C ports of Intel ISIS applications

The c-ports directory tree contains my ports of some of the ISIS development chain to C so that they run under modern processors.

Below are details of what is available, followed by some notes on my approach to porting the applications.

Direct ports from PL/M source

These applications closely follow the decompiled PL/M source. For the most part it should be reasonably simple to identify the ports of most of the functions. As I have IDE support for C, I have focused on the C code for asm80 and plm80c, to identify how the applications work. As a result the variable naming and comments are more up to date than the decompiled PL/M source.

Note I have currently not added the auto drive mapping and other features I built into thames.

Environment variables ISIS_F*n* are used to specify the directory associated with drive :F*n*:. :F0: defaults to the current directory.

asm80

The port of asm80 v4.1 and is based on asm80.ov4 which is the macro version of the assembler. I have seen this distributed as asm80 without the other overlays, as the main asm80 just determines whether to run asm80.ov4 (macro support), asm80.ov5 (large memory support no macros) or to use overlays asm80.ov1, asm80.ov2 and asm80.ov3 for small memory support.

As noted the comments and variable naming are more up to date than the PL/M version.

lib, link & locate

The ports of lib v2.1, link v3.0 and locate v3.0

plm80c

This is the C port of PL/M 80 v4.0 and is the preferred port, rather than the much older C++ port plm80 noted below. This is the one I am adding commentary to as I understand the compiler internals.

When compiled it generates an executable plm80.exe

Note I kept the old directory naming to avoid messing up the git history.

Other ports

plm80

This was a very early translation from the plm v4.0 binaries to C++. It was done before I decompiled the source to PL/M. It is written in an old version of C++ and is very clunky. I have left it for historical reasons but no consider it obsolete. If you compile the source it will now generate an executable oldplm80.exe

This old version uses a different drive to directory mapping approach. :F1: maps to the current directory, other :Fn: map to subdirectories named fn.

plm81.exe, plm82.exe

These two programs are my port of the Fortran based PL/M compiler to C. Although based on the V2 compiler, some of the patches from V4 have been incorporated to fix invalid code generation.

In addition both compilers now accept the name.plm as a file to compile, the old way of using fort.n is also still supported. This change allows for parallel compilation. If plm82 requires $ config information this will be read from name.cfg is present or fort.1. It is no longer an error for neither file to exist. Note name is not restricted to the ISIS 6 character alphanumeric limit.

When name.plm is specified, the intermediate files from plm81 are name.pol and name.sym, the file name.lst also contains the concatenation of the two listing files from plm81 and plm82, hence both should use the same name parameter or both omit it. Hex output is stored in name.hex.

The other change is that the message previously written to fort.1 by plm82 is now written to the console. This is message re-errors.

```
Usage:   plm81 file.plm                or  [mv | move] file.plm fort.2
                                           plm81
                                           [mv | move] fort.16 fort.4
                                           [mv | move] fort.17 fort.7
                                           [mv | move] fort.12 fort.12a
        plm82 file.plm                     plm82
                                           [mv | move] fort.17 file.hex
                                           [cat fort.12a fort.12 >file.lst |
                                            copy /Y fort.12a+fort.12 file.lst]
cleanup: [del|rm] file.pol file.sum        [rm | del] fort.*

Note plm81 and plm82 also support -v & -V which show version information and
provide simple help
```

binobj, hexobj & objhex

1. These are functionally equivalent ports of the corresponding ISIS tools. They support both the Intel invocation syntax and a more modern less verbose version.

Unlike the other tools these do not support ISIS drives, but do support unix/windows file path names.

Note due to memory constraints the ISIS binobj and hexobj would split very long content records. The ports do not need to do this.

Notes on porting

Prior to porting it is helpful to make sure that the source code follows a number of conventions as this makes it easier to spot variables vs. functions and helps with semi automation of the conversion. In particular

- All procedures follow the Pascal case naming convention. This makes functions stand out from arrays and simple variables.
- All variables follow camel case naming convention. Replacing $letter with Uppercase letter.
- As many basic data types mapped to one the following

```
byte, word, dword, boolean, integer, pointer and address
Here address is limited to when the variable holds both a word or pointer at
different times
I also use wpointer as a pointer to a word variable; common for system
calls.
```

- If ngenpex is used, the pex file is helpful as this contains information that can be use to generate function prototypes, variable types, macros and typedefs

From this base point I use a mixture of bespoke perl scripts and global edits using vim and visual studio to create a base point for further analysis. This includes using support files e.g. command line parsing, I/O and macros from previous ports.

The process is then iterative to get a version that compiles, followed by debugging to get a working version. This can include looking at the code generated from the original PL/M source to check the precise interpretation of complex conditionals.

Below are some of the typical issues that need to be addressed during the porting, which may be of interest to anyone contemplating their own ports.

Variables and memory layout

Other than for trivial programs, it is unlikely that a single strategy can be used for variables and memory layout.

Simple variables

- byte, word, dword, boolean, integer can be mapped to uint8_t, uint16_t, uin32_t, bool, int16_t respectively. If byte level access is needed to word, dword or integer values through based variables, the use of AT statements (effectively unions) or the writing of memory to disk, then it may be necessary to create accessor functions to handle the byte order mapping.
- pointer & wpointer types are more problematical as their size is dependent on the target processor. I handle this in two ways.
    1. Where possible I map to a standard C pointer, the type determined by the base variable usage of the pointer. Where the are different usage scenarios this may need to be a union or a void * pointer. Note wpointer maps to uint16_t *.
    2. Where the the pointer needs to be stored in a word size variable, a scenario typical in accessing heap type memory, I use accessor functions that map the pointer to / from an offset.
- Of the base types, address is the most complex and can be typically handled in two ways
    1. As a union of the numeric and pointer types. Where pointer can be implemented in either of the ways noted above. The challenge is determining the correct type to extract at each point in the program. Until more is known about the specific pointers I typically define a typedef for address as follows

```
typedef union {
    uint8_t b[2];   /* allow for byte pair */
    uint16_t w;     /* word value */
    uint16_t off;   /* a 16bit offset into a memory space */
    uint8_t *p;     /* optional if direct C pointer can be used */
                    /* note not void * to allow for p++ */
} address;
```

2. As an intptr_t, assuming that there are no specific space / alignment requirements. Here specific casts will be needed for each pointer use.

Structures & Arrays

The main consideration in using structures and arrays is in constraints the C compiler places on data alignment and potentially on the data byte ordering for the scenarios outlined above.

If the size of a structure or array needs to match the PL/M size e.g. because it is written to disk or other code depends on the offsets of elements, then if the compiler has a suitable pragma to force byte alignment, this can be used, otherwise a byte array will need to be defined and suitable accessor functions created.

A variant of the above problem is that many PL/M programs assume variables are allocated in consecutive memory. Unfortunately this isn't guaranteed in C, so often the code / data will need to be reworked. For example the following take from isist0.plm v2.2

```
DECLARE MSG1(8) BYTE INITIAL(CR, LF, 'ERROR '),
        MSG2(3) BYTE, /* ERROR NUMBER GOES HERE */
        MSG3(9) BYTE INITIAL(' USER PC '),
        MSG4(4) BYTE, /* USER PC IN HEX GOES HERE */
        MSG5(2) BYTE INITIAL(CR, LF),
        MSG6(5) BYTE INITIAL('FDCC='),
        MSG7(4) BYTE, /* FDCC ERROR DATA GOES HERE */
        MSG8(2) BYTE INITIAL(CR, LF);
/*
The code fills in the strings at MSG2, MSG4 and MSG7 then writes out the whole
string
*/
```

A couple of example solutions to this would be

1. Create a single MSG1 string and define the other MSGn variables as #define statements e.g.

   ```
   uint8_t MSG1[37];
   #define MSG2    (MSG1 + 8)
   ...
   ```

2. Use sprintf or similar to write the whole formatted string to MSG1 or to directly emit the string.

Note for static string variables, unless there are overriding requirements, I typically use a standard quoted string with it's extra trailing '\0' byte. Modern C's convention of concatenating adjacent strings is useful to create some of the longer tables.

At variables

AT is used in two ways in PL/M and typically needs different handling in C.

1. AT(absolute address) - this is used to make to a OS location. Although this could be handled in C by setting a pointer to the absolute address and dereferencing it, in reality it is more likely that any code that uses it will need to be re-written. As an example CP/M iobyte is often accessed this way but has no corresponding meaning in a C port.
2. AT(.var) - this is the way PL/M effectively handles unions. I use one of three approaches to handline this

1. Create a union and either change the variable names or use #define to map them to a separate union variable.
2. Use casts. This may be needed if the AT(.loc) is not to a simple variable
3. Use accessor macros / functions. If the AT function is used to access high / low bytes then this may be necessary to handle byte ordering.

Based variables

Based variables are how PL/M handles pointer dereferencing. However C pointer arithmetic isn't supported. I use four basic approaches to handling Based variables

1. Where a based variable is used in a simple way and the underlying pointer does is only dereferenced in one way, it is often easier to convert the pointer into a C pointer to the base type e.g.

```
declare ptr pointer;
declare wrd based ptr word;
while (wrd <> 0)
    ptr = ptr + 2;

can be converted to

word *ptr;
while (*ptr)
    ptr++;

Note care needs to be taken with the increment of ptr to reflect C's pointer
arithmetic
```

2. If a pointer has multiple ways of being dereferenced, then it can be cast into the appropriate type. Care will still be needed with any arithmetic with the underlying pointer, especially if the data alignment does not match PL/M. For this approach pointer can be declared as uint8_t * as this allows basic pointer arithmetic consistent with PL/M.
3. A variant of the above is to create a union of the types and use C -> to select the appropriate type. In this case the pointer would be declared as a pointer to the union. Note the pointer may need to be cast to uint8_t * to support pointer arithmetic consistent with PL/M.
4. Use accessor macros/functions. This may be needed if byte ordering or special alignment is needed

Heap space

Many non trivial programs use the free space between MEMORY and the top of memory as heap space and have procedures that manage this space. The normal C equivalent would be malloc, however this will have native address pointers which may not be viable without major changes to the application code. I tend to follow the following approaches

- If the memory is a simple buffer or structure from MEMORY upwards. I pre-allocate the buffer either statically or via malloc.
- For true heap management I pre-allocate a memory image either statically or via malloc, which matches the heap size available to the program in ISIS. This is then accessed using accessor function that use offsets to read / write the data on the heap. This requires finding all instances of access to the heap and using the appropriate accessor function to either get the data or return a pointer to the data. An additional function is provided to convert a pointer into an offset. The advantage of this approach is that the data uses is a 16 bit offset

value, i.e. the same size as the original used. Additionally if the base offset is correct, any offsets written to intermediate files will match the ones the native tools generate.

Address of parameters

For PL/M 80, unless a procedure is marked as re-entrant, the passed in parameters are copied into locally reserved consecutive memory locations. This allows the address to be taken of a passed in parameter. In C this may fail as the compiler may pass parameters in registers, or the data will be padded to stack variable boundaries. As an example from the PL/M 80 compiler, *note the data types follow an earlier convention*

```
WrTx2Item2Arg: PROCEDURE(arg1b, arg2w, arg3w) ADDRESS public;
    DECLARE arg1b BYTE, (arg2w, arg3w) ADDRESS;
    call Sub$4251(.arg1b, 5);
    return T2CntForStmt;
end;

Here the three variables are passed to Sub$4251 though the use of .arg1b.
The C code I used is

word WrTx2Item2Arg(byte arg1b, word arg2w, word arg3w)
{
#pragma pack(push, 1)
    struct { byte arg1; word arg2, arg3; } tmp = { arg1b, arg2w, arg3w };
#pragma pack(pop)
    Sub_4251((pointer)&tmp, 5);
    return t2CntForStmt;
}
```

Conditional expressions

For conditional expressions PL/M uses the least significant bit of a value with 1 representing true and 0 false. Additionally PL/M does not have the short circuit evaluation options available in C i.e. && and ||.

Fortunately in many cases it is possible to translate PL/M expressions to use the C short circuit versions however each needs to be checked. Some key points to note are:

- Simple variables in conditionals e.g.

  ```
  if var then
  ```

  If these variables are used as true/false in the application then no change is required, otherwise use

  ```
  if (var & 1) then
  ```

- Function invocation in conditionals e.g.

  ```
  if isalpha(c) or isnumeric(c) then
  ```

If any of the functions have side effects then you will need to replace and / or with the non short circuit variants. Occasionally this can be avoided by re-arranging the order of the tests by putting non side effect functions at the end, allowing them to use the short circuit evaluation.

- Setting a value to the result of a conditional expression e.g.

  ```
  test = a > b;
  ```

  All uses of test need to be reviewed. If test is only ever used as a boolean value, then the expression should be ok, if not use

  ```
  test = a > b ? 0xff : 0;
  ```

  The default C would set true to 1 rather than 0xff.

  When setting a value additional care is needed to check whether the target is actually boolean. For example the following relies on how PL/M handles conditional and could be used to set the debugFlags to verboseFlags if debugLevel > 2, else clear debugFlags

  ```
  debugFlags = verboseFlags and debugLevel > 2;
  ```

  In C this can be represented by

  ```
  debugFlags = verboseFlags & (debugLevel > 2 ? 0xff : 0);
  or more natrually as
  debugFlags = debugLevel > 2 ? verboseFlags : 0;
  ```

Note it may be possible to replace some nested PL/M if statements by more idiomatic C short circuit code for example.

```
declare val based ptr byte;
if ptr <> 0 then
    if val <> 0 then
        ...

can be replaced by
if (ptr && *(byte *)ptr)
    ...
```

8 bit arithmetic

Unlike C, In PL/M 8 bit values are processed using 8 bit arithmetic and only promoted to 16 bit when involved with 16 bit numbers. Care needs to be taken to identify these cases and force the 8 bit calculation in C, for example by using a (uint8_t) cast.

Some examples, note the different handling for arrays when the compiler detects a index + fixed offset.

```
declare a address, b byte, c(257) byte, d byte;
a = -1;             /* sets a = 0xff */
a = -double(1);     /* sets a = 0xffff */
a = double(-1);     /* sets a = 0xff */
a = 255 + 1;        /* sets a = 0 */
b = 255;
d = 1;
a = b + 1;          /* sets a = 0 */
a = b + d;          /* sets a = 0 */
c(b + d) = 0;       /* sets c(0) = 0 */
                    /* but !!! */
c(b + 1) = 0        /* sets c(256) = 0 - because the compiler adds 255 to .c(1)
*/
c(b := b + 1) = 0;  /* sets c(0) = 0 */
```

Expression order

In C the order of evaluation of expressions is not guaranteed in all cases so simple PL/M may need to be split. Probably the most common example I have seen is to get a two byte word value from a data stream. In PL/M the following works

```
return getc() + getc() * 256;
```

In C the order of calling of the two getc() is not specified, so this has to be translated to

```
{
int c = getc();
return c + getc() * 256;
}
```

The same type of modification would be needed if the side effects of various calls needed to be done in order.

Nested procedures

As C does not support nested procedures, these need to be extracted and raised to file scope. For simple procedures it may be possible to define a macro to replace the code, undefining the macro after it is used. The preferred method however is to create a file level static function, possibly renamed to indicate that it was a nested procedure.

As nested procedures have access to their parent variables, these will need to be passed in either as value parameters if not modified or the addresses of the parameters if they modify their parent's data. The code will need to be changed to accommodate this change.

Potentially a single parameter change could be done via return value unless the nested procedure already returned a value.

Public labels

PL/M allows code to jump to known global locations and in doing so resetting the stack pointer. The C equivalent of this is longjmp. In general the modifications to use this are reasonably straight forward, however some careful analysis is required to make sure that longjmp is replicating the functionality of the PL/M label. Some minor code modifications may be needed to make this work correctly.

Chaining

As memory is no longer a major issue, I usually merge all overlays into a single application, with effectively a control switch that transfers control between the various pseudo overlays.

In addition to avoiding overlays this also allows for code sharing to be done, however it does involve extra work to make sure that functions and variable names are not in conflict and make sure that variables are initialised appropriately at the start of each pseudo overlay.

```
Updated by Mark Ogden 26-Sep-2020
```