A Portable Linking Loader

Reinaldo S. C. Braga
Michael A. Malcolm
Gary R. Sager

Department of Computer Science
University of Waterloo

Research Report CS-76-18

April 1976

A Portable Linking Loader

Reinaldo S. C. Braga
Michael A. Malcolm
Gary R. Sager

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

abstract: We are currently engaged in the design and implementation of portable minicomputer system software. In this paper we focus on an important component of this system: a highly-portable linking relocating loader. In our system, the language processors produce a set of directives which contain all of the information necessary to load a program for a specific machine. The loader is implemented as an abstract machine which "executes" these directives to build an absolute form of the executable module. With a proper set of directives, the loader can load for any machine architecture. The loader itself is portable and can therefore function as a cross-loader using the same input directives.

We discuss, in detail, the design of the loader and its directives. We also give some performance figures which indicate the costs of this approach.

1. introduction

As minicomputer hardware continues to decrease in cost and increase in speed and reliability, and software production costs increase, software portability becomes increasingly important. *Portability* is a measure of the effort required to move a piece of software from one environment to another. A piece of software is said to be *portable* over a set of machine environments if it is significantly easier to port to a new environment than to reimplement. To be practical, it should not require much more effort to initially implement a portable version of a program than it takes to implement it in a machine-dependent form; and the portable version should not be prohibitively less efficient.

Techniques for writing portable programs entail programming in some abstraction of the machine environments over which portability is desired. One well-known technique is to devise an "abstract machine" appropriate for the application which can be implemented on a variety of machines. The abstract machine code may be executed interpretively or translated to the target computer's machine language using a portable macro processor or vendor-supplied macro assembler.

Existing methods for achieving portability share common shortcomings: the interface between programs and various operating systems and/or support software is usually crude at best, severely limiting the applications for which portable programming is possible. This problem is difficult to avoid since many operating systems provide only primitive services, and available support software varies widely from one system to the next.

We may enhance the portability of a large set of programs by writing them in a high level language and by developing a highly portable compiler for the language. A large degree of the portability of the set of programs will then depend upon the portability of the compiler. One difficulty of this approach is the reliance upon a pre-existing assembler and/or relocating loader of the target. Each system has different restrictions and conventions for external symbols and linkage. Relocation capabilities differ. Libraries are either not implemented or follow no standard search rule. Hence, serious development of portable systems of programs for use in a variety of operating environments is difficult simply because one cannot insure that external names remain unique, that libraries are searched in the proper order, or myriad other vagaries of the pre-existing software on the target machines.

Many of these standard problems of portable software can be avoided if one is able to port support software with user programs. That is, the more we port of the program's environment, the fewer aspects of the program's "system dependence" cause problems. Of course, porting a system presents a number of new problems, many of which are difficult to solve. However, these problems must be addressed only by the persons who port the system, not by every user programmer who wants to write a portable program.

In this paper we present our solution to a major share of these difficulties: a highly-portable linking relocating loader. In our system, the language processor produces a set of directives which the loader "executes" to build an absolute form of the executable program. With the proper set of directives, the loader can load for any machine architecture. The loader portability is achieved by implementing it in a high-level language which is discussed in the next section.

2. the systems implementation language Eh

We have designed and implemented a language which we call Eh [Braga (1976)]. Eh is based on the language B, which was developed at Bell Laboratories. Along with its predecessor, BCPL [Richards (1969)], and successor, C [Ritchie (1974)], B has been extensively used as a systems implementation language.

Like its companion languages, Eh is a language in which a program must be written as a collection of functions. The functions are dynamically nested, so that all program variables are either global to the whole program, or local to a particular function and dynamically allocated in a stack. Eh is typeless and word oriented; it contains a powerful set of operators and control structures. Its code is reentrant and functions may be recursive. Thus, it is well suited to systems programming.

The Eh compiler is designed for portability. It consists of two separate programs, or passes. The first is a syntax-directed parser which produces an intermediate language called EhIL. The second generates relocatable load code from EhIL. The first pass is the same for all machine implementations of Eh. The second pass generates machine-specific directives for the loader. The second pass must be partially rewritten to generate code for a new machine. Both passes are written in Eh, that is, they are self-compiling. In addition, Eh requires very little input/output support for system programs such as the compiler and loader.

It is important to observe that although the load code generation pass is machine-specific, it *is not* machine-dependent. The compiler can execute on any machine to generate code for the target machine. In this way it can function as either a compiler on the target, or as a cross compiler on another machine.

Of the alternative methods for code generation, we have already eliminated those which must rely on existing software on the target machine. However, we still might consider those compilers which translate programs into absolute modules directly, bypassing the assembling, linking and relocating steps altogether. We feel that this type of compiler is inappropriate for maintaining a large library of interrelated systems programs, parts of which may be written in languages other than Eh. In particular, we wish to interface with programs written in assembly language and assembled by a rudimentary assembler [Stafford (1976)] which outputs relocatable load code compatible with that output by the Eh compiler.

3. the linking loader

A linking loader has three major functions: (1) the linking of modules via external symbols, (2) the relocation of addresses within modules, and (3) loading the executable module into memory, and (perhaps) initiating execution of it. The relocation and loading steps may be combined, depending on the machine architecture. The three functions may be accomplished by one or more programs.

Our *Universal Loader* is an abstract machine which takes the language processor output (load code) and executes it as a set of directives to build an executable program. With the proper use of the directives, the loader can load for any machine architecture we have encountered. Several of the load code directives describe to the loader the addressing structure and relocation procedures to be followed; this machine-specific information can be included as a standard "prolog" which is the first load code module presented to the loader.

We have implemented the Universal Loader as two distinct phases; first, the *linking relocator* performs all linking and relocation, and outputs an absolute form of the executable module which is loaded by the second phase, or absolute loader. The first phase generates an absolute module instead of a core image for two reasons:

(1) during a port, we must communicate executable programs to a bare target machine. This final communication step is easily accomplished by an absolute loader which can usually be coded in a small number of machine instructions. For example, an absolute loader for the Data General NOVA has been programmed in 39 instructions.
(2) the Eh compiler relies on a backplugging capability during code generation, and ULD doesn't resolve these since it is much easier to take care of with an absolute loader.

The linking relocator is written in Eh and forms the heart of the Universal Loader; we shall henceforth refer to it as "ULD". ULD makes two passes over the load code before generating the absolute module. This means that ULD need not reside in the same storage as the absolute module it is building, and therefore does not impose any size restrictions. The two pass operation also avoids the necessity of forming lists of unresolved references. In the first pass, symbol tables are built and the sizes of program modules are computed. In the second pass, external references are resolved, relocations are performed, and the absolute module is generated.

In order to avoid machine dependencies, ULD works with 8-bit bytes. Arithmetic is done as a byte-serial operation by an algorithm which requires at least 9 bits per word on the machine executing ULD. It is also assumed that pointers into the symbol tables can be stored in a word. On machines in which the largest addressable cell is insufficient for these requirements, the Eh compiler generates code which simulates a "word" using two or more cells.

A list of the load directives interpreted by the ULD machine and their semantics is given in the appendix.

The abstract machine which defines ULD is illustrated in Figure 1. The control unit reads command bytes into the command byte register (CBR) and operates on the remainder of the directive and the registers based on the CBR and the PASS (1 or 2). The working register (WR) is a 32 byte register used to hold data to be relocated and output as absolute module records. The relocation unit is capable of performing the byte-serial arithmetic and logical operations necessary for relocation. It adds symbol values from the working symbol dictionary (WSD) into fields of the WR indicated by

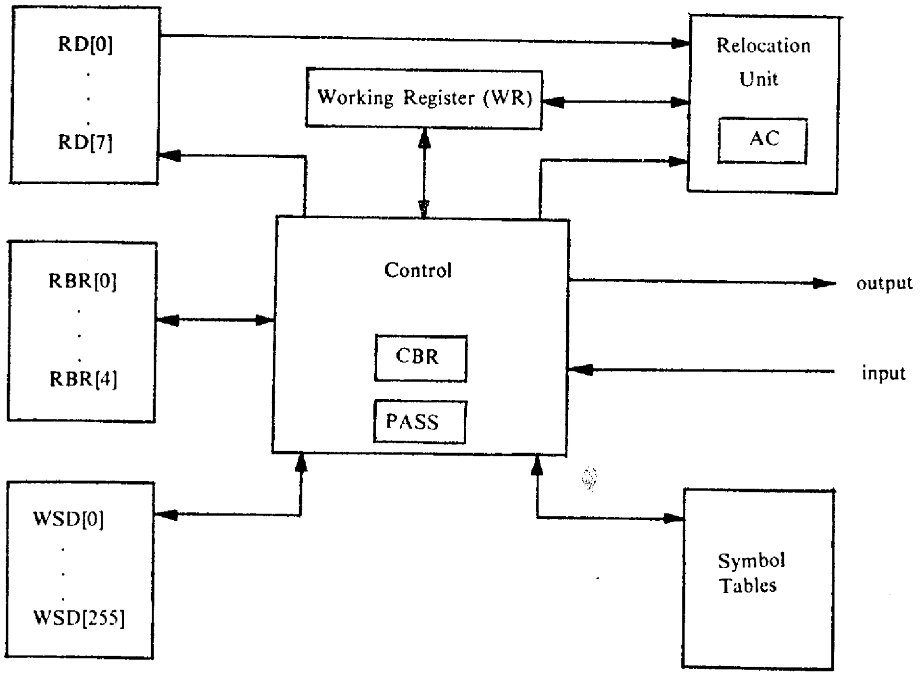

Figure 1: Structure of the ULD "abstract machine".

the specified relocation descriptor (RD[0] through RD[7]). The AC register is used to hold intermediate results. The relocation base registers (RBR[0] through RBR[4]) are used to establish the absolute locations of symbols to be entered into the symbol table during the first pass. Multiple RBRs are provided so each module can be loaded into several "sections" of memory to take advantage of addressing features of the target machine.

Relocation is accomplished by first loading the working register, then issuing the directive to relocate and output. This directive operates on data in the working register, a relocation descriptor and a symbol value from the working symbol dictionary. The relocation descriptor contains a shift count which allows division of a symbol value by a power of 2, and a string of mask bytes used to define a field within the working register. The shift count is used because pointer values in Eh are sometimes different than addresses in the machine. Up to 8 different relocation descriptors can be defined at any time. If more are needed, the compiler can redefine relocation descriptors as many times as necessary during a load. The working symbol values are kept in the symbol table as strings of bytes. In each module, up to 256 different working symbols can be referenced.

In order to clarify the role of the working register, relocation descriptors and working symbols, we will describe the operation of the relocation unit in more detail. The relocate and output directive (issued after a load working register directive) contains, for each field to be relocated, the following information:

- the byte of the working register containing the rightmost bit of the field
- the relocation descriptor number
- the working symbol number

This information is encoded as byte pairs in the relocate and output directive.

Note that a different relocation descriptor must be defined for each possible address size and each possible rightmost bit position in a working register byte. We will show two examples: if the target machine has a word size of 20 bits, and the address is located in the lower 10 bits, a target machine word would be defined as 3 bytes with the first byte having data only in the rightmost 4 bits. The descriptor for relocating such addresses contains:

- shift count: 0
- mask bytes: 003 377 (octal notation)

If the address could appear in the upper 10 bits of a word, then another descriptor would contain:

- shift count: 0
- mask bytes: 017 374 000

Using the information from each byte pair in the relocate and output directive, ULD locates the byte of the working register containing the rightmost bit of the field to be relocated, then from the relocation descriptor mask, the field within the working register which must be modified is isolated. The working symbol number tells us which symbol table entry to use for the relocation (i.e. what value to add into the field). After all byte pairs in the directive are processed, one record is output in absolute load code form:

1. inter-record bytes (defined by the B directive)
2. address bytes (typically 2 or 3)
3. byte count
4. data bytes
5. checksum byte

The final record contains the starting address (defined by the S directive) and a byte count of zero. The inter-record bytes option is provided for ease of locating and positioning of the records if paper tape is used. In the cases of magnetic tape or disc peripheral devices, the B directive is not used.

4. Experience with ULD

We have tested ULD by loading programs for the Honeywell 6060, Data General NOVA/2, Microdata 1600/30, and TI 990. The ULD loader has executed on the Honeywell and Microdata. The ULD program consists of 387 Eh statements (approximately 9000 characters); it requires 7,040 36-bit words on the Honeywell, and 13,950 bytes on the Microdata. One may obtain an idea of the relative sizes of programs and load modules from the data of Table 1, in which we present the results of compilations and loads for several machines. The figures given in Table 1 are for an Eh program called Thoth, a small real-time executive [Melen (1976)]. Thoth consists of 237 statements which implement 21 functions; without comments or indentation, Thoth source is 6385 characters. The EhIL module output by the first pass of the compiler is 5935 characters.

Table 1: Sizes of ULD load code, absolute modules and executable modules for Thoth. All figures given below are in 8 bit bytes.

target machine	ULD module	absolute module	core image
HIS 6060	9482	5225	4153
NOVA	11027	5332	3854
Microdata 1600/30	7568	4136	3250

The internal structures of the four machines we have loaded for show a great deal of variation:

(1) Honeywell 6060: 36 bit word, word addressing, 18 bit address, the address may appear in the left or right half of the word
(2) NOVA: 16 bit word, word addressing, 8 or 15 bit address
(3) Microdata: 16 bit word, byte addressing, 8 or 15 bit address, variable-length instructions
(4) TI: 16 bit word, byte addressing, 8 or 16 bit address

Converting the Eh compiler to generate code for a new "target" machine requires four important tasks:

(1) write new code generators for the second pass of the Eh compiler
(2) construct a prolog module for the target
(3) implement a few primitive input/output functions
(4) write an absolute loader for the target

The first of these tasks will consume by far the most time. Constructing the ULD prolog requires less than one hour. For the input/output functions, we have used two approaches; first, we have hand-coded ULD modules for the functions, and second, we have used a rudimentary "pre-fabricated" assembler [Stafford (1976)] to generate the ULD modules. This assembler can be adapted to assemble code for a new machine in approximately one day by an experienced person. The assembler has been developed mainly to alleviate the problems of coding interrupt-driven input/output functions. As was mentioned in the previous section, the absolute loader requires only a small number of machine instructions. If necessary, the absolute loader can be hand coded in machine language and keyed into the target's memory.

Once these tasks have been accomplished, one may concentrate on the physical movement of the system to a new (target) machine from the old (host) machine. In order to get the Eh compiler and ULD to the target machine, both must be compiled on the host and the primitive input/output functions must be prepared on the host. The version of ULD which executes on the host can then be used to create an absolute module of the ULD program for the target. At this point, one has the choice of creating the absolute module of the Eh compiler on the host or the target, and as more systems software moves to the target, the potential uses of the software for cross and on-site operations grow.

5. conclusion

Our primary use of the software described in this article has been to compile, link, relocate and load programs using the Honeywell 6060; core images are then transmitted to the appropriate minicomputer for execution. The reason for the cross-operation is the lack of an adequate editor and file system on the other machines. Our solution to this problem will of course be to implement these as further components of our portable programming system. These will require support from a portable operating system which is currently under development.

We are beginning to feel liberated from the jungle of software which confronts anyone who deals with more than one minicomputer; ULD is one of the key tools in this liberation. With it, we have recognized and overcome an obstacle which others have typically treated as an integral part of their target machine; The result is that they have a broader interface to fit on their target, with resultant limitations on the complexity of the software systems they can support. We are finding that we are already developing highly inter-related components for our portable system, secure in the knowledge that we can support these inter-relationships on any new machine.

6. Acknowledgements

We are pleased to acknowledge several students who have helped in the development of ULD and related software: Randy Bruce, Doug Martin, Lawrie Melen, Gary Stafford and Fred Young.

We gratefully acknowledge the National Research Council of Canada for their support, in part, of the research reported in this paper.

7. References

Braga, R. (1976), "the programming language Eh", unpublished manuscript.

Colin, A. J. T., K. Shorey and W. Teasdale (1975), "the translation and interpretation of STAB-12", *Software Practice and Experience,* vol. 5, 123-138.

Griswold, R. E. (1972), *The Macro Implementation of SNOBOL 4.* W. H. Freeman and Co., San Francisco.

Melen, L. (1976), "a portable real-time executive", unpublished manuscript.

Poole, P. C. and W. M. Waite (1973), "portability and adaptability", in *Lecture Notes on Econ. and Mathematical Systems, no. 81: Advanced Course on Software Engineering,* Springer-Verlag, 183-277.

Richards, M. (1969), "BCPL: a tool for compiler writing and system programming", *Proc. Spring Joint Computer Conf.,* 557-566.

Ritchie, D. M. and K. Thompson (1974), "the UNIX time-sharing system", *Comm. A.C.M.,* vol. 17, no. 7, 365-375.

Stafford, G. J. (1976), *The Last Assembler reference manual.* unpublished manuscript.

8. Appendix: ULD Directives

The general form of a ULD directive is:

c s d...d k

where:

c	is the command byte
s	is the number of data bytes
d...d	are the data bytes
k	is a checksum (computed as the exclusive-or of c, s, and d...d).

Individual directives are listed below. Refer to the section describing the structure of the ULD abstract machine for a more complete description of the registers, tables, etc. Since directives may have different actions during passes 1 and 2, the descriptions are divided into explanations of their actions for the two passes.

G **global symbol definition**

G s rb aa nn...n k

where:

rb	indicates the relocation base register (RBR)
aa	is a two-byte offset value relative to RBR[rb]
nn...n	is the symbol name (<= 16 characters)

pass 1: If the symbol is already in the symbol table and defined, an error message is printed and the second pass is not executed. Otherwise, the value of the symbol is defined to be the result of aa+RBR[rb], and the symbol and its value are added to the symbol table. If this is the first global symbol in the module, it is taken as the "module name".

pass 2: The first global symbol definition encountered in a module causes WSD[0] to be set to a pointer to the symbol's entry in the symbol table. This symbol, the module name, determines the local symbol table to which local symbol references are made.

T **local symbol definition**

T s rb aa nn...n k

where:

rb	indicates the relocation base register
aa	is a two-byte offset value relative to rb
nn...n	is the symbol name (<= 16 characters)

pass 1: The value of the symbol is defined to be the result of aa+RBR[rb], then the symbol and its value are added to a symbol table which is associated with the module in which the definition occurs.

pass 2: nothing.

g global symbol reference

g s p nn...n k

where:

p	is a number which is used to refer to this symbol in subsequent O directives.
nn...n	is the symbol name.

pass 1: If the symbol name is not already in the symbol table, an entry is made for it and its value is set to "undefined".

pass 2: The referenced symbol is accessed by placing a pointer to its symbol table entry in WSD[p] .

t local symbol reference

t s p nn...n k

where:

p	is a number which is used to refer to this symbol in subsequent O directives.
nn...n	is the symbol name.

pass 1: nothing

pass 2: The referenced symbol is accessed by placing a pointer to its symbol table entry in WSD[p] .

M beginning of module

M 0 M

pass 1: used to detect when the next global symbol should be used as a module name.

pass 2: same as in pass 1, plus the WSD is initialized to zeros to prepare for loading of the new module.

E end of module

E 0 E

This directive is used to indicate the end of a relocatable object module. The next directive should be either M or $.

I **increment relocation base**

I s rb aa k

where:

rb	is the relocation base register to be incremented
aa	is a two-byte increment value
pass 1:	RBR[rb] = RBR[rb] + aa
pass 2:	RBR[rb] = RBR[rb] + aa
note:	the use of a two-byte increment here and in the G and T directives gives a practical limit of 2**16 addressable units per relocation base register for any single relocatable object module. However, the output absolute module can be arbitrarily large.

D **relocation descriptor definition**

D s d b mm...m k

where:

d	is the descriptor number
b	is a shift indicator (see explanation of O directive below)
mm...m	is a mask which defines the field to be relocated
pass 1:	b and mm...m are copied into RD[d].
pass 2:	b and mm...m are copied into RD[d].

L **load working register**

L s dd...d k

where:

dd...d	is a string of bytes to be copied into the working register.
pass 1:	nothing
pass 2:	The string is copied into the working register, starting at the leftmost byte. The contents of WR will later be output (by the O directive) as an absolute module record. Thus, the contents of WR must resemble an absolute module record; the most convenient form of these records will vary according to the machine and the device used. As an example, we may decide that each absolute record should consist of a two-byte address followed by a byte count and the number of data bytes indicated by the byte count. In this case, the first two bytes of WR must be converted to an absolute address by the next O directive.

O relocate and output working register

O s cw cw ... k

where:

cw	is a two-byte pair, as follows:
c	has two fields: the first 5 bits (bn) give a byte number in the working register; the remaining 3 bits (rn) is a relocation descriptor number.
w	is an index into the working symbol dictionary (WSD).
pass 1:	nothing
pass 2:	For each cw pair, the following relocation algorithm is executed. The symbol table value pointed to by WSD[w] is copied into the AC register, then the AC is right-shifted (as in a division by a power of 2) the number of bits specified in the b field of relocation descriptor RD[rn]. Next, a field F in the working register is isolated by aligning byte bn of WR with the rightmost byte of the mask bytes from RD[rn]; the 1 bits in the mask bytes define the field F. The value in F is added to the (shifted) value in AC and the result is stored in F. In the event of overflow (the sum cannot fit in F) an error message is printed. Note that linking is a special case of relocation. Because of the information collected during pass 1, a "link" to another module is established by adding its symbol table value into the field F rather than the current module symbol table value (from WSD[0]).

R rb definition

R s rb aa...a k

where:

rb	is the relocation base register to be set
aa...a	is the value to be stored into RBR[rb].
pass 1:	RBR[rb] = aa...a
pass 2:	RBR[rb] = aa...a

S start symbol definition

S s nn...n k

where:

nn...n	is a global symbol which must appear in the input module.
pass 1:	nothing
pass 2:	An appropriate starting address record is placed at the end of the absolute module.

B **inter-record bytes**

B s dd...d k

where:

dd...d	is a string of bytes.
pass 1:	nothing
pass 2:	the bytes dd...d are placed between each absolute module record output by ULD. Such inter-record bytes may be useful in certain media such as paper tape. If no B directive is specified, a null string is used.

$ **end of file**

$ 0 $

This directive denotes an end of file. It is used due to the lack of end of file indicators on some devices.