The PRC Format

as guessed/compiled from various sources by
Theodore Ts'o
Last updated February 15, 2000

Recent Changes to This Document

NEW Thanks to Ian Goldberg, the format of the RLE compression in the Data resource is now documented.

Introduction

Up until now, there really hasn't been a public documentation of the PRC format. This was was frustrating for me since I wanted to write a BFD backend for PRC executables. I tried looking at the sources for some of the alternative software Pilot development programs, such as Pila and prc-tools. Unfortunately, these programs would often treat header fields as "magic", and often different programs would do completely different things with the same fields.

This document is my attempt to rectify this situation. It is the product of both research into existing implementations, as well as experimentation to clarify some minor points of how the Pilot tools work. Some of my sources include:

- The Pila compiler for Unix
- The obj-res and emit-prc programs from the prc-tools package
- Inside Macintosh

Any mistakes in this document should be considered my responsibility, however, and not the responsibility of these sources. If you find any mistakes, or have anything to contribute, please contact me via email at tytso@mit.edu.

High-level format of a PRC file.

An application for the pilot is simply a Pilot resource database with a number of mandatory resources (CODE 0, CODE 1, DATA 0, PREF 0, etc.) The PRC file, then, is simply the flat file representation of a Pilot resource database. When the PRC file is loaded into the Pilot, it is converted into a resource database using the PalmOS routine dmCreateDatabaseFromImage().

The PRC format consists of the following major pieces:

1. *PRC Header*
2. *PRC Resource Headers*
3. *PRC Resources*

PRC header

The PRC Header is located at the very beginning of the file, and contains the following information:

offset	name	type	size	notes
0x00	name	char		

32	[1]			
0x20	flags	int		
2	[2]			
0x22	version	int		
2	[3]			
0x24	create_time	pilot_time_t	4	[4]
0x28	mod_time	pilot_time_t	4	[4]
0x2C	backup_time	pilot_time_t	4	[4]
0x30	mod_num	int	4	[5]
0x34	app_info	int	4	[5]
0x38	sort_info	int	4	[5]
0x3C	type	int		
4	[6]			
0x40	id	int		
4	[7]			
0x44	unique_id_seed	int	4	[5]
0x48	next_record_list	int	4	[5]
0x4C	num_records	int	2	[8]

[1] The name field is zero terminated and is usually zero padded. The pila assembler sneaks 'Pila' into the last 4 bytes of this field

[2] The 'flags' field is 0x01 for PRC executables. The 0x40 bit is set if the executable is considered non-beamble. (Note that this means it's probably fairly easy to make a non-beamble application to be beamable...)

[3] The 'version' field is 0x01 for PRC executables

[4] Pilot time is defined to be the number of seconds since January 1, 1904 (i.e, Macintosh time).

[5] This field must be zero for PRC executables

[6] The 'type' field must be 'appl' for PRC executables

[7] The 'id' field is a four character "creator code", ala the Macintosh

[8] The 'num_records' field contains the number of resources in the PRC file.

PRC Resource Headers

The Resource headers follow immediately after the PRC Header field. The num_records field in the PRC Header indicates the number of resources contained in the PRC file, and there is a 10 byte resource header for each resource.

name	type	size	notes
name	char	4	Name of the resource
id	int	2	ID number of the resource

offset	int	4	Pointer to the resource data

PRC Resources

The actual data for the resources follow after the resource headers. The resource data records are stored in order as they appeared in the resource headers. (Since the resource header does not have a size field, the size is determined by examining the where the offset pointer for the next resource.)

The Mysterious Code #0 Resource

The contents of this resource have been (up until now) somewhat mysterious, with different packages --- Metroworks, Pila, and the obj-res program from the prc-tools package --- generating in some cases very different values.

Pila creates an 8 byte resource, with the first four bytes described as the initialized data size and the next four bytes described as the unitialized data size. Pila stores the size of the data segment in the first field, and the second field is always filled with zeros.

The obj-res program from the prc-tools package does something quite different. It creates a 24 byte resource, which is filled in as follows:

offset	value
0	0x00000028
4	[bss+data segments rounded up to 4 bytes]
8	0x00000008
12	0x00000020
16	0x0000
18	0x3F3C
20	0x0001
22	0xA9F0

The obj-res program treated most of the fields in the 24 byte resource as magic values, apparently obtained from looking at the contents of the code 0 resource from PRC files generated by the Metroworks compiler, since applications generated by the Metrowerks compiler have similar code 0 resources.

I believe that the code 0 resource is identical to that which is used by the 68k Macintosh. This explains why pila can use such a different code 0 resource, and yet still produce working appications. To explain this, though, we need to take a detour and look at the Macintosh memory management model.

The Code 0 Resource on the Macintosh

On the Macintosh, when an application is lauched, MacOS allocates allocates a block of memory known as an *application partition*, which has the following format:

```
                   +--------------------+
                  /|                    |
                 / |  Jump table        |
                |  +--------------------+
A5 world ---> |  |  Appl. Params      |
                |  +-------------------+  <--- A5 pointer
                 \ |                   |
                  \|  Appl. Globals    |
                   +-------------------+  <--- Stack base
                   |                   |
                   |     Stack         |
                   |      vvv          |
                   +-------------------+  <--- Stack pointer
                          ....
                   +-------------------+
                   |                   |
                   |      ^^^          |
                   |      Heap         |
                   +-------------------+
```

The *A5 world* is very important to a Macintosh application. It roughly corresponds to the data and bss segments of a Unix executable, but it serves a few additional functions related to the application's memory management.

In general, 68k Macintosh executables don't seem to bother with relocations; instead, the code segment of an application uses only position-idependent code, and it references its jump table, application parameters, and application global variables as fixed offsets (positive and negative) from the A5 register. The A5 register is always pointing at a fixed location inside the application's A5 world.

The code 0 resource is generated by the linker (for example, the Metrowerks linker) and contains the necessary information so that MacOS can setup an application's A5 world. It has the following structure:

offset	size	description
0	4	size above A5 (jump table+parameters)
4	4	size of application globals
8	4	size of jump table
12	4	A5 offset of jump table
16	8	Jump table entry #0
24	8	Jump table entry #1
...		
16+8n	8	Jump table entry #n

The jump table is used to transfer control between different code segments, which may not yet be loaded into the system. Jump table entry #0 points to the start address of the application. Jump table entries have two forms, depending on whether the destination code segment to which the entry points is loaded or unloaded. Initially, all code segments are unloaded, and so all jump table entries have the following form:

offset	size	description
0	2	Offset of this routine from the beginning of the segment
2	2	m68k push instruction
4	2	segment number (arg for push instruction)
6	2	_LoadSeg trap

When an application transfers control through a jump table to an unloaded segment, it casues a call to the _LoadSeg trap, which loads the segment and then modifies all of the jump table entries for the application that point to the now-loaded segment with the following jump table entry:

offset	size	description
0	2	Offset of this routine from the beginning of the segment
2	6	m68k long jump instruction to the routine in another segment

Hence the calling macintosh application can always transfer to another code segment by jumping to offset 2 for a particular jump table entry. This scheme has the effect of a "poor man's virtual memory", since it allowed code segments to be demand-loaded as necessary, without requiring an MMU (which early Macinoshes didn't have!).

What Does All This Have to do with the Pilot?

An examination of Pilot applications which were created using the Metrowerks compilers makes it clear that the code 0 resource of the Pilot follows the format of the code 0 resource of a 68k Macintosh. In particular, the format of the jump table entry is unmistakable. Consider the code 0 resource for the PalmPilot's Expense application:

offset	size	value	description
0	4	0x00000030	size above A5 (jump table+parameters)
4	4	0x00000060	size of application globals
8	4	0x00000008	size of jump table
12	4	0x00000020	A5 offset of jump table
16	2	0x0000	Jump table entry --- offset
18	2	0x3f3c	Jump table entry --- push instruction
20	2	0x0001	Jump table entry --- segment
22	2	0xA9F0	Jump table entry --- SegLoad trap

All of the fields from this pilot application match up correctly with a 68k macintosh code 0 resource. The size of the jump table is correct (8 bytes), as is the start address of the application (code segment 1, offset 0) in the first (and only) jump table entry. Hex 0x3f3c is a push instruction which places segment 1 on the stack.

Now that we have confirmed this hypothesis, what does this have to tell us about the A5 world of a Pilot application? First of all, like the Macintosh memory model, the application globals are located **below** the A5 register. Hence, accessing application globals requires making negative offsets to the A5 register. The expense application reserves 48 bytes of space above the A5 register for the jump table and "application parameters". What gets stored in the application parameters space? More on that a little later.

However, apparently not all of the code 0 resource is used by the Pilot, at least not in PalmOS 1.0 or 2.0. For example, the Pila assembler only creates a code 0 resource which is 8 bytes long, and PalmOS 1.0 and 2.0 don't seem to mind that the rest of the code 0 resource isn't present.

I also tried selectively corrupting the jump table of an application generated by the Metrowerks compiler, and this did not affect the behavior of the application. Hence, it appears that the Pilot does not use the jump table to determine the application start address.

A much more interesting way of confirming our observations thus far is to consider the Pila's alternative memory model. In the code 0 resource generated by the Pila, the "Application Global" size is 0, and the size above A5 is set to the size of the Pila program's data segment. In other words, Pila programs have their data segment **above** A5, instead of below it.

Does the fact that the data segment for Pila-compiled programs is located where the "jump table" and "application parameter" section cause any problems? Yes, although Pila has a workaround that apparently works for PalmOS 1.0 and 2.0. Currently, the PalmOS loader stores a pointer to the application's SysAppInfo at the beginning of the applications parameter section --- that is, at the four bytes starting at the A5 register. Some of the PalmOS ROM routines depend on this pointer being present. To avoid overwriting it, Pila's startup routine reserves four bytes of space at the beginning of the segment, and when Pila constructs the compressed Data segment, it is set up to start decompressing starting at an offset four bytes beyond the A5 register.

One useful data point which we can infer from Pila's non-standard memory module is that only the first four bytes of memory above the A5 register currently appear to be in use. Otherwise, Pila compiled programs would likely cause some kind of crash or Pilot malfunction. Apparently the rest of the 32 bytes reserved by the Metrowerks compiler for "Application Parameters" is reserved for future expansion, but is not being used now. In addition, since some Pila-compiled programs have data segments greater than 32 bytes, this also confirms our theory that the jump table is also currently not being used.

This raises a cautionary note that while Pila-compiled programs work now, they may fail in the future if later versions of PalmOS use additional memory above the A5 register beyond the first four bytes. The PalmPilot Developer Technical Brief explicitly warns that "If your application was not developed with the Metrowerks CodeWarrior for Pilot, it may run into problems." Developers would do well to heed this warning, especially in the case of Pila where it is using a radically different memory model where the data segment of the assembly language program is overloading memory space reserved for application preferences and for the jump table.

The Code #1 resource

This resource contains the actual code for the application. For some reason, PRC executables generated by the Metrowerks compiler have the a four-byte word 0x00000001 (ori.b #1, %d0) at the beginning of the code resource. The obj-res program duplicates this behaviour, although Pila does not, and it doesn't seem to make a difference.

It is not clear whether the four byte word is meant to a flag or bitfield, or whether it is some other kind of signal. When PalmOS starts executing the application, it obviously starts at beginning of code segment #1. To test to see if the initial four byte word was intended to be interpreted as a instruction, I tried replacing it with a rts instruction. This test made it clear that the ori.b #1, %d0 instruction is actually executed. However, this instruction doesn't appear to do anything useful. It merely sets the low bit in data register 0; however, data register 0 is never used until it is later re-initialized.

OPEN QUESTION: Why does the Metrowerks plance an initial 4 byte prefix (0x00000001, or ori.b #1, %d0) in the CODE 1 segment, and why does it matter?

The Data Resource

The data resource is perhaps the most mysterious resource, because it is neither documented by the USR-provided Pilot Tutorial and Cookbook books, nor in Inside Macintosh, since the Data resource is unique to the Pilot. (MPW uses a similar, although different, mechanism which is used to initialize global variables, involving the use of the A5init segment.) Most of the information in this section has been taken from comments in the Pila assembler. Apparently Darrin Massena, the author of Pila, had some contacts inside the Pilot development group which gave him some of the necessary technical information.

The major purpose of the data resource is to initialize global variables. The data resource can also contain relocation tables to handle arrays containing pointers to static data (for either constant data stored in the code 1 segment, or writeable data which is stored in the data resource). The high-level format of the data resource is:

offset	size	description
0	4	offset of CODE 1 xrefs (4+n+m)
4	n	compressed global initializers
4+n	m	compressed DATA 0 xrefs
4+n+m	p	compressed CODE 1 xrefs

The compressed global initializer section

The compressed global initializer section contains the following substructure repeated three times:

- A5 offset (4 bytes), which specifies the destination where the uncompressed data should be written.
- Compressed stream, in blocks
- ASCII 0 (separator).

This flexible structure allows up to three contiguous regions of the application's A5 world be initialized with compressed data. If the flexibility is not needed, one or more of these initializer substructures may be replaced by 5 nulls (4 nulls for the A5 offset and one ASCII null to indicate the lack of compressed blocks).

The Pilot uses an enhanced RLE scheme for its compressed stream. The compressed stream contains a series of RLE blocks, followed by a zero byte to terminate the compressed stream. Courtesy of Ian Goldberg, the following RLE blocks are recognized by the Palm Pilot Pro:

byte stream	description
(0x80 + n) b_0 ... b_n	n+1 bytes of literal data (n <= 127)
(0x40 + n)	n+1 repetitions of 0x00 (n <= 63)
(0x20 + n) b	n+2 repetitions of b (n <= 31)
(0x10 + n)	n+1 repetitions of 0xFF (n <= 15)
0x01 b_0 b_1	0x00 0x00 0x00 0x00 0xFF 0xFF b_0 b_1
0x02 b_0 b_1 b_2	0x00 0x00 0x00 0x00 0xFF b_0 b_1 b_2
0x03 b_0 b_1 b_2	0xA9 0xF0 0x00 0x00 b_0 b_1 0x00 b_2
0x04 b_0 b_1 b_2 b_3	0xA9 0xF0 0x00 b_0 b_1 b_2 0x00 b_3
0x00	end compressed data

OPEN QUESTION: Are all of these compression blocks supported on the old PalmOS 1.0 machines? I am particularly paranoid about the RLE blocks beginning with 0x01 -- 0x04. Also, why did PalmOS define special cases for 0xA9 0xF0?

Code 1 XREF and Data 0 XREF sections

Unfortunately, the format of the XREF sections is totally unknown. The Pila and obj-res programs currently emit 6 longwords of zeros.

The obj-res program supports its relocation of initialized data by manually including a relocation table which get processed by a custom startup routine. It would be cleaner to allow the PalmOS loader to do this work for the application automatically. (Although relying on this would probably limit that application to PalmOS 2.0 devices.)

OPEN QUESTION: What is the format of xrefs?

This page has been accessed times.

Back to Ted's Pilot Page.

Theodore Ts'o