Assembly Language as Object Code

DOUGLAS W. JONES

Department of Computer Science, University of Iowa, Iowa City, Iowa 52242, U.S.A.

SUMMARY

The set of meanings that may be encoded in most object languages is a subset of the meanings that may be encoded in most assembly languages. Careful exploitation of this fact in the design of the SMAL assembly and object language allows the SMAL assembler itself to be used as a linkage editor, thus eliminating the need for an expensive and often misunderstood system program. The SMAL assembler is no more complex than many assemblers in common use today, nor are the relevant aspects of the SMAL language particularly unique.

KEY WORDS Object code Linkers Loaders Assembly language Intermediate code Relocation

INTRODUCTION

SMAL (pronounced 'small') is a symbolic macro assembly language originally developed for teaching purposes at the University of Iowa. SMAL has been designed so that it includes object and loader languages as sublanguages of itself, thus allowing a SMAL assembler to serve as a linkage editor. This paper describes those aspects of SMAL that allow this to be done; the complete machine independent version of SMAL is documented elsewhere.[1] A SMAL assembler has been written in Pascal which is 2142 lines long, including support for macro and conditional assembly, but excluding support for any particular machine instruction set. Undergraduates have added support for various instruction sets to the SMAL assembler in less than a week each, and some machine specific versions of SMAL are now in production use.

The conventional model for translation from an assembly language program to an executable machine language memory image dates back at least to 1957 with the IBM 704 FORTRAN II system,[2] and it persists to the present.[3–5] This model involves four distinct program representations: the assembly language, object language, load language and machine language. Files of text in these languages are referred to as source files, object files, load files, and memory images, respectively. The model involves three translation programs: the *assembler*, which converts a source file to an object file; the *linkage editor*, which converts one or more object files to a load file; and the *loader*, which converts a load file to a memory image. Each of these programs may have a control language that is used to select input files or processing modes.

Formally, each of the programs involved may be considered to perform a homomorphic (many-to-one meaning-preserving) mapping from the input language to the output language, where a language is defined as a set of strings, each with an associated meaning. The mapping performed by each program is characterized by the removal of certain symbolic information from the program; for example, the

0038–0644/83/080715–11$01.10

Received 29 June 1981
Revised 30 July 1982 and 28 March 1983

assembler replaces opcodes and internal symbol references by their values, and the linker replaces external symbol references by their values.

There are a number of common but minor variations on this model. For example, a linking loader can be used to replace the linkage editor and the loader, thus eliminating the load language. Alternately, the load language can be arranged so that it is a sublanguage of the object language, allowing the elimination of the linkage phase for programs that do not contain external references.

The approach used to support SMAL differs from the above model in that the loader language is a sublanguage of the object language, which is in turn a sublanguage of the entire assembly language, as illustrated in Figure 1. The SMAL assembler performs a mapping from the SMAL language to the object sublanguage, and from the object sublanguage to the loader sublanguage. The fixed point of this mapping is the loader sublanguage; thus, if the assembler is applied to a program in the loader sublanguage, the output it generates will be identical to the original program. The use of an assembler as a linker should not be confused with the use of an assembler as the final pass during the compilation process, where its primary purpose is forward reference resolution, object code generation, or the assembly of machine instructions.

The SMAL approach eliminates the need for a distinct linkage editor while preserving the useful functions of a linker. The use of textual object and loader languages results in object and load files approximately three times larger than would be expected using a compact binary encoding, and some run-time overhead is incurred in re-parsing these languages. However, the loader sublanguage is not particularly difficult to parse; a loader 63 lines long has been written in Pascal. These costs suggest that the SMAL approach is well suited to the rapid development of fully functional software environments on new systems, but that efficient special purpose object and loader languages may be needed as such systems mature.

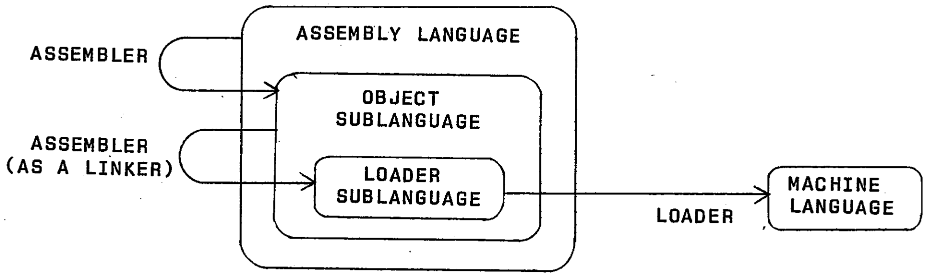

Figure 1. The SMAL model of assembly, linkage, and loading

The Thoth linker and assembler generating kit illustrate an alternative approach to the rapid implementation of fully functional programming environments on new machines,[6] but without exploiting the close relationship between the functions of linkers and assemblers. G. H. Mealy mentions the use of 'relativized code' as one solution to the problem of resolving name conflicts in large programs; this appears to have been similar to the SMAL object sublanguage, except that assembly time linkage was used, and relativized code could not be generated from code which modified the location counter.[7] Mealy also explicitly recognized that linking loaders performed an assembly function, but his generalized assembly system appears to have made no use of a distinct linkage phase. Fraser and Hanson have recently developed a linker using a textual object language that is flexible enough to be used as a rudimentary assembly language.[8]

OVERVIEW OF THE SMAL LANGUAGE

The SMAL language is a typical line-oriented assembly language based loosely on the model of MACRO-11.[9] SMAL is oriented towards machines with an 8 bit byte, a 16 bit address word, and a byte addressable memory, although modifications for the support of other memory addressing structures are obvious. SMAL is a free-format language, using a colon to terminate labels, and a semicolon to mark the start of comments. The SMAL language includes all of the directives listed in Table I; the various sublanguages are based on a subset of this list.

Table I

SMAL directive	Use
Data storage:	
B<expr>	assemble one byte
W<expr>	assemble one word
ASCII<string>	assemble a text string
External symbol control:	
EXT<id>	import an external symbol
INT<id>	export an internal symbol
COMMON<id>,<expr>	declare common name and size
Macro and Conditional:	
IF<expr>	begin conditional assembly
ELSEIF<expr>	begin subcondition
ELSE	invert condition
ENDIF	end conditional assembly
MACRO<id> <formal>	begin macro definition
ENDMAC	end macro definition
Miscellaneous:	
USE<string>	insert text from named file
LIST<expr>	control assembly listing
END	end of file

All of these directives have counterparts in other assembly languages, although it is worth noting that the COMMON directive only declares the name and size of a FORTRAN-like common region, with no effect on the assembly time location counter. As in MACRO-11, the assembly location counter is represented by the special symbol . (period), the equals sign is used for assembly time assignment, and the assembly origin may be set by assigning a value to the location counter. Unlike most other assemblers, the relocation base may be changed by assembly time assignment; thus, if C is the name of a common, . = C sets the location counter to the start of that common. Because of this, it is possible to set an absolute assembly origin by merely assigning an absolute value to the location counter.

THE LOADER SUBLANGUAGE

The loader sublanguage is considerably simpler than the full SMAL language, not only because fewer directives are supported, but because the free format features are eliminated, the forms of the allowed expressions are restricted, forward references are prohibited, and only one symbolic name is allowed. The following extended BNF grammar describes the entire loader sublanguage to the character level:

```
<load file>::= R=.<end of line>
               { <load directive><end of line> }
               <end of file>
<load directive>::= .=<load value>
                  | B=<load value>
                  | W=<load value>
<load value>::= <hex number>
              | <hex number>+R
              | <space>R
<hex number>::= #<hex digit> {<hex digit>}
```

Each load file begins with a prefix that sets the symbol R, the relocation base, to the current location. The file continues with a sequence of directives that either change the current location or load a value and advance the location appropriately. The loaded values may be either absolute values, indicated by a simple hexadecimal number, or they may be relocatable values, indicated by a hexadecimal number with R added to it. Hexadecimal numbers are prefixed with #; this serves both to indicate the number base and to separate the number from any preceding symbols.

The following assembly listing illustrates the result of assembling a simple program written in the SMAL loader sublanguage. The object code produced by the assembler in processing this program is identical to the program itself.

```
line address value        listing
1                         ]R=.
2  +0000:  12             ]B#12
3  +0001:  1234           ]W#1234
4  +0003:+3210            ]W#3210+R
5                         ].=#0006+R
6  +0006:  34             ]B#34
7                         ].=#0005
8   0005:+0001            ]W#0001+R
9   0007:  0005           ]W#0005
```

The ± before a hexadecimal number in the address and value columns indicates that the number is relocatable. Lines 3, 4, 8 and 9 illustrate all possible combinations of absolute and relocatable values to be stored at absolute or relocatable load addresses. Except in system programs, absolute load addresses are rarely used. On the other hand, absolute load values are very common; they correspond to opcodes, constants, and relative addresses.

THE LINKER SUBLANGUAGES

When the SMAL assembler is used as a linkage editor, three distinct sublanguages come into play: The object sublanguage, the linker control sublanguage and the library sublanguage. Of these, the object sublanguage is the most obvious; it is the language in which object files are encoded. The linker control sublanguage tells the assembler which object files are to be linked and which libraries are to be searched. Finally, the library sublanguage is used to organize the directories of object libraries; as such, a library sublanguage program consists of a specially organized sequence of references to the object files making up some library.

The object sublanguage

The SMAL object sublanguage includes three features that are not present in the loader sublanguage: First, values may be relocated relative to any linkage time symbol, thus allowing assembly of external and common references. Secondly, each object file contains an optional sequence of linkage time symbol definitions. Finally, each object file may contain common definitions that first set the size of a common if the size was not previously set, and then allocate storage if storage was not previously allocated. An extended BNF grammar for the SMAL object sublanguage is as follows:

```
<object file>::= R=.<end of line>
               { <object directive> <end of line> }
               { <internal definition> <end of line> }
                 <end of file>

<object directive>::= .= <object value>
                    | B<object value>
                    | W<object value>
                    | <common definition>

<internal definition>::= R<identifier> = <object value>

<object value>::= <hex number>
                | <hex number> + <relocation base>
                | <space> <relocation base>

<relocation base>::= R | R<identifier>

<common definition>::= IF\DEF(S<identifier>)<end of line>
                          S<identifier> = <hex number> <end of line >
                       ENDIF<end of line>
                       IF\DEF(R<identifier>)<end of line>
                          R<identifier> = C<end of line>
                          C = C+S<identifier> <end of line>
                          CT = .<~~end of~~ line>
                          . = C<end of line>
                          . = CT<end of line>
                       ENDIF
```

That the object sublanguage includes the loader sublanguage can be seen by comparing the two grammars and noting that the differences consist entirely of additional alternatives or optional parts in the definition of the object sublanguage.

Two SMAL object files may be linked by concatenating them and then assembling the result. Ignoring, for the moment, the complexity of common definitions, the following pair of SMAL assembly language programs demonstrates the basic function of the object language; when these programs are linked, the result is two cyclically linked pairs of pointers.

```
1                    ]; FIRST PROGRAM
2                    ]   INT A
3                    ]   EXT B
4  +0000: +0000      ]A: W B
5  +0002: +0002      ]   W B+2
1                    ]; SECOND PROGRAM
2                    ]   INT B
3                    ]   EXT A
4  +0000: +0000      ]B: W A
5  +0002: +0002      ]   W A+2
```

Assembling the concatenated object code for these programs produces the following listing:

```
1                    ]R = .
2  +0000: +0004      ]W RB
3  +0002: +0006      ]W#0002+RB
4                    ]RA = R
5                    ]R = .
6  +0004: +0000      ]W RA
7  +0006: +0002      ]W#0002+RA
8                    ]RB = R
```

The load file corresponding to the above assembly listing is:

```
R =
W#0004+R
W#0006+R
W R
W#0002+R
```

In practice, files to be linked are not directly concatenated as shown above; instead, the linker control sublanguage is used to control which object files are read by the assembler.

Each common definition in a SMAL source program introduces a block of 10 lines into the object file. This 10-line block uses the conditional assembly features of SMAL to determine if the size of the common has been specified and to determine if space has been allocated for it. If the common size has not previously been specified, for example, by linker control language commands or by a previous common definition, the common size is set to the size specified in the current common definition. If storage has not previously been allocated for the common, storage is allocated in the common pool maintained relative to the linkage time symbol C. The final lines of the object language common definition inform the assembler of the highest location used relative to the default relocation base.

Throughout the object sublanguage, user-defined external symbols and common names are prefixed with the letter R. This prevents conflicts between user-defined symbols and the default relocation base R, the default common allocation base C, and the linkage time temporary CT. Similarly, the symbolic name for the size of each common region is made from the common name prefixed with the letter S.

The linker control sublanguage

Although the assembler can link object files if they are simply concatenated before assembly, the linking process is normally controlled by the linker control sublanguage. This sublanguage is based on the SMAL USE directive, which causes the assembler to read the named file before continuing with the original source file. For example, files A, B, and C may be linked by assembling the following linker control file:

```
USE "A"
USE "B"
USE "C"
```

When the files being linked reference common regions, or when various default linkage actions are to be overridden, the linker control file must contain additional information. The basic SMAL linker control sublanguage is described by the following extended BNF grammar:

```
<linker control program> ::= { <linker control> }
                             { <object specification> }
                             [ <common specification> ]
                               <end of file>

<linker control> ::= C = <number> <line end>
                   | . = <number> <line end>
                   | R<identifier> = <number> <line end>
                   | S<identifier> = <number> <line end>

<object specification> ::= USE<quoted string> <line end>

<common specification> ::= C = . <line end>

<line end> ::= [;<any text> ] <end of line>
```

Unlike the object and loader sublanguages, the linker control sublanguage is intended for human use; thus it is free format, and comments are allowed.

As described above, a linker control program begins with an optional sequence of linker control directives. These are used primarily to override default actions that would otherwise take place. For example, the form C = 4096 causes common storage to be allocated starting at absolute address 4096, and the form . = 1024 causes the resulting load file to begin at absolute location 1024 instead of relocatable zero. The default allocation of individual common regions may be overridden; for example, if COM is a common name, SCOM = 50 sets the size of COM to 50 bytes, and RCOM = 16384 sets the base of COM to the absolute address 16384. If, by default, common storage is to be allocated immediately after the storage for all other object files, the form C = . must be included immediately after all object specifications.

The complete linker control sublanguage is considerably more complex than is described here, because the assembler is available to modify the linkage process. For example, it is possible to direct the assembler, as a linker, to produce an object file as output. When this is done, some external names are bound by the linker, whereas others remain unbound awaiting later linking with other modules; this introduces linkage time scope rules, a facility originally introduced in the LSS linking loader.[10] In order to do this, there are linker control directives that describe which external names are to be imported or exported from the resulting object file, and directives to describe what is to be done with common regions. These facilities are fully described in Reference 1.

The library sublanguage

The SMAL assembler, when operating as a linkage editor, is able to search object libraries and selectively link those library elements that were referenced from previously linked material. From the point of view of the linker control sublanguage, an object library is just another kind of object file. In fact, however, an object library is written in the library sublanguage, not the object sublanguage. A SMAL library file consists of a sequence of references to the object files in the library, along with information about which symbols are defined by each file. As a linker, the SMAL assembler scans the library file until it finds an entry that defines a previously referenced but as yet undefined symbol, at which point it links the associated object file before continuing the scan. The SMAL assembler has no special facilities to support library searching; instead, the library file consists of a sequence of conditional assembly directives that determine what needs linking by testing for unresolved forward references. The basic SMAL library sublanguage is described as follows:

```
<library file>::= { <object reference> }
                  <end of file>

<object reference>::= IF<symbol list> <line end>
                         USE<quoted string> <line end>
                      ENDIF<line end>

<symbol list>::= <symbol reference> {!<symbol reference> }

<symbol reference>::= FWD (R<identifier>)
```

As with the linker control sublanguage, the library sublanguage is intended to be understood by humans. As a result, SMAL requires no special object library management software; instead, object library files are maintained with a text editor. The following example illustrates the use of a SMAL object library file:

```
IF FWD(RSIN) ! FWD(RCOS) ! FWD(RTAN)
  USE "TRIGFUNC"
ENDIF
IF FWD(RFLOAT)
  USE "FLOATFUNC"
ENDIF
IF FWD(RFIX)
  USE "FIXFUNC"
ENDIF
```

In this example, it is assumed that the file system contains the object files TRIGFUNC, FLOATFUNC, and FIXFUNC. If, when the assembler reads this file, there is an unresolved reference to the external symbol FLOAT, the contents of the file FLOATFUNC will be read and linked with whatever was previously linked.

A significant problem with the SMAL object library scheme is that the library must be topologically sorted. For example, if object files A and B are in a library, and file A references a symbol defined in B, then file A must occur first in the library. For large libraries, finding an appropriate order is usually possible, but it can be difficult. If a SMAL library is randomly ordered, however, it may still be used by making multiple passes over it, for example, by including multiple references to the same library in the linker control file.

A problem for human users of the library sublanguage is its verbosity. This may be solved by using a macro to package each object reference on one line. A similar solution may be used to package common references in object files.

Extensions

SMAL demonstrates that a carefully designed assembly language and assembler can serve the common functions of an object code and a linker. Link editors, however, are frequently asked to perform other functions. Among these are overlay management, management of multiple relocation bases, and support of strong type checking for external names.

Overlay management appears to have originated with the IBM 7090 FORTRAN[2] and the Atlas HARTRAN[11] systems around 1961, and it has become a major feature of the linkers on most large systems.[12] F. P. Brooks has pointed out that overlays are inappropriate on multiprogrammed systems which use dynamic memory allocation,[13] but they still remain appropriate for some small systems. General automated overlay management is clearly not possible if an assembler is used as a linker, since assemblers cannot be expected to construct an object module dependency graph; however, partial automation is possible.

If each overlay is limited to having only one entry point, the calling program can be linked without knowledge of the internal structure of any of the overlays. Each overlay, on the other hand, must typically have access to data structures and procedures contained in the calling program. This requirement could be met by dumping the SMAL assembler's symbol table as an object file after linking a program, and then including this object file when linking each overlay called from that program. The SMAL linker control sublanguage can be extended to allow load-on-call overlay management by introducing a macro which constructs a load-on-call stub for each overlay.

Multiple relocation bases are supported to greatly varying extents in different assembly and object languages. As an extreme example, MACRO-11 permits any number of program sections, each of which may be separately relocated. The attributes of a program section include control over how program sections of the same name from different object files are to be treated; they may be overlaid, as with common blocks, or they may be concatenated in sequence, as with the default relocation base. The mechanisms included in SMAL are already sufficient to handle common blocks, but it is not clear how they can be used to generalize on the default relocation base without maintaining multiple relocation bases all the way through to the loader.

Strong type checking has come to be an important feature of linkers for some high level languages.[14, 15] There are a number of different semantic definitions for type equivalence, ranging from structural equivalence, where objects with the same structure are of the same type, to name equivalence, where each distinct type definition introduces a different type. Because all characters of an identifier are significant in SMAL, type information can be encoded in a suffix appended to each name. Structural equivalence can be enforced by suffixes that describe the structure of the associated types. Name equivalence may be enforced by suffixes which identify the files and location in those files where the referenced objects were declared. The current version of the SMAL assembler limits identifiers to 80 characters; this bounds the amount of type information that may be encoded this way.

CONCLUSION

The approach to object code representation and link editing used with SMAL is simple to implement when compared with the traditional approach, yet it is only marginally less expressive. All of the link editor features needed to support FORTRAN have been implemented in the SMAL system; and it appears that useful support for semi-automatic overlay linkage can be provided, along with limited type checking. The use of an assembler as a linker actually expands the possibilities for link-edit-time parameterization of a program far beyond what is commonly used, although the Atlas HARTRAN system and the LSS format supported similar facilities.[10, 11] Thus, it should be quite practical to eliminate the link editor as a special component of new programming systems.

This suggests the following software development sequence for new computer systems: first, a cross assembler-linker for the new machine should be written, using object and loader sublanguages such as are used by SMAL. This should suffice during the initial testing phase of the new system. The next stage should be to develop cross compilers using the object sublanguage before migration to the new system begins. At some point, the lack of efficiency of the object and loader sublanguages may become important; when this happens, the assembler can be modified to produce a compact binary output, and a linker can be constructed from this modified assembler by changing its input scanner so that it accepts the new binary format. If this is done carefully, it need not involve any change to the semantics of the object and loader languages since the syntactic changes need not introduce any new operations.

ACKNOWLEDGEMENTS

This work would not have been completed without Cecil Coker, who introduced me to the internals of the DDP 516 assembler and linking loader, Scott Lathrop, with whom I wrote my first assembler, and all of my students who worked with early versions of SMAL.

REFERENCES

1. D. W. Jones, *Machine Independent SMAL: A Symbolic Macro Assembly Language*, Report 82–03, Department of Computer Science, University of Iowa, Iowa City, Iowa, 1982.
2. W. P. Heising and R. A. Larner. 'A semi-automatic storage allocation system at loading time', *Comm. ACM*, **4** (10), 446–449 (1961).
3. L. Presser and J. R. White, 'Linkers and loaders', *Computing Surveys* **4** (3) 149–167 (1972).
4. P. Wegner, 'Communication between independently translated blocks', *Comm. ACM*, **5** (7), 376–381 (1962).
5. P. Wegner, 'Intermediate languages and programming systems', Chapter 5 of P. Wegner, (Ed.) *Introduction to System Programming*, Academic Press, London, 1964.
6. D. R. Cheriton, *et al.*, 'Thoth, a portable real-time operating system', *Comm. ACM*, **22** (2), 105–115 (1979).

7. G. H. Mealy, 'A generalized assembly system (excerpts)', Chapter 5B of S. Rosen (Ed.) *Programming Languages and Systems*, McGraw-Hill, New York, 1967.
8. C. W. Fraser and D. R. Hanson, 'A machine-independent linker', *Software—Practice and Experience*, **12** (4), 351–366 (1982).
9. Digital Equipment Corp., *MACRO-11 Reference Manual* (DEC-11-OIMRA-B-D) Maynard, Mass., 1976.
10. J. McCarthy, F. T. Corbato and M. M. Daggett, 'The linking segment subprogram language and linking loader', *Comm. ACM*, **6** (7), 391–395 (1963).
11. A. R. Curtis and I. C. Pyle, 'A proposed target language for compilers on Atlas', *The Computer Journal*, **5** (2), 100–106 (1962).
12. B. C. Lanzano, 'Loader standardization for overlay programs', *Comm. ACM*, **12** (10), 541–550 (1969).
13. F. P. Brooks, *The Mythical Man Month*, Addison Wesley, Reading Mass., 1975, Chapter 5.
14. R. G. Hamlet, 'High-level binding with low level linkers', *Comm. ACM*, **19** (11), 642–644 (1976).
15. R. B. Kieburtz, W. Barabash and C. R. Hill, 'A type checking program linkage system for Pascal', *3rd Int. Conf. on Software Engineering*, Atlanta, Georgia, May 1978, pp. 23–28.