Miscellaneous Notes

Packed source format

To allow for ease of navigation and support for global find and replace, I usually edit none C files as a single source file. This format also makes it easy to split a file into multiple parts e.g. as PL/M object file boundaries are detected.

A packed file is is simply made up of a concatenations of multiple text files with each file preceded by

```
^Lfilename
```

Where ^L is the character control L and filename is the source file name with spaces replaced by ` (back quote).

To spilt a file the ^Lfilename needs to be inserted at the split point.

As an extension nested packed files are handled by inserting a ? after the ^L for each level of nesting.

By convention for use by make, the packed source file name has the name *directory*_**all.src**, where *directory* is the immediate containing directory name.

Notes on decompiling

Over several years I have refined my approach to reverse engineering old 8085/z80 code and built a number of tools to help me.

Note some of the code in the repository followed older approaches and standards.

In general I follow the steps outlined below as I work through a specific project, however to support these I have a basic naming convention as I start understanding the code.

- procedures and functions are Pascal case. This allows them to be spotted easily when I start translating to high level language as for example in PL/M without an explicit call a simple name or name(expression) might be data reference or a function call, dependent on how it was defined. Using Pascal case makes this obvious
- variable names use camel case. This is a personal preference cf. using $ or _ to separate parts of a name, but also avoids conflict with procedure and functions.
- Unknown procedures are renamed to SubAddress, where Address is the location in memory in hex.
- Unknown data bytes are similarly named bAddress, and words are named wAddress, array variants are named baAddress and waAddress
- Unknown structures are named sAddress and where possible I define a structure type sSize_t to match the structure. It the structure item names are not known I use bOffset or wOffset where b is for byte and w for word, with Offset being the hex offset in the structure.
- Pointer variables either have a p prefix or $p suffix, dependent on whim and/or the detected original source language ($p is typically for PL/M).

1. Disassemble the code. I use IDA Pro which has a number of features I use to make this part reasonably quick. Specifically I have developed a loader for Intel OMF files, created a set of signature files for known PL/M libraries and produced some IDA scripts to handle, common defines, calling conventions, switch tables and other specialist decodes. The ability to rename labels and declare structures is also very useful, the later helps with grouping local data for functions where these aren't stack based.
2. Post an initial disassembly I work through more complex code idioms e.g.
    - Calls with inline data following the call, cleaning up the code accordingly
    - Clean up other invalid code disassembly
    - Special treatment needed for early PL/M compilers
        - Working out true address references as the compilers try to optimise the loading of HL by just loading L and/or incrementing/decrementing H.
        - The calling convention used needs special attention if there are more than two parameters, as the caller writes these to locations defined by the caller, rather than on the stack.
        - Identifying known functions as signature analysis cannot be used, because the code generated can be different depending on where the data areas fall.
    - Indirect calls and jump, often leading to new code areas.
3. For functions I identified by the signatures, or ones that are easy to spot, I update repeating comments to help me understand the calling parameters at other points in the code. I also do this to give meaningful comments for library (instruction emulation) functions.
4. As much as possible I identify the size of data variables, correcting any false word types guessed by the disassembler, The ability to cross reference the code in IDA helps considerably.
5. Once I have a reasonable disassembly and will produce an asm listing which I then edit in vim.
6. I replace library functions with a commented reference, just to make sure I can check the order later
7. I replace known functions with snippets I already have.
8. Dependent on the suspected source language, I run a number of global replace statements over the asm listing. These help convert the asm to closer to the source language, e.g. converting

```
mov a,m
inx h
mov h,m
mov l,a
to
hl=*hl

lxi d,var1
lxi b,var2
call Func
to
Func(.var2, .var1)
```

9. I identify common control structure idioms e.g. PL/M's iterated do loop, and translated these into source code.

10. I then identify potentially new template blocks of code to do global text replacements on. Here vim's regular expressions help, however in more complex cases I will use perl scripts, e.g. to create procedure declarations.
11. Other blocks I hand decompile, for example expressions. I use example code fragments to help me do this.
12. Throughout I look to determine the type of various variables, including arrays and structures, with a view to aid understanding and potential future porting. Some examples
    - Byte variable being used only in a boolean context - define as type bool
    - Address variable used only as an unsigned integer - define as type word
    - Address variable used only as a BASED address - define as pointer (or wpointer if word based address)
    - Address in other contexts - defined as address. Porting will need to figure out how to handle this
    - In older C an int had the same size as a pointer, as an interim declare as intptr_t and use casts. Declare intptr_t as an int for the old compiler.
    - In older C function prototypes were not used, this meant that often different types could be passed without casts. You may need to add casts, which the older compiler should handle. This may also lead to complex union definitions if you intend to port to a modern compiler.
    - Modern C compilers often pass variables in registers, this may cause problems with older code that handled a variable number of arguments. You will need to annotate this as requiring special attention during any port as it will typically need you to use stdarg.h.
13. In general I iterate from steps 6-12 updating data names and structure definitions as I understand more of the code.
14. Once I have what I feel is close to the decompiled source, I identify object file boundaries. In PL/M this the first pass of this is reasonably simple as "DATA" is inline with source and ordinary variables are consecutive. However more work is usually required, potentially moving data declarations to sensible points and for PL/M identifying nested functions. I use the capabilities of my packed source file to mark break points.
15. For PL/M I also create a separated file section for the PEX file to which I add common definitions e.g. TRUE, FALSE, and copy the procedure templates and known global data.
16. I create a default makefile that will at least allow me to attempt to compile all of the source files, without linking.
17. I will then unpack my file and try to compile each created file. Often this will fail and corrective steps are needed
    - syntax errors, these are fixed. Mostly typos or missing) , ;, then or end.
    - Undefined externals, literals / #define, these are added either to the pex file or inline as appropriate
    - Incorrect object boundaries, for example illegal forward references in PL/M.
18. Assuming all of the files can be made to compile, I update the makefile and attempt to link the files. This will often through up additional errors. Most commonly missing public definitions, that cause externals data and procedures to not be found
19. Once the files compile and link, an iterative process begins to fix remaining errors. I do this by building the application and comparing with a reference version. Additionally I use the relst.pl tool to create listing files showing allocated addresses. This allows me to compare with the original code in IDA.

Note I initially focus on major errors due to code differences, starting at the lowest address. Single byte differences due to data being located incorrectly will often change as the code blocks are fixed and I keep these until the end, unless spotted as part of an earlier correction. Typical issues that will need fixing

- Incorrect decompilation. From being plain incorrect , e.g. incorrect end of control structure or incorrect test expression, but also including minor differences like simple swapping of variables, e.g. A < B should be B > A.
- Related to the above, incorrect parameter types or rarely the original source had an error i.e. passed a BYTE vs. ADDRESS. You may need a special external declaration to force this bug into the code.
- Incorrect data declarations. Often this will be incorrect ordering, possibly needing to move declarations around, but occasionally it may because unreferenced variables were declared in the original source, so dummy variables need to be added.
- Related to the above, some more complex issues relate to the fact that the original code may have used a simple array when more modern usage would dictate a structure. If the code generated is different then you will need to work out how to recreate the original.
- Occasionally I have found that inserting dummy labels can force the compliers to generate different code.
- In extreme cases it may not be possible to perfectly match the code, for example the code may have been generated by a vendor's pre-release compiler. Thankfully this has only occurred a few times and often there are work arounds to simulate the special compiler. For PL/M one solution is to compile the code and hand edit the assembler listing to force the match. If needs long variable names use asm80x.pl or if you can globally change the public names to shorter 6 character ones.
- For the Fortran based PL/M compiler the iterative approach can be more problematic as the code generated will vary dependent on where the data is located, so unlike other compilers where a data variable one or two bytes out will throw up a small number of errors, for the older compiler it may cause the code to shift leading to many differences. In this case you need to assess whether the code is incorrect or merely shifted due to data shifts and proceed accordingly. As an example that took me time to identify the problem, is that several applications I hade decompiled, used a buffer at .MEMORY, but one used an explicit buffer. Until I identified this, the data was 128 bytes out and the code was mismatched.

20. For the more complex files, where there overlays, then additional work is required to construct a makefile that will locate the overlays correctly. I usually force the overlays to fixed locations, consistent with the original code, although the original may have calculated these locations. The main downside to my approach is that if may fail if you wish to change features in the original.

Note the above doesn't capture all of the approaches I use, but hopefully it gives you an insight to my approach. I have often contemplated writing a decompiler but so far haven't. I must admit the more I decompile the easier it becomes as I recognise more code blocks.

See Cports.md for more notes on porting PL/M80 to C. Of particular interest is that in porting to C it is often easier to use a debugger to see what a piece of code does and hence enables better understanding of the original code. This in turn supports better function and variable naming and comments.

Note my github site ws21 has also some examples of porting old C code.

```
Updated by Mark Ogden 12-Sep-2020
```