Recreating Basic-E v2.1 & Run v2.3 from published source

When I originally decided to recreate the binary for Basic-E I expected it to be relatively simple, especially as the source has been readily available for many years. However it turned out to be more of a challenge than I expected and the notes below are are record of the issues I encountered and my solutions.

The PL/M compiler

Although the original source compiled with out error using the resident PL/M 80 compiler, looking at the code generated for the individual files it was clear that the compiler used was not one of those I have access to (v3.0, v3.1 and v4.0). Fortunately, with the exceptions noted below it did generate the same code, so I was able to use the v4.0 compiler and handle the exceptions as noted

Un-optimised code

There are several places in the source code where the original compiler generated sub-optimal code c.f. the ones I have.

Un-optimised tests

```
IF NOT expr then            and         DO WHILE NOT expr;

Generate the following code
Original compiler                       Newer compiler
calc expr                               calc expr
cma                                     rar
rar                                     jc ...
jnc ...
```

The net effect is that all subsequent code is a byte out.

To fix this I introduced an additional ENABLE; statement after the IF or WHILE to make sure all later content aligned and do a post build patch to revert back to the original code. I used a literal fixCMA to allow this fix to be optionally applied.

An example

```
IF NOT NUMERIC THEN         is modified to          IF NOT NUMERIC THEN do;
    CALL ERROR('IF');                                   fixCMA;
                                                        CALL ERROR('IF');
                                                    end;
Note the additional do;...end; is only needed for single statement IF
```

Un-optimised loads

```
PTR = expr
expr using variable based on PTR

Generate the following  code
Original Compiler                       Newer Compiler
calc expr                               calc expr
shld ptr                                shld ptr
lhld ptr

The same problem occurs in the case
var1 = expr
var2 = LOW(var1)
```

The net effect is that the subsequent code is three bytes out.

I used one of two solutions to this

- Where possible I introduce a label to force the newer compiler to reload hl. I declared several forceN literals to do this, for the case where there are multiple instances in a single procedure. By defining these literals to a space, the fix can be optionally omitted.
- There was one instance where the above trick didn't work, in this case I inserted 3 ENABLE instructions to force code alignment. Again a defined a literal fixLHLD to allow this to be optionally applied. This option requires a post build patch to replace the 3 EI instructions with the LHLD instruction.

Examples

```
Assuming
DECLARE TEMP1 ADDRESS, ADR1 BASED TEMP1 ADDRESS;

TEMP1 = TEMP1 + 2;              is modified to  TEMP1 = TEMP1 + 2;
                                                force1 /* defines flab1: */
ADR1 = POINT;                                   ADR1 = POINT;


Assuming
DECLARE TEMP3 ADDRESS, .... EXTENT BYTE;

TEMP3 = HIGH(BLOCKSIZE) * BRA(1);   becomes     TEMP3 = HIGH(BLOCKSIZE) *
BRA(1);
EXTENT = SHL(LOW(TEMP3) ,2) +                   fixLHLD;
         SHR((HIGH(TEMP1) + TEMP2),6);          EXTENT = SHL(LOW(TEMP3) ,2) +
                                                         SHR((HIGH(TEMP1) +
TEMP2),6);
```

Optimised code

There was one instance where the original compiler generated better code than the newer compilers and is something I had seen previously. It relates to the iteration variable in a DO loop, as an example

```
DECLARE VAR ADDRESS, I BYTE;
DO I = 1 to 10;
    VAR = expr;
END;

code generated after setting VAR
Original compiler                   Newer compiler
inx h                               lxi h,I
inr m                               inr m
jnz loop                            jnz loop
```

The net effect of this is that the new code is 2 byte longer and hence all subsequent code is two bytes out. Unfortunately the only viable solution with minimal patching is to replace the VAR = expr code with something that is two byte shorter. This keeps the code aligned but requires a longer patch post build.

As an observation, the newer PL/M compilers should have been able to generate the same code as the technique is used elsewhere.

Linking the code

Once I had compiled code, linking the code together was unfortunately not straight forward, even excluding the address fixups noted later.

There were three main issues with the linking, two common to both basic.com and run.com, the other just applicable to run.com.

Bespoke plm80.lib usage

PL/M 80 uses a number of helper functions. These are in plm80.lib, which is structured such that loading one helper function may load a block of related functions as they are stored in a single object module.

In the original compiler, there are instances where the helper functions have not been loaded as group, as such the source of the individual functions used, has to be built separately and explicitly linked rather than using the plm80.lib file.

Library helper functions binding

With the newer compiler any helper functions are shared and even with overlays the helper functions in the root module are shared with the overlays.

In the original compiler, the helper functions appear to be included as part of the PL/M object file and as such there can be duplicate instances of the helper functions in the overall program.

It is simple to synthesise original compiler linking a new PL/M object file with the helper functions, either using plm80.lib or the bespoke functions noted above. What is a problem however is that the link will not link multiple modules together as it complains of duplicate definitions.

As I have seen this issue before, I already had a utility (fixomf.pl) that allows me to remove the helper function public definitions from the synthesised object file, thereby eliminating the error.

Multiple main modules

Run.com is built out of 3 core modules

- Floating point code
- run -- the runtime interpreter
- build -- loads the INT code

Once the INT code is loaded the build module is released for data usage

The way the code was written, both run.plm and build.plm were written as standalone programs, with the jumps to the respective programs being stored in the floating point code that is located at the start of the program.

Whilst I suspect the original build had elements of manually installing the modules into memory, I was keen to automate the process. Unfortunately linking to standalone programs generates an error in the linker as two main programs is not supported.

To get around this issue, I modified my fixomf.pl utility to optionally clear the main module indicator in an object file. This allowed me to link all the files together.

Floating point code precision

When analysing the differences in the generated code vs. the original, I spotted a small number of differences in the floating point code. On deeper inspection, I discovered that the Basic-E version has a small modification in that the displayed precision is one digit less than the floating point code used.

I modified the source code to support an equate BASICE, which if set to 1, will generate the modified precision version.

Address fixups

Basic.com

At the start of basic.com there is a jmp instruction to the main code in the baspar module. The original program notes that this address was patched.

As I wanted to automate the build I would normally aim to use the technique described below for run.com, however in this case this wasn't possible, so I updated my obj2bin utility to write a jmp start instruction at the beginning of the program, using the start information stored in the object file.

The other fixup was around the MON1 & MON2 procedures, which again noted manual fixup. To get around this I replaced the GOTO's in the two procedures with a GOTO CPM and declaring CPM as an external label. In addition I created a simple asm file that defines CPM as a absolute location at 5h. By adding the cpm.obj file to the link, the GOTO CPMs are automatically resolved

Run.com

The fixup information for run.com is at the start of fpinit.src. Specifically

```
    jmp start of build
    jmp start of run
    dw base address of build module
```

In the original source all of these were noted to be patched.

To get these addresses I would normally declare the respective entry points as public labels which would expose the address, but this has the side effect of including an additional lxi sp instruction into the code.

I discovered however that the original binary already had the additional lxi sp instructions, which indicates that this was done in the original build. As a result I modified the source to declare these entry points as public labels, creating the alias MAINLN for MAINLINE as the assembler only accepts up to 6 characters.

With these public labels created filling in the two jumps was simple.

Getting to the base address required me to create a simple asm file bbase.asm, which purely defined a public label in the CSEG. With this I was able to link it just before the main build.obj file, which gave me a simple label to use for the base address of the build module.

The resultant modifications to fpinit.src

```
    extrn mainln
    extrn build
    extrn bbase
    ...
    jmp build-3         ; backs up to the first lxi sp,,,,
    jmp mainln - 3      ;             "
    dw  bbase
```

Patching

As noted above there are a small number of cases where I have to apply patches post build to work around the compiler differences. In addition as the compiled code has uninitialsed data and the com file is padded to a sector boundary. When the Intel omf file is converted the a com file, these additional areas get loaded with what ever data was in the computer memory at the time.

To create the final com files, I apply patch files to the generated omf file that provide the required binary image. There is some annotation in the patch file that describes whether the changes are real patches, random date for uninitialsed areas or random padding to the sector boundary.

Makefile

With the modifications as noted above I now have a source set that compiles to match the original basic 2.1 and run 2.3 com files. The corresponding makefile allows make to automatically build the code and there is the option of building without byte matching by undefining the BYTEMATCH macro definition.

To change the Floating point precision you will need to modify the *.src files to set BASICE to 0.

Note, if you do modify the code and in particular if you move the floating point library, the fpdata.src file declare data using INPAGE, which the rest of the floating point library relies on.

```
Updated by Mark Ogden 16-Oct-2020
```