A FORTH-Based Object File Format and Relocating Loader used to Bootstrap Portable Standard Lisp

Harold Carr and Robert R. Kessler

Utah Portable Artificial Intelligence Support Systems Project
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract: For a quarter of a century much Artificial Intelligence research has been accomplished using Lisp as the implementation language. Portable Standard Lisp (PSL) was created to support research activities on a wide variety of processors and operating systems. To aid in porting PSL, we developed an ASCII object file format whose relocation directives are essentially FORTH language statements. A Portable Linker is then used to directly link compiled Lisp code normally dependent on the Lisp runtime system. Finally, since the object file contains FORTH statements to handle final relocation of segments, we wrote a relocating loader in FORTH to load and execute these "exported" Lisp programs. The PSL runtime system is ported by processing it with this system as just another exported program. This paper discusses the FORTH-based object file format (which is general enough to handle other languages besides Lisp) and the FORTH-based relocating loader.

Porting PSL: PSL [Griss 82] is ported by first moving its object file loader (FASLIN) to the target machine. This is done by cross-compiling FASLIN on a host machine. It is then linked using a machine-independent linker (PLINK) [Carr 85] which reads and generates the FORTH-based object file format. Once linked, it is transferred to the target machine then loaded and executed using a variant of the relocating loader discussed later. FASLIN then incrementally loads cross-compiled object files which define the rest of the PSL runtime system. Once completed, the runtime system is used to compile the compiler on the target machine, thereby making the ported system independent of the host machine. Besides assisting in the porting of PSL, PLINK and the machine-specific loader give us the ability to develop programs in the full Lisp environment then "export" them to run as standalone programs.

Motivation: The writing of the machine-independent linkers and machine-dependent loaders is simplified by using FORTH language statements as relocation directives. Coupling this with an ASCII format gives several benefits. First, we can use built-in FORTH words to process object files when writing loaders. Second, we can use standard text tools to examine and modify the object files - very useful during a port. Third, it provides a standard for number representation, freeing us from dealing with different number systems, byte ordering, etc., which may occur between the host and target machines. Finally, many target machines do not yet have sophisticated file-transfer programs (FTP). The ASCII format allows us to use simple FTP programs over serial lines.

The File Format: The relocatable object file is a mixture of absolute code (represented by numbers) and directives. For example, before linking, two separately compiled files may look like (source - left, object code - right):

```
(global '(x y))          ~ seg code ~
                         0 0 0 0 ~ code 4 + entry-point ~
(de om ()                ~ code 4 + def-fca om ~
 (setf x 10)             MOVL 10 ~ q x vca dw ~
 (setf y  8)             MOVL  8 ~ q y vca dw ~
 (gcd))                  JMP ~ q gcd fca dw ~
                         ~ 32 len code ~
                         ~ seg data ~
                         0 0 0 0 ~ data 0 + def-vca y ~
                         0 0 0 0 ~ data 4 + def-vca x ~
                         ~ 8 len data ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(de gcd ()               ~ seg code ~
 (while (~= x y)         0 0 0 0 ~ code 4 + def-fca gcd ~
  (if (> x y)            CPML ~ q y vca dw ~ ~ q x vca dw ~
   (setf x (- x y))      BNEQ 5
   (setf y (- y x))))    MOVL REG11 REG1
 x)                      BRB 45
                         CPML ~ q y vca dw ~ ~ q x vca dw ~
                         ...(etc.)
                         ~ 75 len code ~
```

In this example absolute code is shown as VAX mnemonics rather than opcode and addressing mode numbers. A tilde (~) marks the beginning and end of a directive. Inside of directives, numbers are pushed on the parameter stack and other symbols are expected to be executable FORTH words, as in a standard FORTH interpreter. Outside of directives we just deposit the numbers in the current segment. The compiler and loader must agree on the size of the numbers, usually the size of an addressable cell (8 bits on the VAX).

Function entry points are defined with DEF-FCA and referenced with FCA. Global variables are defined with DEF-VCA and referenced with VCA. PLINK is used to resolve function and variable references and to collect each file's contribution to the various segments into contiguous segments. An object file is ready for loading once all references are resolved, except segment base addresses, which are determined at load-time. Once PLINK processes the above files the resultant relocatable object is:

```
~ 8 len data ~
~ 107 len code ~
~ seg data ~
0 0 0 0 0 0 0 0
~ seg code ~
0 0 0 0
~ code 4 + entry-point ~
MOVL 10 ~ data 4 + dw ~
MOVL  8 ~ data dw ~
JMP ~ code 32 + 4 + dw ~
```

```
0 0 0 0
CPML ~ data dw ~ ~ data 4 + dw ~
BNEQ 5
MOVL REG11 REG1
BRB 45
CPML ~ data ~ ~ data 4 + dw ~
...(etc.)
```

Segment length definitions come first so that all segments can be created before loading. Next, a segment definition initiates sequential loading into the named segment. Relocatable references into the various segments are handled by making offsets from the segment base such as: ~ data 4 + dw ~. In this case, DATA is replaced with its base address, 4 is added to this address and the result is deposited into the next word with the DW function. Since word size varies from machine to machine, a machine specific function such as DW must be supplied for each new machine. A BNF for the file format of a resolved object file (as in this small example) is:

```
<file>                        ::= { number | <directive> }*
<directive>                   ::= ~ <command> ~
<command>     ::= <segment-directive> | <segment-length-directive> |
                  <relocation-directive> | <entry-point-directive>
<segment-directive>           ::= seg <identifier>
<segment-length-directive> ::= <length> len <identifier>
<relocation-directive>        ::= <relocation-info> dw
<entry-point-directive>       ::= <relocation-info> entry-point
<relocation-info>             ::= <segment-id> {<offset> +}*
<offset>                      ::= The offset from segment base.
```

The Relocating Loader: This format is machine-independent and straightforward to process. PLINK uses an extended version of this format. To create a loader we need to write a top-level function to open the file, read and process one symbol at a time until the end of the file is reached, and then close the file. When we read a symbol, if it is an executable FORTH word, we execute. If it is not a FORTH word we assume it is a number, in which case we either deposit it into the current segment or push it onto the stack if we are inside a directive. Assuming the open-read-process-close functions are in place we can define the directives as FORTH words (here written in Bill Sebock's - Princeton University - VAXFORTH):

```
0 var &directive          ( 1 -> in directive)
0 var &loading-point      ( address counter)
0 var &entry-point        ( start of instructions)
32 constant blank         ( ascii blank)

: ~     ( --- )           ( toggle directive mode)
  &directive @ com &directive ! ;

: len   ( n ---name )     ( modeled after definition of ARRAY)
  here 1 and allot        ( make sure even addresses)
```

```
   create                 ( get segment name from input stream)
   here                   ( beginning of storage for segment)
   swap dup allot         ( allocate the storage - N bytes)
   0 fill ;               ( initialize storage to zero)

: seg   ( ---name )
   blank word spush       ( get segment ( put segment base in address ctr)


: dw    ( 32-bit-number --- )
   &loading-point @ !      ( 32 bit store )
   4 &loading-point +! ; ( increment address counter by 4 bytes)

: entry-point ( 32-bit-number --- )
   &entry-point ! ;        ( Save it for later execution)
```

Conclusion: FORTH's syntax avoids issues of precedence. Its built-in symbol table, stack and its execution model make it straightforward to write linkers and loaders based on this file format. This format is simple, easy to manipulate (with text editors) and easy to extend (by defining new FORTH words). It allows us to concentrate on other issues during the porting of PSL. However, these object files are generally twice as large as a typical binary object format and slower to load. Once PSL is ported we revert back to using a standard binary format.

Acknowledgments: Work supported in part by the Burroughs Corporation, the Hewlett Packard Company, the International Business Machines Corporation, the National Science Foundation under Grant Numbers MCS81-21750 and MCS82-04247, and the Defense Advanced Research Projects Agency under contract number DAAK11-84-K-0017.

References

[Carr 85] Carr, H.
A Portable Linker for Portable Standard Lisp.
Master's thesis, Department of Computer Science, University of Utah, January, 1985.

[Griss 82] Griss, M. L.; Benson, E.; Maguire, G. Q. Jr.
PSL: A Portable LISP System.
In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 88-97. ACM, Carnegie-Mellon University, Pittsburgh, Pa., 1982.