0.1 Split code for PICs

This text is to be used in conjunction with the tutorial in section 2.5 of the PIC compiler manual. Its purpose is to provide an example of the process as outlined in the PICC Manual for the creation of split code. Explanations or further information on certain sections of this text can be found in the manual.

Please refer to the following C source (and header) files. The processor used as an example in the PIC Manual tutorial is the PIC17C756 - as is this example. The files that are part of this example include :-

`fixed.c`	code for the fixed internal memory
`r_main.c`	main replaceable code for the external memory
`r_ext.c`	replaceable code for external memory
`r_ext.h`	header file for `r_ext.c`
`runsplit`	batch file for the linking, `XSTRIP` & `OBJTOHEX` processes

These files are set up to run the PIC compiler found in the standard directory: C`:\HT-PIC\`. If this is not where you compiler is located, you will need to edit these file to specify the location of the library files used.

Within `fixed.c`, a call is made to the function **`twice()`** which is defined within this file as well as calls to the library functions **`strcpy()`** and **`strlen()`**. The other two functions defined in this file **`inc()`** and **`dosomething()`** are called by the external code. The interrupt service routine defined within this file, **`isr10()`**, calls an ordinary function defined in the external code placed at a specific address (4200h) by the linker.

0.1.1 Creating the Object Files

As outlined in the PICC Manual's tutorial, first compile the files to ensure that the program works correctly and fix any errors. Once this has been completed, we can then create the object files.

This is done for both the internal and external files using the `PICC` driver. Include all the options as required (for example optimizers). In this example, we are using the MPLAB IDE to run the code, hence the `-G` and `-FAKELOCAL` options are used to generate source-level debugging information and MPLAB-specific debugging information, respectively.

```
PICC -17C756 -C -G -FAKELOCAL fixed.c r_main.c r_ext.c
```

The next step is creating a file required by the internal code to prevent local variables and parameters from being overlapped in memory. Notice in fixed.c that the functions **`inc()`** and **`twice()`** are never called. If this file were to be linked in the usual way, the parameters and local objects (both these functions only have parameters in this example) would all be overlapped as the linker would believe that they are never actually executed. Although nothing in the fixed code calls these functions, code in the external space may. The file callgrph.as contains the necessary assembler directives to convince the linker that these functions have been called. You should include a **`FNCALL`** directive for any functions in

your fixed code that are *only* called by the replaceable code. You will need a **GLOBAL** directive for any symbol referenced in this file. Compile this file using a command like,

```
PICC -17C756 -C callgrph.as
```

0.1.2 Creating the Symbol-Only Object File

The symbol-only object file is to used to supply the location of symbols used in the internal code that are required by the external code.

The first step is to create the initial linker command file,

```
PICC -17C756 -MF_SYM.MAP -L-F -L-I -V fixed.obj callgrph.obj > f_sym.lnk
```

The above command produces a linker command file, `f_sym.lnk`, which will be modified to suite our application. The option `-M` produces a map file, `f_sym.map`, whereas the `-F` and `-I` after the `-L` linker option tells the compiler to create a symbol-only object file and ignore any undefined symbols, respectively. The next step is to modify the generated linker command file. The PICC Manual Section 2.3.5 explains how to modify the linker file. In this example we have modified the output object filename to be `f_sym.obj`. Removal of the excess lines as well as the `HLINK` command at the start is also required.

Linking the file with the following command produces the `f_sym.obj` object and `f_sym.map` map files.

```
HLINK < f_sym.lnk
```

On inspection of the UNUSED ADDRESS RANGES in the map file `f_sym.map` (this file is overwritten later, but has been included as `forg_sym.map` for reference here), we can determine the amount of RAM that the fixed code requires for all the memory banks. Our example shows that only BANK0 is used. The default memory range available in bank 0 is from 1Ch to FFh. (Look at the -ABANK0 linker option in this file.) The unused BANK0 memory extended from 28h to C5h (see the UNUSED ADDRESS RANGES in this file). This means that the address range 1Ch to 27h was used (Ch locations) and from C6h to FFh was used (3Ah locations). The total memory required is 46h locations. We will now adjust the linker options so that the linker will allocate all the memory at the lower end of the available bank 0 range from 1Ch to 61h. (1Ch + 46h - 1 = 61h) The upper range of this memory will then be available for the replaceable code. The memory allocated to the internal code is changed in the linker file to reflect this. The new bank 0 memory range available to the linker is specified by the options:

```
-ABANK0=001Ch-0061h
```

The next step is to localize certain global symbols from the symbol-only object file. The reason for this is that we want to linker to re-link the code/data associated with these symbols rather than using the symbols that were defined in the fixed code. Localizing them tells the linker that it cannot use any

globally defined symbols that it may have seen and that it will have to search for a new definition. For example, the copy_data routine is already defined by the internal code to initialize variables that that code uses. But this code will be using the address range applicable for the fixed code and does not know about variables defined elsewhere. To force the linker to link in another **copy_data** routine specific for the replaceable code we essentially undefine the symbol by localizing it. If there are any routines defined in the fixed code that you do not want the external code to use then include their name here. You must then ensure that you provide the definition for the routines in the external code. The XSTRIP utility can be used as shown.

```
XSTRIP -start,_exit,intlevel0,intlevel1,copy_data,clear_ram,
copy_bank0,clear_bank0 f_sym.obj
```

0.1.3 Creating the Absolute Object File for the Fixed Code

Start by copying the symbol-only linker file and renaming it, in our example it has been copied and renamed to f_abs.lnk. The -F and -I options are removed. The map, object and symbol files have been changed to f_abs.map, f_abs.obj and f_abs.sym, respectively (i.e. options -Mf_abs.map -Of_abs.obj -H+f_abs.sym).

This file is then used to link the relevant output files.

```
HLINK < f_abs.lnk
```

0.1.4 Creating a Modified Run-Time File

As described in the PICC Manual, copy the picrt66x.as and sfr.h files from the SOURCES subdirectory of your distribution. (You can rename the run-time file so that it is not confused with the standard run-time module - in this case it has been renamed to extrt.as). Now, within this renamed run-time file, replace all references to the symbol **_main** to the assembler name of the external main function. In our example, the main function of the external code has been called **ext_main()**, hence all references to **_main** would be replaced with **_ext_main**.

0.1.5 Creating the Absolute Object File for the External Memory

Copy and rename the linker file used for the absolute object file (in our example, the f_abs.lnk has been copied and renamed to replace.lnk). The output map, object and symbol files are changed to replace.map, replace.obj and replace.sym, respectively (i.e. options -Mreplace.map -Oreplace.obj -H+replace.sym).

As the internal code only used the RAM in the addresses 001Ch - 0061h, we can allocate the remaining addresses to the external code using the following linker option.

```
-ABANK0=0062h-00FFh
```

The ROM settings are changed so that only external memory is defined.

```
-ACODE=4000h-5FFDh
```

The ROMDATA class is changed so that it only uses the external memory ROM space of 8000h - BFFFh.

```
-AROMDATA=8000h-BFFFh
```

Change the init psect's address.

```
-pintcode=08h,powerup=00h,init=4100h,end_init,clrtxt
```

(Although this option places intcode and powerup at internal locations, there should be nothing placed within either of these two psects by the external code.) Now, remove internal memory object files and replace them with the external memory object files. (The files removed were fixed.obj, callgrph.obj - replaced by r_main.obj and r_ext.obj) The symbol-only fixed code object file is added to the list (f_sym.obj). The modified run-time module is to replace the run-time object, hence, the reference to picrt714.obj is replaced with the extrt.obj file.

Finally, the psect isr10, which contains the external interrupt, needs to be positioned at the absolute address of 4200h. The following is added to the linker command.

```
-pisr10=4200h
```

This is then linked to produce the absolute object file for the external memory.

```
HLINK < replace.lnk
```

0.1.6 Creating HEX Files

The OBJTOHEX application is used to create HEX files from the absolute object files. The conversions are as follows.

For the internal code,

```
OBJTOHEX -I -16,2 f_abs.obj fixed.hex
```

For the external code

```
OBTOHEX -I -16,2 replace.obj replace.hex
```

2.5.2.10 Creating COD Files for Debugging

The COD files are used for debugging on some emulators/simulators, e.g. MPLAB. The CROMWELL utility is used.

```
CROMWELL -F -M -P17C756 fixed.hex f_abs.sym replace.hex replace.sym -ocod
```

This produces a fixed.cod file only. For this example, the MPLAB IDE was used to run the code. As the HEX files must be loaded into memory, it is recommended that a copy of the fixed.cod file is made

in the name of `replace.cod` so that it does not matter which HEX file is loaded first, as the COD files exist for both HEX files.

0.1.7 Checking Results

Check the following by viewing the map files.

a) Ensure that the addresses of all the fixed code are in internal memory and that the addresses of RAM symbols defined by the fixed code are within the RAM space allocated.

The PIC17C756's internal memory is from 0h - 3FFFh and from the CODE class listing, all the code is within that range. (In our example, the code is listed in locations 00h-16h, 1EF1h-2011h.

b) Ensure that the addresses all the replaceable code are in external memory and the addresses of RAM symbols are in the space allocated.

Viewing `replace.map` shows that the code is placed from address 4000h onwards, which is the external memory location. Note that all the psects linked into the ROMDATA class (e.g. `cstrings` and `idata_n` psects) are placed at address 10000h. Although we specified that the linker place these objects in the ROM space 8000h-BFFFh (`-AROMDATA=8000h-BFFFh`). This is due to the fact that the compiler can place two data bytes in each ROM word location. The address used to refer to ROM data (as opposed to code) is a byte address that is twice the corresponding word address.

c) Ensure that **`auto`** and parameters areas for each function are not at the same address, e.g. **`?_ext_main`** is at 0064h and **`?a_ext_main`** is at 0067h. If a function does not have parameters then these addresses will be the same.

d) Ensure that functions that are active at the same time and have **`auto`** or parameter areas do not overlap. e.g. The **`ext_main()`** routine calls the functions **`multiply()`** and **`strcat()`** (in addition to others). The **`auto`** and parameter addresses for **`ext_main()`** and **`multiply()`** do not overlap, nor do the addresses for **`ext_main()`** and **`strcat()`**. However, note that **`?_multiply`** and **`?_strcat`** are the same since **`multiply()`** never calls **`strcat()`** and vice versa.

e) Ensure that the values assigned to the entry routine and external ISRs are as specified.

Looking through the `replace.map` file, the **`_start`** and **`_ext_isr10`** routines are at 4100h and 4200h respectively.

f) Ensure that symbols that are shared are have the same value in the map file for both the fixed and replaceable code.

In our example, this would be the function **`dosomething()`**. The symbol **`_dosomething`** is at 005C in the `f_abs.map` file as is in the `replace.map` file. The same can be said about the library calls to **`strlen()`**, **`strlen()`**, **`inc()`** and **`twice()`**.

g) Note in `replace.map` that some symbols in the Symbol table are defined in a psect called (`abs`). This indicates that the symbol already had a value assigned to it when linking was performed. Essentially this means that the symbol was defined in the fixed code and was present in the symbol-only object file that was linked with the replaceable code. Check to ensure that routines or symbols that should be referenced in the replaceable code are not using any similar definitions in the fixed code.