FTL Modula-2 Z80 Advanced Programmer's Kit

Introduction

This Advanced Programmer's Kit contains a number of items which advanced programmers will find useful (i.e. advanced is intended to refer to the users - not the kit!). These are:

i) A new linker which allows you to trim unwanted fat from modules and to create overlayed programs.

iii) An Overlayer. This allows programs to have more code than will easily fit into memory at once. Combined with the trimmer/ linker, this means that you can write much large Modula-2 programs.

iii) The source code to the trimmer and overlayer.

We also have available a multi-tasking kernel, which will not run on many systems, since it requires regular clock interupts that the user's program can trap. It will not work on the Amstrad CP/M systems because of the memory map when interrupts occur. If you are designing an embedded system and would like further details of this please contact us.

To use the new linker you must use `ML.COM` and `M2APK.OVR` from your Advanced Programmer's Toolkit disk.

The Overlayer

The overlayer consists of some features in the linker together with a stand-alone program MLOVLD and the overlay loader (OVERLAYE.REL). To produce an overlayed program you link using a special from of command-line and then run MLOVLD.

The source code to OVERLAYE and MLOVLD are supplied. The overlay assembly routine (OVERLAYE.ASM) may need some changes to suit your application. In particular, the version that is supplied makes the following assumptions:

i) It is assumed that the overlays will not occupy more than 32K bytes when loaded into the overlay file. To increase this, increase the value of the equated constant extents near the beginning of the assembly language source in OVERLAYE.ASM. You will also need to add extra fcbs and add extra entries to the fcb pointer table (at label fcbpnt). To create new fcbs, copy the existing fcb2, change the label to fcb3, fcb4 etc and set the extent number in the fcb to 3, 4 etc. Add new dw directives to the overlay pointer table (fcbpnt).

   You need an fcb for every 16K of overlays.

ii) The name of the overlay file is assumed to be OVERLAY.OVR. You will probably want to change this to reflect the name of your application. To do this, change the name in each fcb and also in the file not found error message.

The routine must be called OVERLAYE.REL when the link is performed - this name is hard coded into the linker. The overlayer is not a conventional module in that it has no entry points. Rather, control is always transferred to it at the beginning of the module. Don't put any data at the beginning of OVERLAYE.ASM, or it will be executed! You can, of course, put the data elsewhere in the module.

In order to create overlays, you have to provide the linker with quite a lot of information. While it would have been possible to have the linker calculate this information itself, the resulting linker would have been larger and might have imposed severe restrictions on transient program area size.

To link an overlayed program, the following information is required:

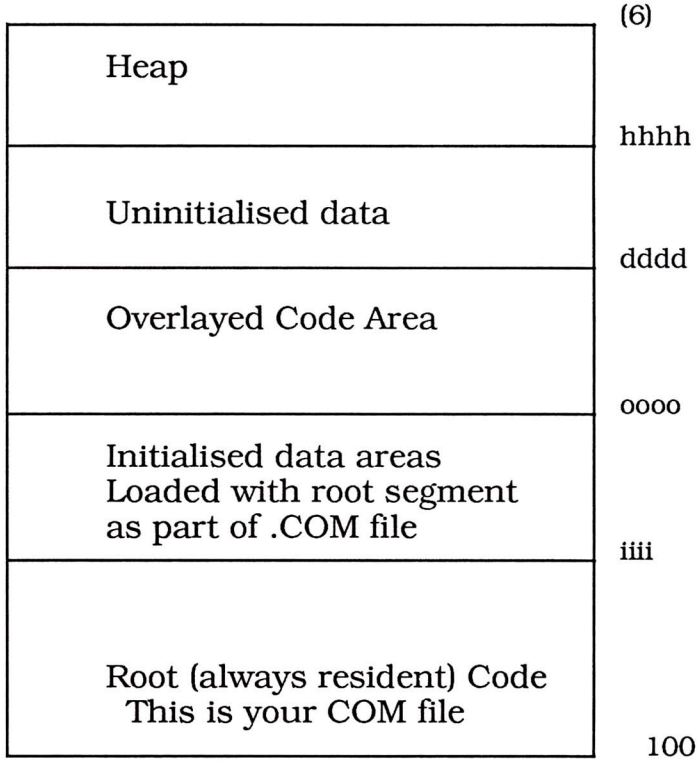
i) The names of the modules that are to be placed in each overlay.

ii) The start address for the initialized data.

iii) The start address for the un-initialized data.

iv) The start address for the overlay area.

An overlayed program uses memory as shown in the following diagram:



The values hhhh, dddd etc are supplied by you to the linker. They are:

hhhh	Start (bottom address) of heap
dddd	Start of uninitialized data
oooo	Start of overlay area
iiii	Start of initialized code area

The linker command line syntax looks like:

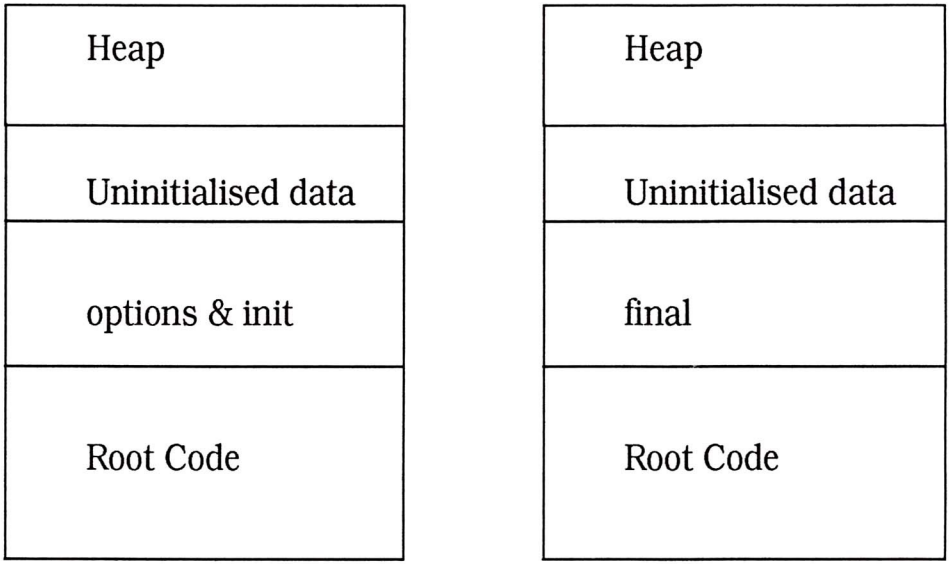
```
ML main/H:hhhh/D:dddd/I:iiii

(:oooo,ovl1mod1,ovl1mod2,...|ovl2mod1,ovl2mode2,...|...)
```

For example:

```
ML Fred/H:8000/D:7a00/I:7000(init,options|final)
```

The modules in the brackets are the modules to be loaded into overlays. More than one module can be loaded into an overlay - the individual overlays are separated by vertical bars (|) so that, in the example above, `init` and `options` are loaded into overlay 1 while `final` is loaded into overlay 2. Also, `init` and `options` are loaded together - they are not alternatives to be loaded one at a time. That is, at any one instant, memory looks like either:



There are some restrictions on the placement of modules in overlays:

i) A module in one overlay cannot call a module in another overlay. (The call will work but when you try to return to the caller, havoc will reign because the overlay caller only interposes itself on the calling side, not the return side) You could make this work by modifying `OVERLAYE.ASM`. See below for some discussion of this point.

ii) Any module that is part of an overlay must be referenced from a module that is in the root segment. Actually, this is not too much of a restriction since any module which is called inside an overlay that has not already been loaded will be included in the overlay automatically. The reason for this restriction is that the root segment needs to know how many entry points exist in each overlayed module so that the required linkage tables can be built.

In other words, you should only give the names of the modules that are to be in overlays that can be called from the root segment.

If this is not observed, you will get the error message

```
**ERROR - module modulename not imported into root segment
```

To fix the problem, just add an IMPORT modulename statement to a module in the root segment. Note that the data for an overlay is not overlayed. That is why you have to specify an area for the initialized data - otherwise, if it was loaded as part of the module, it would be re-initialized every time the module was called. (Actually, you can achieve this by not giving the linker an address for the initialized data - this will cause it to be loaded into the overlay file.)

How the overlayer works

The core of the overlay system is the assembly language module OVERLAYE.ASM. This module is called whenever a routine in an overlay is to be called. It is called with a group number in the A register and an address in the HL register.

The group number is the number of the overlay that contains the routine to be called. The address in HL is the address to jump to to enter the routine. At the start of each overlay is a table of jumps.

At the end of the root segment, the linker builds the following structures:

i) The module OVERLAYE.ASM. This is the overlay handler. It contains the file control blocks for the overlay file, and the code to load and execute overlays.

ii) For each label in each overlayed module, a code stub is created which sets up some parameters in the A and HL registers and then hands control to the start of the OVERLAYE module. Whenever a reference is made to this label, this code is executed. The code takes the form:

```
ld a,ovlno
ld hl,offset
jp overlaye
```

Where `ovlno` is the number of the overlay that contains the module and `offset` is an offset from the start of the module to a jump to the actual label in the overlay. That is, this is an index into the table that is built at the start of the overlay, described next.

At the start of each overlay, the linker builds a table of jumps. Each jump jumps to an entry point in a module loaded into the overlay. Hence, each jump is three bytes in length.

Loading the overlay file

Before the program can be executed, the overlays must be loaded into a single file using the program `MLOVLD`. To create the overlay file, run `MLOVLD` and enter the name of the root segment of the program being overlayed.

Execution of main program parts

Modules in overlays can have main program parts. These are executed when the **program** is loaded. They are not executed every time the **overlay** is loaded.

The normal execution order of main program parts is not observed. Rather, the main program parts in the overlays are always executed before the main program parts in the modules in the root segment.

This means that if a module in an overlay calls a procedure in a module in the root segment, that module may not have been initialized yet. Hence, you may need to modify the modules to overcome this limitation.

An example program

Here is a simple example of an overlayed program. The program has two overlays which simply print a message.

```
MODULE TestOvl;
FROM Terminal IMPORT WriteString,WriteLn;
IMPORT Ovl1,Ovl2;
BEGIN
       WriteString('overlay test ');WriteLn;
       WriteString(' calling ovl1 ');
       Ovl1.Fred;
       WriteString(' calling ovl2 ');
       Ovl2.Fred;
       WriteString(' done');WriteLn;
       END TestOvl.

DEFINITION MODULE Ovl1;
PROCEDURE Fred;
END Ovl1.

IMPLEMENTATION MODULE Ovl1;
FROM Terminal IMPORT WriteString,WriteLn;
PROCEDURE Fred;
BEGIN
       WriteString(' in ovl1 fred ');WriteLn;
       END Fred;
BEGIN
       WriteString(' in Ovl1 main line');
       END Ovl1.


DEFINITION MODULE Ovl2;
PROCEDURE Fred;
END Ovl2.
```

```
IMPLEMENTATION MODULE Ovl2;
FROM Terminal IMPORT WriteString,WriteLn;
PROCEDURE Fred;
BEGIN
       WriteString(' in Ovl2 fred ');WriteLn;
       END Fred;
BEGIN
       WriteString(' in Ovl2 main line');
       END Ovl2.
```

The first step is to compile all the modules in the usual way. Next, we link them to produce the overlay structure.

```
ml testovl/d:8000(:4000,Ovl1|Ovl2)

FTL Modula-2 APK Linker V1.28 Copyright (C) Dave Moore
1986

creating TESTOVL.COM

TESTOVL 0103 TERMINAL 017e SYSTEM 04c7 CPM 08b8

OVERLAYE 8d9

main program linkage

TERMINAL 04aa TESTOVL 0103
Data Size : 008f
Code Size : 0b28
Data in Code: 0000
Top Address : 80a3

creating C:TESTOVL.000

OVL1 4009
```

```
main program linkage

OVL1 401c

Data Size : 008f
Code Size : 4052
Data in Code: 0000
Top Address : 80a0

creating C:TESTOVL.001
OVL2 4009

main program linkage
OVL2 401c

Data Size : 008f
Code Size : 4052
Data in Code: 0000
Top Address : 80a0

link complete
```

This command loads the overlays at 4000h with the data at 8000h. The next step is to load the overlays (`TESTOVL.001` and `TESTOVL.002`) into the overlay file:

```
A>mlovld
Overlay load program
Copyright (C) Dave Moore, FTL Modula-2 November 1987
Enter name of file being overlayed :testovl
Processing testovl.000 overlay number 1 Length 1
Processing testovl.001 overlay number 2 Length 1
```

We can now execute the program:

```
A>testovl
in Ovl1 main line
in Ovl2 main line
overlay test
calling Ovl1
in Ovl1 fred
calling Ovl2
in Ovl2 fred
done
```

Of course, to run your completed program you only need TESTOVL.COM and OVERLAYE.OVR.

The Trimmer

The trimmer allows you to link only those parts of modules which are actually accessible. This results in a .COM file which is smaller than it otherwise would be. For example, trimming un-used code out of the Trimmer program described below reduces its size by thirty percent.

Producing a trimmed file requires three steps:

i) The linker is run with the gather flag (/G). This produces a file with the extension .LRL which contains lists of all the labels, by module, that are referenced in the program. When the linker is used with the /G flag, no .COM file is produced.

ii) The program Trimmer is run. This program takes as input, the .LRL file produced by running the linker with the /G flag and produces as output a file with the same base name but with the extension .TRM This file contains, by module, a bitmap of all the referenced labels.

iii) The linker is run again, this time with the excise flag /E. This produces a .COM file with only the required portions retained.

So, to link a trimmed version of the program ME, the following commands would be issued:

```
ml me/g

trimmer

Enter name of program to trim:Me
```

(here, you see a variety of output from the trimmer. At the end is a list of modules followed by the labels that are accessible in the modules)

```
ml me/e
```

Trimming Overlayed Programs

An overlayed program can be trimmed. To do this, use the /G flag as normal and then use the /E flag during the overlay link. Do not specify the overlay structure while performing the gather (the /G) link.