is of type ADDRESS. The ADR function is used with routines such as ReadRec from the module Files to create a pointer to a block of memory. See the section on Files for an example. The memory to which the pointer is being created need not be on the heap.

If you want to do arithmetic on an address, you should use the standard procedure INC:

```
INC(pch,SIZE(text));
```

In previous versions of the manual, we suggested that you should use:

```
pch:=pCHAR(CARDINAL(ADR(text))+SIZE(text));
```

We now suggest that this is unwise because, on a segmented architecture, such as the 8088 large memory model, addresses are not cardinals. They are not even 32 bit equivalents of cardinals. INC for arithmetic on pointers works in all versions of **FTL Modula-2** including the Large Memory MSDOS compiler. On the Z80, either approach will work. In fact, you will find examples of the second form in the standard modules.

Do not confuse the type ADDRESS when used to modify the type of an expression with the function ADR. The difference is that ADDRESS takes an expression as its parameter, but ADR takes a variable.

Hence:

```
ADDRESS(pch);
```

is the address of the character pointed to by pch. But:

```
ADR(pch)
```

is the address of pch itself. In fact: ADDRESS(pch)=ADR(pch^)

The function SIZE returns the size (in bytes) of a variable. The function TSIZE returns the size (in bytes) of a type.

In revision 3 of Wirth's book, the SIZE function became a standard function which did not need to be imported from SYSTEM. This compiler allows you to use it in that way, or to import it from SYSTEM. This maintains compatibility with existing programs.

You can use SIZE and TSIZE to pad out a record to 128 bytes for use with ReadBlock from the module Files.

```
TYPE   Block=RECORD
       Count:BYTE[0..20];
       Entries:ARRAY[0..20] OF INTEGER;
       Filler:ARRAY[1..128-SIZE(Entries)-2] OF BYTE;
       END;
```

Alternatively, we could have declared Filler as:

```
Filler:ARRAY[1..128-SIZE(Count)-21*TSIZE(TEXT)] OF BYTE;
```

9.15 Direct CP/M Calls: CPM, CPMBIOS

The module `CPM` contains definitions for various data structures used by CP/M , such as the file control block, which is used to control reading and writing disk files. As well, it contains a set of mnemonics for calling the BDOS functions.

The module `CPMBIOS` contains mnemonics and routines for making direct BIOS calls. Direct BIOS calls allow you to access the internals of CP/M itself. As a result, they are very powerful but they can be dangerous - you can even corrupt your disks.

Using the BDOS calls directly in your programs will result in programs which are not portable to other operating systems, such as MSDOS. Using direct BIOS calls will result in programs which probably will not run on 'CP/M-like' systems, such as MP/M and Earth Computers Z80 card for the IBM-PC. The FTL Modula-2 compiler, and the supplied modules, do not use direct BIOS calls, so that the compiler and the programs it produces will all run on these 'CP/M-like' operating systems.

If you want to use these modules directly, a book on CP/M internals, such as that by Hughes (Lawrence E. Hughes; System Programming under CP/M-80), is worth getting. There are one or two other books which cover similar material. Unfortunately, most books which claim to teach you CP/M either want to teach you how to use the basic commands, or else want to teach you assembly language, which leaves little space for interesting topics, such as how to read strange disk formats.

9.16 Creating Processes: Processes

This section is not for the faint hearted. Once you get into processes, you are getting into difficult territory. The concepts briefly covered in the next few pages comprise the best part of an entire subject in a Computer Science course.

The `Processes` module supports multi-tasking. That is, it allows you to run several *processes* concurrently.

Of course, as your computer has only one processor, only one process can be running at any one time. However, the computer can be made to share processor time between several processes - it is not necessary for each process to run to completion before the next one executes.

Each process has its own stack and its own program counter. When you stop executing one process in order to execute another, the program counter, registers and stack pointer for the currently running process are saved and those of the process to be restarted are restored. This operation is called a context switch.

The `Processes` module in FTL Modula-2 contains not only the procedures contained in that module in Wirth's book. It also contains a number of procedures and types which Wirth's book expects to find in the module `SYSTEM`. We put them in `Processes` because, in this compiler, the code to implement these procedures could be written in Modula-2 (with some help from the assembler) so it seemed reasonable that the end user should be able to modify them.

These procedures are `NEWPROCESS`, `TRANSFER` and `IOTRANSFER`. These are the basic operations on which all the process operations are built.

9.16.1 The Basic Procedures

The procedure `NEWPROCESS` creates a descriptor for a process. This descriptor is of type `ADDRESS`. In earlier versions of Modula-2, there was a type `PROCESS` which was used for this purpose, but Wirth's recent amendments have replaced this type with the type `ADDRESS`. In the `Processes` module, you will see a type `PROCESS` which is equivalenced to the type `ADDRESS`. We suggest you use this type for process variables in order to maximize portability.

The `NEWPROCESS` procedure is used to create a descriptor for a new process. It requires four parameters:

1 A parameterless procedure which is to be the starting point of the process.

2 The address of a work area which will be used for the process's stack.

3 The size, in bytes, of this work area.

4 A variable of type PROCESS which is to receive the new process descriptor.

To start the new process executing, you must call the procedure TRANSFER. This procedure takes two parameters.

The first parameter is a variable into which will be placed the descriptor for the currently executing process, which is about to be suspended in favour of the new process. The second is the descriptor of the process to be started. These two may, in fact, be the same variable. This allows you to switch between two processes by sharing a process descriptor between them.

For example:

```
VAR    a,wait:PROCESS;
       Work:ARRAY[1.100] OF BYTE;
PROCEDURE A;
BEGIN
       LOOP
            (* do whatever*)
            TRANSFER(a,wait);
            END;
       END A;
BEGIN(*mainline*)
       NEWPROCESS(A,ADR(Work),SIZE(Work),a);
       TRANSFER(wait,a);
       WriteString(' Done');
```

This example shows the use of both NEWPROCESS and TRANSFER. Starting with the mainline, the first step is to set up a process descriptor for the new process, using NEWPROCESS. Next, in the call to TRANSFER, the mainline is suspended, and its state saved in the process variable wait. The procedure now executes until it reaches the TRANSFER statement in the procedure. This saves the state of the procedure in the process variable a and restores the state saved in the process variable wait. This results in the mainline continuing with the statement after its TRANSFER procedure call; in this case the WriteString statement.

So far this looks fairly boring. We have just found a much more complicated way to do a procedure call! However, this is rather different for two reasons:

1 We can run several copies of procedure A simultaneously by doing several NEWPROCESSs and TRANSFERs.

2 If we now do another TRANSFER(wait,a), the process starts executing, not from the beginning of procedure A, but from where it left off. That is, it resumes from the next statement after its TRANSFER statement. This is why we used a LOOP statement; the process will run for ever, being called from time to time to do whatever it does.

The final procedure of these three basic procedures is IOTRANSFER. This procedure is very similar to TRANSFER. It takes three parameters. The first two are exactly the same as for TRANSFER. that is, the status of the current process is stored in the process variable which is the first parameter and the process state is restored from the variable which is the second parameter.

The third parameter is an interrupt address. When IOTRANSFER is called, the normal context switch associated with TRANSFER takes place but, in addition, an interrupt vector is set up. When the interrupt occurs, the current context is stored back into the *second* parameter and the context is restored from the first parameter. This means that the process containing the IOTRANSFER is resumed.

When the interrupt handler is completed, it can return control to the interrupted task by using either TRANSFER or IOTRANSFER to switch contexts again.

Normally, IOTRANSFER only has three parameters but, in the Z80 implementation, there is a fourth parameter. This is a work area in which the Processes module will set up the context switching code for the interrupt.

You can use mode 0, mode 1 and mode 2 interrupts. For mode 2 interrupts, you must set the variable Mode2 which is exported from Processes to TRUE. In the interrupt address is the address of the two byte jump table entry for the interrupt. (Note that it's the address of the particular entry, not the address of the entire table). In modes 0 and 1 it is the address in low memory to which control is transferred when the interrupt occurs.

9.16.2 The Pre-packaged Procedures

As well as the basic procedures, the `Processes` module contains a number of procedures which simplify the use of multiple processes. You should not mix the use of these procedures with those above; either handle all processes yourself using `NEWPROCESS` etc, or rely on the following procedures to do it for you.

A process can be created using the `StartProcess` procedure. This procedure takes as parameters the name of a parameterless procedure and the size of the work area required for the process:

```
PROCEDURE Proc1;
  ...
StartProcess(Proc1,100);
```

This starts executing `Proc1` using a one hundred byte block from the heap as a stack for the procedure.

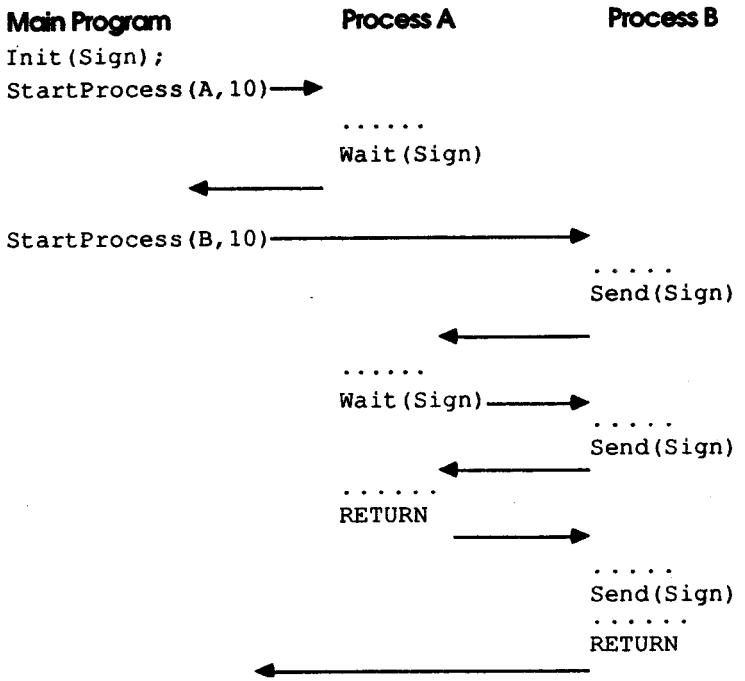
Each process must have its own work space. This is used for the process's stack. It is up to you to calculate the amount of space required for the stack, and to pass that value as a second parameter to `StartProcess`. In the above example, 100 bytes is required for the process work space. The required storage is allocated from the heap.

Processes can be synchronized with the aid of signals, also supported by the `Processes` module. A procedure may wait for a signal, or it may send a signal. Sending a signal causes a process that is waiting for the signal to be restarted.

The distributed version of `Processes` restarts the process which has been waiting the longest. Wirth in his book gives an example of a `Processes` module which restarts the most recently suspended (which is easier). You can change the code in `Processes` to implement a variety of scheduling algorithms. An example may make this clearer. Consider a main program which creates two processes:

```
MODULE Main;
FROM Processes IMPORT StartProcess,SIGNAL,Send,Wait,Init;
VAR Sign:SIGNAL;
PROCEDURE A;
BEGIN
      ..
      Wait(Sign);
      ..
      Wait(Sign);
      ..
      RETURN
      END  A;
PROCEDURE B;
BEGIN
      ..
      Send(Sign);
      ..
      Send(Sign);
      ..
      Send(Sign);
      ..
      RETURN
      END B;
BEGIN (*main program*)
      Init(Sign); (*must be done first*)
      StartProcess(A,10);
      StartProcess(B,10);
      HALT;
      END Main.
```

The execution profile of this module is shown on the next page.



The arrows represent transfers from one process to another. Note that a process may terminate by executing a `RETURN` or by falling out the bottom of the initial procedure.

If a signal is sent but nothing is waiting, it is ignored, and the process sending the signal continues.

You are permitted to use `Wait` and `Signal` in the main program as well as in created processes. Beware the situation where everything is waiting for a signal which can never arrive, because there is nothing which can be run to `Send` it.

9.17 Mathematical routines: Maths, Solve

Two of the supplied modules assist with mathematical applications. The module `Maths` contains a number of standard mathematical functions. The formulae for these have been taken from the book 'Computer Approximation' (HART et al., SIAM Series in Applied Mathematics, John Wiley and Sons, New York), and are accurate to at least 15 significant digits.

The module `Solve` provides a means of solving sets of equations using Gaussian elimination with partial pivoting.

The first parameter to the procedure `Gauss` in module `SOLVE` is a matrix of equations to be solved. The next to parameters give the depth (number of rows) and length (number of columns) of this matrix. To solve a set of linear equations, the length would be one greater than the depth. For example, to solve the equations 2x+y=3;4x+y=6, the matrix would look like:

```
2    1    3

4    1    6
```

The depth is two while the length is three. After calling solve, the right hand side (the last column) is replaced by the solution. The left hand side is destroyed in the calculation.

The matrix is stored as an array of columns. Because the compiler does not yet support the multiple dimensioned open array construct, the size of the columns has been set at 10. Solving systems bigger than this using Gausian elimination is problematical. You will probably find it better to use an iterative method, perhaps combined with sparse matrix techniques.

You can use this procedure to invert a matrix. For example, to invert the left hand side matrix in the above example, start with the matrix:

```
2    1    1    0

4    1    0    1
```

That is, the input matrix consists of the matrix to be inverted followed by the unit matrix. After the call to solve, the result is the inverse matrix.

9.18 String manipulation: Strings

The module `Strings` provides a number of procedures for manipulating strings.

In Modula-2, strings are arrays of characters which have a zero byte - the character `0x` (the `NUL` character) - as a terminator if they are shorter than the variable in which they are stored. For example:

```
VAR   a,b:ARRAY[1.10] OF CHAR;
BEGIN
      a:='0123456789';
      b:='abcdefg';
```

The variable `b` will have a zero byte terminator. The variable `a` will not since the string exactly fits the variable.

There is no length byte in a string. Furthermore, strings are a standard part of Modula-2, so the problem of buggy and incompatible implementations of strings that plague Pascal are not a problem with Modula-2.

The procedures in the `Strings` module are:

`Pos`	search for a substring in a string.
`Insert`	insert characters into a string.
`Concat`	concatenate two strings.
`Assign`	assign a string to another variable. Useful if the variables are different lengths or one of them is an open array, since open arrays cannot be assigned (the compiler cannot tell how big they are).
`StoS`	Copy Strings with optional blank fill.
`Delete`	delete characters from a string.
`Copy`	copy a substring from a string.

The complete definitions of these routines can be found in the definition module for `Strings`. There is one potential point of confusion. Indices into strings start at zero, not one. The index is in fact the number of characters to the left of the first character in the string. This definition is much more sensible than starting at one (since the array index of the strings usually starts at zero) but can cause confusion if you are used to another language.

9.19 Debugging Modula-2 programs: Debug

The module `Debug` is used with the trace flag (`/T`) of the compiler and linker. It will output the trace information for modules which have been compiled and linked with the `/T` flag. The method by which this is done is described in the module.

When you are using program tracing, it is essential that `Debug` be imported into every module that is compiled in trace mode, since the main program part of `Debug` must be executed before any part of any module which is executed with trace. Use the statement

```
IMPORT Debug;
```

to achieve this. You may wish to add procedures to `Debug` to enable you to turn tracing off and on.

10 Memory Layout

The programs produced by this compiler run as ordinary CP/M .COM files. There is no interpreter to be loaded, nor is it necessary to have a support program on disk, as with some Basic compilers.

A program consists of executable code, un-initialized data areas, initialized data areas, a heap and a stack.

The un-initialized data is used for variables that you declare at the outermost level of a module and which are not given initial values. Most global variable fall into this category.

Initialized variables are global variables which are given initial values. So, in the following:

```
VAR    i:INTEGER;
       j:INTEGER=2;
```

The variable `i` is allocated in the un-initialized variable area while the variable `j` is in the initialized data area.

The stack is used for variables declared in procedures. It also contains the link information to support procedure calls and returns. The stack always grows from higher addresses to lower. That is, it grows downwards.

The heap is whatever memory is left. If normally starts after your code and extends upwards to the stack. As the bottom of the stack is not in a fixed place, there is memory which can be part of the stack at one time and part of the heap at another time. Of course, it cannot be part of both simultaneously.

There are a number of flags you can use to affect the way memory is organized. Normally, only the /D flag is used. We leave discussion of the remaining flags to the section on running the compiler. The default condition is discussed here.

The program starts at the address 100h in memory and works its way upwards. If no /D flag is used at link time, the data areas for variables declared at the level of a module (that is, the static variables), are contained in the .COM file. If a /D flag is used, then only the variables which have been given initial values are part of the .COM file.

If no /S link flag is used, and the /D flag with no explicit start address is also not used, the stack works down from one less than the address given in address 6 of memory. Under CP/M, absolute address 5 always contains a jump to the start of BDOS, so address 6, which is the address field of this instruction, points to the bottom of the operating system.

If the /D flag is used without an explicit start address, then space for the static uninitialized variables is allocated from the top of available memory, as determined from absolute address 6, and it works downwards. The stack then starts from below the static data.

When you use a debugger, the value at absolute address 6 is altered to reflect the presence of the debugger. When a debugger is present, the address is actually a jump to the bottom of the debugger. Hence, it is possible to run programs produced by this compiler under a debugger (such as DDT). However, if you use the /D flag with no parameters, the static data will have been set up at starting at the address originally given at absolute address 6. As a result, the program will corrupt the debugger.

If you try to run a program compiled with the /D with no parameters from within a SUBMIT file on CP/M Plus the program will destroy the SUBMIT RSX.

This problem is overcome by giving an explicit starting address in the /D flag, or by omitting the flag altogether.

The heap works its way up from the top of your program. If you have placed your data at an explicit address, then it starts from above the data.

Every program requires a display. The display is an area of memory which is used to retain pointers to the start of areas on the stack called *activation segments*. Whenever you call a procedure, an activation segment is created for the procedure. If the procedure has any local variables, or any parameters, or if it returns a value (that is, it is a function), then the register IX, an internal Z80 register, points to the activation segment for the duration of the procedure.

Suppose procedure A has another procedure B nested within it. Then, when B is executing, the IX register will point to the activation segment for B. If variables local to A are to be accessed in B, then the activation segment for A must be accessible. The display is used to save the activation segment register for A.

Each procedure has a *lexical level*. This is simply the number of procedures inside which it is nested plus 1. So that A would have lexical level 1, B level 2, a procedure nested inside B would have level 3, and so on. Another procedure following A, not nested inside it, would once again be at level 1. For each level, there is an entry in the display.

If a procedure has *Up Level Addressing*, that is, if a procedure has procedures nested within it, and those procedures access variables in the current procedure (including parameters), the display is modified to reflect the location of the activation segment. To do this, the current value of the required entry is saved on the stack and replaced in the display by the new value of IX.

The required location is determined by the nesting level of the procedure.

The WITH statement works in a similar fashion. When a WITH statement is entered, the pointers to the variables in the WITH header are placed in the display. If a WITH is in the main program part and is not nested in another WITH statement, it will have lexical level 1. If it is in a procedure at level 1, such as A it will have level 2, and so on.

Note that there are several optimizations performed in the call of procedures.

i) If a procedure has local variables, an activation segment must be created. This means that space must be taken from the stack for the local variables.

ii) If the procedure has local variables, or parameters, or returns a result, IX must be set up.

iii) If the procedure has up level addressing, the display must be updated.

The compiler only performs those operations which are required, so for a parameterless proper procedure, no entry or exit code is generated (except, of course, for a RET instruction).

When a procedure is called, the previous value of IX is saved on the stack. Assembly language routines must preserve the value of IX across calls. This is the only register which need be preserved.

The parameters for a procedure are evaluated at the point where the procedure is called. They are placed on the stack before the call is made. If the procedure returns a value (i.e. it is a function), then space is allocated on the stack for the result before any of the parameters are evaluated.

Parameters are pushed onto the stack in left to right order, so the left most parameter is furthest up the stack.

Except for one byte parameters, the number of bytes taken up by a parameter is equal to the size of the parameter. For one byte parameters, two bytes are used. The significant byte is the byte with the lower memory address.

Open array parameters always require 6 bytes of stack space. The first two bytes (higher in memory) contain the address of the parameter. The next two contain the number of elements in the array. The final two bytes give the number of bytes contained in the array. If the parameter is a value parameter, a copy is made by the called routine, not the calling routine. The actual variable will therefore be at the bottom of the stack below the activation segment.

Open array parameters are copied onto the stack by a procedure which is called at the start of the procedure code. This procedure call is followed by a number of bytes of data which give the position on the stack of the parameters that are to be copied.

If you disassemble this code, it will often produce illegal or unintelligible instruction mnemonics, and it can cause the debugger's disassembler to lose synchronization with the instruction stream.

10.1 Real Number Formats

Each real number requires 8 bytes of memory. The first byte is the exponent with a bias of 128. The remaining seven bytes are a twos complement hexadecimally normalized mantissa.

Because hexadecimal normalization is used, the exponent gives a power of 16, not a power of two. The mantissa is normalized if the top nibble is not zero or not hex F, or if the top bit of the second nibble differs from the top bit of the first nibble.

As a result, the exponent lies between 16^127 and 16^-128, which is about 10E152 and 10E-153 while the mantissa has about six and a half bytes of significance, which represents about 15 decimal places.

Zero is represented by 8 bytes of binary zero.

For example:

1.0	is represented as	81 08 00 00 00 00 00 00
2.0	is	81 10 00 00 00 00 00 00
-1.0	is	81 F7 FF FF FF FF FF FF

Because of the high precision of the real numbers in **FTL Modula-2**, some of the operators do not execute as quickly as in some other compilers with lower precision. Also, in the `Maths` module, more terms are needed to calculate the standard functions to the required accuracy. As a result, the compiler will not run floating point benchmarks as quickly as some other compilers. On the other hand, you can have more confidence in the results that it does produce.

10.2 Set Formats

Sets are stored as bit patterns. There is one bit for each possible value in the set. This bit is a one if the value is in the set and a zero otherwise.

Sets are stored in correct order. They are not stored 'backwards'; the first value occupies the most significant bit of the first byte of the set. Here are some examples of `BITSET`s which demonstrate this:

```
CARDINAL({})=0H
CARDINAL({0})=8000H
CARDINAL({7})=100H
CARDINAL({15})=1H
CARDINAL({0..14})=0FFFEH
```

11 Hints for Efficient Programs

Here are a few ways in which you can make your programs run faster. It is best not to worry about these notes until you have got your program working, as some of these techniques are in direct opposition to good programming style.

i) Use static or local variables

Recall that the variables at the global level of a module are always static (even in a nested module). The Z80 can access static variables several times faster than stack variables.

ii) Make array element sizes a power of two. The compiler performs multiplies by powers of two by repeated adds (which are faster than shifts), rather than using the general multiply routine.

iii) Remove local variables and parameters. Parameterless, variable-less proper procedures are very efficient, since there is no need to set up the stack frame pointer or the display and no space for local variables needs to be allocated. The only overhead compared to having the code in-line is the `call` instruction and the `ret` instruction.

iv) Avoid up level addressing. This saves having to update the display. Up level addressing occurs when a procedure nested inside a second procedure accesses variables local to the second procedure.

v) Use `WITH` statements for records accessed by pointers. Avoid `WITH` statements for global variables. Subfields of a global (simple) variable can be accessed faster directly than they can be accessed through a `WITH` statement.

vi) Use arrays of records rather than separate arrays of the elements of the record. Combined with a `WITH` statement, this can save multiplications.

vii) Use `VAR` parameters rather than value parameters for parameters of more than two bytes wherever possible. Actually, this is a trade off. It takes longer to access a `VAR`

parameter, since the pointer to the parameter must be loaded, but less time to set it up for the call. Access to reference parameters inside a `WITH` for the parameter are as fast as references to a value parameter.

viii) Keep total size of local variables in a procedure small. If an array is required, declare it last. This allows one byte offsets from the stack frame pointer to be used.

On the Z80, only 8 bit offsets are available so that compiler has to change the stack frame pointer to get at variables which are not in the first 128 bytes. This slows things down markedly.

ix) Assign an entire structured object rather than assigning individual elements. The compiler will then be able to use a block move instruction (such as the Z80's `LDIR` instruction) for the entire move.

x) Remove as much constant expression evaluation as possible from loops. This compiler performs no optimization, so any optimization you can do yourself will be useful.