

PowerPC PEF

Program file format

Thu, Nov 18, 1999

This note results from studying an example of the PEF program file format used for PowerPC software development. The test example was a dummy program that had no outside references, and it was compiled into position-independent code. A document that describes this file format in detail was used to correlate with the example file.

The file header includes some keys that may be checked to insure that the contents of the file are being interpreted properly. For a local application, which is the reason of this study, the first instruction will likely be a "mflr r0" instruction, which seems to be used in a routine that calls another, since in this case the link register must be saved and restored. Every local application will include a main program that invokes DoMain, so that the main code occurs first in the program instruction part of the file.

In the example studied, there is a section 0 that contains code, a section 1 that contains compressed data, called Pidata and a section 2 that contains relocation instructions. (The latter was not used for the expected purpose of resolving references to external routines, since there were none. It seemed to be used for creating a kind of transition vector that could be used by a program to invoke the main program.)

In order to ultimately call a PowerPC function that is a local application (or page application) taking two arguments, the argument registers r3 and r4 must be loaded properly, and the RTOC register (r2) set to point to the data area used by the function that is called. The entry point for the function will be at the start of the code, but we must find the data area somehow.

In the PEF, we use fields in the section headers to find the appropriate stuff required. We must expand the compressed data section and concatenate it with the code section to produce the file that must be downloaded via TFTP. In the course of doing all this, we will know where the data section is that we appended to the code section. But we must retain this information somehow in the downloaded file that is placed into nonvolatile memory, so that when it is time to invoke the program, we can find the location of the data area to pass via the RTOC register.

One approach might be to precede the downloaded file with 16 bytes that consist of a key and an offset to the data area from the start of the program. This will provide the information that we need to invoke the program function later. Or, the first two long words of this header can hold an offset to the code and an offset to the data area.

It appears that in order to achieve having no external references in our local/page applications, we will have to implement some sort of TRAP mechanism to reach system routines. Without doing this, we shall have to deal with relocation in the programs as they are invoked, which seems too complicated. The reason for doing this has to do with the desire to retain as much as possible the functionality we now have in working with the nonvolatile memory-based file system. We have support for version dates by using TFTP to download the first copy, which causes an automatic time stamp of a version date, then copying that copy to any other nodes while retaining the version date.

We need a utility that either provides the equivalent functionality as the MPW-based TFTPtool, or we need a utility that first produces a CODE (or other) resource that looks like the one that TFTPtool can handle. It may in any case be helpful to have a utility that reads a PEF file and writes a file that contains exactly what must be TFTP'd to a target front end node. This utility

would have to perform the expansion of the compressed data section, append it to the code section, and precede all this with a 16-byte header. (The PowerPC seems to prefer 16-byte boundaries.)

The VxWorks approach would use downloaded programs that are loaded via the `ld` command at the telnet prompt. Note that the above scheme does not preclude the VxWorks approach. It is still available to be used in cases where it makes sense. There may be some diagnostic reporting function that might be useful, for example.

Details:

The `dumppef` output defines the fields that need to be processed to select the code to be used as a file to download. After the standard container 40-byte header, one would expect to find two sections defined, at least. Section 0 comprises the position-independent code, and section 1 holds the data used by the code in a compressed format.

The container header fields include keys that can be checked and the number of sections defined in the following section headers. The number of loadable sections is probably 2. Maybe the memory address field should be expected to be zero for position-independent code.

In the section 0 header, the `execSize`, `initSize`, and `rawSize` fields should be equal. The `regionKind` should be `0x00` to indicate a code section. The alignment likely will be 4, indicating 16-byte alignment is needed, although this is not important for building the downloadable file. The `containerOffset` field will specify where the code begins, as an offset from the start of the container header.

In the section 1 header, the `execSize` will indicate the resulting size after expansion. The `rawSize` indicates the size of the compressed data of this section. The `regionKind` field should be `0x02`, which implies the compressed format, sometimes referred to as "pidata," for pattern initialized data. The alignment may again indicate 4, meaning 16-byte alignment. Since the code section is 16-byte aligned, it may be necessary to use filler bytes between the end of the code section and the start of the data section. Again, the `containerOffset` field in the section 1 header indicates where the pidata begins, in terms of an offset from the start of the container header.

In order to expand the pidata section into a raw data block that is to be appended to the end of the code section, a series of pattern initialization opcodes must be interpreted. There are several of these specified in the PEF Structure chapter. In the simple example used as a test for this analysis, only two types of opcodes were used. The general format of an opcode is a byte of opcode followed by optional arguments that depend upon the opcode. The actual opcode value is in the most significant 3 bits of the byte, and the lower 5 bits is a count field. The optional arguments follow, in which each subsequent byte can hold 7 bits, with the other (sign) bit carrying the continuation flag. The last byte of such a continuation sequence has the sign bit clear. This implies that 5 such bytes would be needed to declare a 32-bit argument, but smaller arguments, but the more-likely smaller argument values would be less. In the case that the count is more than 31, so that it cannot be expressed in 5 bits, then the 5-bit count field is set to 0, and the first argument holds the actual count.

Looking at the example that was analyzed, we see an opcode byte of `0x28`, which specifies a 1 opcode value (blockCopy, or BLK) and a count of 8 bytes. This case requires no arguments, so

the following 8 bytes are to be copied into the output byte stream. (Think of interpreting these pattern-initialization instructions to produce an output stream of bytes that will form the raw uncompressed data.) The next instruction is 0x08, which specifies a 0 opcode value (Zero, or ZERO) with a count of 8 bytes. Again, this instruction has no argument, so the next 8 bytes to be produced are zeros. The next instruction indicates 0x22, so that the following 2 bytes of data are copied into the output stream. The last instruction is 0x0E, meaning that 14 more zero bytes end the production of raw uncompressed data. (Three more instructions, each of which specifying at least one argument, are defined in the PEF Structure chapter.) The end of the series of instructions is detected from the rawSize field in the pidata section header.

The raw data is appended to the code of section 0, and the resultant concatenated structure constitutes the program file. It may be useful to first build a temporary file that holds the concatenated result, then initiate downloading from this file using the TFTP protocol, which specifies that each downloaded datagram holds exactly 512 bytes—except the last one, which may hold 0 bytes, if required. A receiving node, upon detection of a short datagram, knows that it is the last one of the file to be transferred.