FACTORS INFLUENCING THE DESIGN, IMPLEMENTATION,
PERFORMANCE AND USE OF LINKING LOADERS

by

Karl Reed

This project was submitted to the Department of Computer Science,
Monash University, for the practical requirements completing M.Sc.
Preliminary candidature. This work also has had support from
L.M. Ericsson Pty. Ltd., in the form of study leave.

DEDICATION

To give due weight to those who have stimulated and assisted the
author variously is difficult indeed.

Professor Chris Wallace, who supervised the work, discussed the
problems and tolerated the delays as well as showed a genuine interest in
the author and the project, has my heart-felt thanks.

Dr. Len Whitehouse who also tolerated delays but who provided the
job which made it possible for the author to complete the work at all,
is someone to whom I am extremely grateful.

Peter Herman who prodded and argued, and was responsible for the
author producing Section 10.3.2 has been a valuable friend and helper.

To these people I am in debt – particularly Chris Wallace and
Len Whitehouse.
To them, and of course, my wife, without whose willing sacrifice this work could not have been undertaken, I dedicate this work!
CONTENTS

1. SCOPE OF THIS PROJECT
2. INTRODUCTION
3. THE PLACE OF THE LINKING LOADER IN COMPUTER SOFTWARE
4. FUNCTIONAL REQUIREMENTS OF THE LINKING LOADER
5. THE IMPACT OF LINKING LOADER REQUIREMENTS UPON SUPPORTING SYSTEMS, AND VICE-VERSA.
6. SUBROUTINE AND EXTERNAL SYMBOL LINKAGE AND THEIR RELATIONSHIP WITH PROGRAMMING LANGUAGES
7. SUBROUTINE AND EXTERNAL SYMBOL LINKAGE AND THEIR RELATIONSHIP WITH TARGET INSTRUCTION SETS
8. SOME ELEMENTARY MATHEMATICAL EVALUATION OF THE ECONOMICS OF PROGRAM DEVELOPMENT
9. OVERLAY PROCESSING AND ITS RELATIONSHIP WITH TARGET INSTRUCTION SET.
10. AN EXAMINATION OF AUTOMATIC CALL LIBRARY ORGANIZATION AND CREATION ON SOME EXISTING SYSTEMS.
11. SOME EXPERIMENTS WITH LINKING LOADERS
12. FACTORS TO BE CONSIDERED IN THE DESIGN OF A GENERALIZED LINKING LOADER
13. THE CONCEPT OF A GENERALIZED LINKING LOADER
14. TECHNIQUES FOR SPEEDING UP LINKING LOADERS
15. CONCLUSION
APPENDICES

A-1 SPECIFICATION OF A LINKING LOADER FOR A NEW MACHINE

A-2 CODE AND RESULTS FROM HP2100 MODIFICATIONS

A-3 REFERENCES
1. SCOPE OF THIS PROJECT

The purpose of this project is to examine the Factors Influencing the Design, Implementation, Performance and Use of Linking Loaders with a view to consolidating the existing knowledge in this area as far as possible. It is not expected that this work will be definitive - that would be presumptuous in the extreme - but it is hoped that the results will at least make the reason for the existence of Linking Loaders clearer and provide data which will simplify the task of their designers.

The project will discuss the role of linking loaders as components of operating systems and program development facilities. Special attention will be paid to functions which should be found in a Linking Loader and the impact of these functions on the system supporting it. It is believed that the role and characteristics of Linking Loaders are best viewed in this way since the problem of relocation which is traditionally covered in the literature, in fact, reveals very little of the reason for the retention of the Linking Loader. The relationship of the loader to language processors which produce pre-compiled modules as loader input is discussed and their relationship to the target instruction sets and relocation will be treated in some detail at this point.

Some elementary mathematical evaluations of the economics of program development are presented, which, despite their simple nature, are original to the best of the author's knowledge.

A study of this type must consider both overlay processing and the use of automatic call libraries. The latter item is of sufficient importance to merit separate treatment, together with the examination of the techniques used by some existing systems to overcome this problem. This problem has
received only cursory attention in the literature.

Experimental work involved adding a directory to the HP2100 system library and modifying the loader to make use of it when searching the automatic call library.

Factors influencing the design of a generalized linking loader are discussed and the basic outline of a design presented.

Finally, techniques for speeding up Linking Loaders are discussed.

The specification for a Linking Loader and some of its design are included in the Appendices.

Reference is made to existing literature and manufacturer's documentation where appropriate.

It should be noted that the relocation problem will be regarded as secondary during this study.
2. **INTRODUCTION**

2.1 **Preamble**

The term "linking loader" is used loosely to describe what are in fact two related types of systems performing essentially the same functions. We will cheat slightly by not distinguishing between them because the majority of the problems are common to both.

2.1.1 **The Relocating, Linking Loader** is a package designed to take the output from language translators and bring it into memory, linking separately translated routines together and locating any "library" routines. The resultant memory load is relocated (if necessary) and is ready for execution.

2.1.2 **The Linkage Editor** does essentially the same thing, except that the resultant load module is not placed in memory. It is placed in a library.

We will point out in Section 7 that the ease and efficiency with which the latter can be achieved (or rather, loaded later) will depend somewhat upon the computer architecture. This point is covered in [GLI] in some detail.

2.2 **Some Definitions**

We will adopt some definitions which will avoid confusion in what follows:

2.2.1 **Object Module**

The output produced by translating a single procedure, subroutine or program which consists of only one such element. The object module contains all the information needed to resolve references to other separately translated object modules and is generally executable.

2.2.2 **Load Module**

A load module is capable of being executed. It contains no unintentional unresolved references to object modules not included in the load module. It may
be core resident or in a library.

2.2.3 **External Reference**

A symbolic reference to some entity (data, program label etc.) which was not defined in the source which contains the reference. Note that external references will have been declared.

2.2.4 **External Symbol**

A symbol representing some entity (data, program label etc.) which has been declared in such a way that the translator will place data in the object deck which will enable external references to this symbol to be resolved.

These definitions are similar to those given in IBM Linkage Editor manual [MD4], and those used in [GL1] p.151.

2.3 **Why the Linking Loader?**

The reason for the existence of a linking loader is both historical and economic. Program language translation, even on the most modern systems, is a slow process despite all the current developments in that area. In the past it was even slower. It was considered necessary, therefore, to translate source into an intermediate form (the object module) which was complete except for the presence of absolute addresses and unresolved external references. Commonly used object modules would be held in a library and, when execution was required, the loader would load the required object modules and search the library, resolving any remaining unresolved external references.

Most early loaders accepted the input from nominated files without enabling object modules to be selected. However, this still made it possible for commonly used programs to be held in object module form saving compilation time. An example of this is the CDC 3200 MSOS Loader.
It will be seen in Section 8 that there are economic gains to be made from compiling modular rather than monolithic programs, and the linking loader makes these gains realizable. Strangely enough, this point seems to have been overlooked in the literature. The approach taken in the literature, by those who actually discuss the raison d'être of linking loaders is to examine in detail the problem of address relocation. Reference [Gll0] for example, which describes the loader for the IBSYS/IBJOB project lists three prime reasons for having a linking loader:

- determine the operational locations of a program's parts,
- translating all non-absolute references to absolute form,
- positioning program and data in core.

Another reference was found to the desirability of being able to link together programs produced by different compilers. This particular view was regarded by the author with some scepticism, but subsequent discussions with computer staff of sufficient seniority who were active in the field when loaders were first used in their modern form, suggested that this factor may have been of great importance in supporting the marketing of the first FORTRAN compilers [Gll6]. The linking loader probably dates from that time.

It should be noted that simple relocatable assemblers existed on many early systems (e.g. Elliott 803). These accepted machine instructions in numeric form with relocatable address references.

It should also be noted that it is possible to write assemblers which run at extremely high speeds, high enough to make it necessary to consider the possibility of avoiding a linking loader if the only translator was an assembler.
2.4 A Popular Software Item

The linking loader is provided on most manufacturers systems. In fact, the B5500 system is the only one of any size that has been offered without this support package in the last 15 years as far as I am aware. Most mini-computer systems now include a linking loader in their software support systems. There is also very little discussion of this essential item in the literature. This study seems to be justified from this point of view alone.
3. THE PLACE OF THE LINKING LOADER IN COMPUTER SOFTWARE

The linking loader's place in computer software has been mentioned in Section 2 of this report. However, in its modern form, it plays an essential role in software production due to its library manipulation capability as can be seen from [MD2][MD4][MD7] and [MD9].

Coupled with suitable file structures such as the IBM OS360 partitioned dataset (PDS) [MD10], the user can readily control the origin and hence status of the modules which are incorporated in a system. However, we seem to be stepping ahead somewhat.

The existence of a linking loader turns compilation into what is essentially a two or three step operation as follows:

STEP1 Compile source and produce object modules
STEP2 Link object modules and resolve references to library
STEP3 (optional) Load and execute.

We can add that unless a compiler is producing code directly in core for immediate execution, it will not be possible to hold compiler output in absolute form on most modern multi-user systems because the actual program loading point will vary from one run to the next.

It would seem, therefore, that once a decision has been made to hold pre-compiled code in some form, that a linking loader follows. The form of the pre-compiled code can be quite elaborate, but this will not be discussed in detail.
4. **FUNCTIONAL REQUIREMENTS OF THE LINKING LOADER**

We have established so far a picture of the linking loader and its functions. It is necessary now to attempt to define these in the light of experience with existing systems, the literature and what one might wish to see.

4.1 **List of Requirements**

4.1.1 **Object Module Selection**

Select object modules from some library structure as specified by control cards.

4.1.2 **Mass Loading of Object Modules**

Include all object modules contained in some specific file without discrimination.

4.1.3 **Call-Library Resolution**

Resolve un-resolved references against specified or default libraries permitting user control of this function.

4.1.4 **Program Linkage**

Link the various external references to the appropriate external symbol with due regard to data type etc.

4.1.5 **Perform Any Appropriate Relocation**

Perform relocation as is appropriate. If loading to core, produce absolute code. Otherwise whatever action is needed.

4.1.6 **Construct An Overlay Structure**

Construct, with the aid of user specifications, an overlay structure of suitable flexibility which permits the program to run with only those segments required in core at any one time. The user should not be forced to alter his source programs to achieve this. Refer Langano [GLl1].
4.1.7 Permit Saving and Re-Use of Symbol Dictionary

It should, where load module size is sufficiently large, be possible to save the symbol dictionary created by the loader so that additions can be made to an existing load module in the form of a load module loaded after it. This can be valuable where the loader is part of a system producing load modules for a dedicated computer with slow speed terminals.

4.1.8 Production of a Memory Map

A memory map should be produced if required by the user, and if appropriate to the system.

4.1.9 Allow for Removing Individual Programs from an Existing Load Module

This function is useful when extremely large modules are being maintained since the time taken to remove an individual program can be a lot less than that required to re-link the whole module from scratch. However, it also implies the existence of data which will permit the resolved references to the "edit" program to be located.

4.1.10 Type Matching At Link Time

Where appropriate, the types, or attributes of external references should be checked against those of the external symbol with the same name.

We will consider these functions in a little detail, without examining those aspects which properly belong in later sections, but with a view to creating a better picture of the whole problem.
4.2 A Model of Program Development

The functional requirements listed are those needed to enable a number of people to work simultaneously on the development of a large suite of programs. In this situation, the programmer will experience the need to access different types of program libraries at different phases of the project. In particular, it is valuable to be able to segregate the code used in one level of release from another, while permitting individual modules to be selected from earlier or different releases.

A programmer may need to create a library of dummy routines so that he can test his own code while waiting for final versions to be completed. He will, therefore, need a fair amount of freedom in selecting load modules. Of course, this must also be convenient.

It will also be necessary to control the call libraries in which unresolved references are to be found. Unfortunately, this is not just a refinement, as will be seen.

The question of convenience is paramount.

The control cards which must be present to the linking loader will constitute a description of a load module. Altering that description should be as simple as possible since it will effect the ease with which a large system can be developed, and this will ultimately affect the system reliability.

4.3 Examination of the Requirements
4.3.1 Object Module Selection

The user must be able to select specified object modules for loading. Ideally, one wants to be able to specify the name of a module which is to be loaded irrespective of both its subprogram name and of its entry points.

We would also not be disturbed if one name was used to load more than one object module, although, if the name used was a subfile name, we would not want to be in the situation of always having to load the contents of the subfile.
As will be seen in Section 5 where this matter is examined, the IBM Partitioned Data Set Structure allows complete freedom in this area.

Figure 4.1 and 4.2 show two possible arrangements. Note that attempting to load SIN in Figure 4.2 causes COS and TAN to be loaded, and neither COS nor TAN could be referred to by name.

Incidentally we require that the loader include all modules named, even if a module which has external symbols the same as those in a loaded module is specified.

This will permit individual external references to be directed elsewhere without excluding the remainder of the module. (I am indebted to J. Marquett for pointing this out.) Of course, a warning should be produced.

A further feature could permit programs to be selected by other parameters such as date and time of creation should these be available.

4.3.2 Mass Loading of Object Modules

This is covered in 4.3.1.

However, a likely case leading to the mass loading of files is that which occurs during program development when the source for several temporary "drivers" may be compiled in one invocation of a compiler, resulting, on most systems, in the object modules being stored in one file. Obviously, all of these should be loaded.

4.3.3 Call Library Resolution

At some point during the loading process we wish to have any unresolved references matched against some library in an effort to find the routines containing the required external symbols. Traditionally, this is the last thing which is done, after all named object modules have been loaded. There is no reason why this should be so, and it is possible to propose a design in which external reference resolution can be invoked explicitly as required. (Ref [MD11])
Fig 4.1 Sub-files used to permit access to individual routines.

Fig 4.2 Sub-files used to force access to several routines in a sub-file.
There are two problems associated with this process, both of which are not strictly associated with the process of linking specified modules (or the contents of specified sub-files).

4.3.3.1 The External Symbol Problem

Since we are not specifying a module explicitly, it is possible that it may be referred to by any one of a number of its external symbols. This means that there must be some way of ensuring that the fact that there exists a module with a particular external symbol can be discovered by the loader. We will deal with the technicalities of this point in Section 5.

4.3.3.2 The Generation of Additional External References in the Process of Call Library Resolution

There is no reason why a module loaded from a call library should not reference other modules in that or some other library. Such an approach can lead to what this writer regards as the most desirable and yet rarely achieved approach to specifying a loading operation, that is, specifying the name of just one program whose external references result in all other routines being loaded. See Figure 4.3.

It will be pointed out in Section 5 that this requirement presents performance problems and/or introduces other complications in the maintenance of call libraries.

4.3.4 Program Linkage

This requires little elaboration, except that, whatever steps are necessary to ensure that the referencing program will be able to use the referenced symbol as it should must be taken. One can envisage the need for the compiler to provide additional data to the loader in some cases.
A small but important point relates to the ability to use constants defined in another module. In fact, we want to be able to reference externally any class of object which can be referenced locally!

4.3.5 Perform Any Appropriate Relocation

Relocation can be performed essentially in two ways:

4.3.5.1 Base Register Setting

The code necessary to set base registers may be added if the translators do not provide it.

4.3.5.2 Address Relocation

It may be necessary to physically convert relocatable addresses to absolute addresses, or address relocated with respect to a complete load module.

This process is described as "address binding" by several writers ([GL1], [GL15]). Clearly, the problem can assume many different forms depending upon the computer architecture involved.

4.3.6 Construct An Overlay Structure

Overlay structures permit the execution of programs which will not otherwise fit in main storage. In theory, it will not be necessary to manually arrange overlays on computers with some form of virtual memory. By virtual memory we mean in part, combined hardware/software mechanism for automatically ensuring that the required program or data is in main storage. Such schemes allow the logically addressed space to exceed physically available space. Physically available space can be considerably less than the space available in main memory after allowing for the operating system.

In practice, it is possible that on a machine with fixed size paging, that the virtual memory scheme may try to keep as many users as possible in main memory at one time. The user would still be required to manually overlay his logical address space under such a regime (for example, the DEC10 paging processors).
FIGURE 4.3  Intra-module and Intra-library References

Implicit loading of a library object module due to an intra-library reference.
An overlay structure permits the memory no longer needed to be overlaid by a copy of some required program or data where this overlaying process is controlled only by "calls" from one subroutine to another. The overlaid portion is saved on backing store. The process is, as a result, quite different to what could be called "genuine" virtual memory systems where logical addresses are mapped (somehow) on to an available physical address space, permitting non-resident data to be automatically loaded. In particular, data local to the overlaid program is overwritten and is not available to the overlaying code. This means that an overlaying program cannot communicate with its predecessor through data common only to both.

Many overlay systems do not even permit direct calls from overlaid to overlayer for this reason [GL11].

Lanzano [GL11] presents an excellent discussion of overlay processors, touching upon many desired features. What follows is largely taken from his comments.

4.3.6.1 Overlay Invocation

An overlay process should occur whenever a call is made to an external symbol in the section of code to be overlaid (we will call this a segment). In other words we restrict this process of communication to the need to access some non-resident program rather than its data!

This restriction exists in all overlay systems of which the writer is aware, and constitutes a major difference between overlay systems and virtual memory systems.

Our overlay structure will be described by control statements loaded independently of the object modules and which will not require user modification of source programs as would be necessary in several existing systems. This last point is extremely important.
Further, the overlay structure must be such that any sequence of calls possible in a non-overlaid version of the program must be allowed. It should be noted that any parameter transfer mechanisms must be preserved.

4.3.6.2 Data Communication

Data communication will be via data which will be resident when any of the subroutines using it are present. We require that the capability specified in 4.3.6.1 should not result in all global data being forced into the root segment when an overlaid segment calls to the root segment which in turn calls a module in a different segment which overlays the originating segment. See Figure 4.4 from which it is apparent that if segment 1 occupies the same physical space as segment 2, then the data in COMMON/BB/ which could be used for communication between OUTER11 and OUTER21 will have been destroyed by the call to OUTER21.

Our requirements are that this destruction of data should not occur, and furthermore, that the user should be able to choose whether he wishes to have BB moved inwards or saved between overlays on backing storage. In other words, the user should be able to have the loader detect where a named common is used in mutually overlaying segments and leave it intact (somehow) so that it is available for communications. It should be saved along with the rest of the overlaid code if not needed.

It should be noted that the existence of a cyclic path of the type shown will force BB to the innermost segment in the cycle in most existing systems.

At this point we can suggest that the programmer should be able to call OUTER21 from OUTER12 or OUTER12.

4.3.6.3 Alternate Segment Selection

Lanzano ([GL11] pp.549-550) suggests that (what he calls) imposed tree structures should be eliminated. He points out that "General purpose simulations ... incorporate many sets of functions within which overlays are appropriate for
PROGRAM ROOT
COMMON /AB/ .....  
CALL OUTER11  
END  
SUBROUTINE ROOT1  
COMMON /AB/ .....  
CALL OUTER12  
END  
SUBROUTINE OUTER11  
COMMON /BB/ .....  
CALL ROOT1  
END  
SUBROUTINE OUTER12  
END  
SUBROUTINE OUTER21  
COMMON /BB/ .....  
END  

ROOT SEGMENT
{  
ROOT
ROOT1
{  
SEGMENT1
OUTER11
OUTER12
SEGMENT2
OUTER21

FIGURE 4.4.2 Shows OUTER21 physically overlaying OUTER11, OUTER12.

FIGURE 4.4.1 A Normal Calling Sequence

SYS  ROOT  OUTER11  ROOT1  OUTER21  R  ROOT1
OUTER12  R  ROOT1  R  OUTER11  R  ROOT  R  SYS

FIGURE 4.4.2 Call Sequence
the functions which are mutually exclusive. These function-sets are not in any way dependent upon each other; their utilization depends on the state of data which changes" (during the simulation).

He proposes a "floating" segment concept (my title - his is alternate) which can be used at several points in overlay structure, starting at different "addresses" (i.e. where the overlays have different lengths). Simplifying his involved example from Figure 7 ([GL11] p.550) we have

A
  /    \
 B  C
  /   /  /
 D11 D12 D13 E E E
  /   /   /
 D  E  F11 F12 F12
  /   /
 D11 F11

FIGURE 4.5 Alternate or Floating Segment Selection

The broken line indicates that there is more than one root requiring this sub-overlay.

For example, possible in-core arrangements are:

A, B, any one D11, any one F11

A, C, D, any one E11, any one F11
Clearly, the $F_{li}$ must be held in a relocatable form. It is not clear from Lazano's remarks whether he would allow an arrangement such as $A, B, F_{li}$ but this follows from his proposal.

4.3.6.4 Specification of Overlay Structure

The modules constituting an overlay segment should be specified as already indicated by direct inclusion of the object decks, but primarily by specifying the names of modules in libraries. It should be possible to control library call resolution within an overlay segment or group of segments readily.

Most of the problems which should be handled by the overlay processor are described.

However, the preceding discussion is somewhat idealistic. Quite effective overlay processors which do not provide all of the facilities listed exist and are in common use. For example, the PDP10 linking loader, LINK-10, [MD7] is an elaborate loader which meets most of our requirements except that it does not allow parallel segments to call each other.

The IBM linkage editor ([MD4] pp.68) has a curious restriction. Calls between overlaying segments (called exclusive calls) are allowed only if there is a call to the target routine in an inner root. Further, "exclusive" calls which return directly to the caller (without going into a root) can cause abnormal terminations of COBOL, PL/1 and FORTRAN programs.

Lanzano [GL11] (to fall back on an old standby) discusses these points and it is apparent that many of these features are not absolutely necessary, only desirable. Pankhurst [GL17] describes an extremely limited but yet clearly very useful system which allows for only two overlay levels.

My own belief however, is that the desirable should be clearly stated so that the consequences of eliminating capabilities can be weighed against implementation savings.
will provide information on the file from which a module was taken and includes the time and date of creation - but not that of the source module.

The time and date of creation of the load module is also written on the map.

A cross reference listing of global symbols (including common where applicable) and references to global symbols, is also desirable.

4.3.9 Editing a Load Module

The comments made in 4.1.9 are sufficient.

4.3.10 Type Matching at Link Time

The symbols which are external references will have types or attributes in some cases. This will be true in higher level languages where the symbols will be typed. Strangely enough, many linking loaders [MD8, MD2] do not check this. The IBM Linkage Editor warns when named common's have used subroutine names, however. [MD4].

One would expect, particularly with block oriented languages, that type-checking of external symbols would be performed.

There is also no reason why parameter types cannot be checked at link time. This would require both actual and formal parameter string descriptions, and would remove some of the "unstructured" power provided by systems such as [MD8] where any data type is accessible in FORTRAN.

Incidentally, we can propose external reference and symbol "typing", and parameter checking even in assembler if so desired.

4.4 Some Additional Functions for Linking Loaders

We have in fact overlooked two important functions which have occasionally been delegated to linking loaders in the past.
However, the use of modified base addresses has the unfortunate property that it dictates the actual word boundaries used to start both arrays.

Consider:

We have two arrays which are to start at \( \text{START}_1 \) and \( \text{START}_2 \) and each has a size given by \( 2^1 \) and \( 2^2 \) respectively. Therefore, if the elements are indexed by \( I \) we have:

\[
A_1 = \text{START}_1 + I \ast 2^1
\]

\[
A_2 = \text{START}_2 + I \ast 2^2
\]

but we require that \( A_1 \) be used as an index i.e. we want \( \text{START}_3 \) and shift \( Z_3 \) such that:

\[
A_2 = \text{START}_3 + A_1 \ast 2^3
\]

i.e. \( \text{START}_2 + I \ast 2^2 = \text{START}_1 \ast 2^3 + I \ast 2^{1+2} + \text{START}_3 \)

equating variable and constant parts we have:

\[
I \ast 2^2 = I \ast 2^{1+2}
\]

\[
Z_3 = Z_2 - Z_1
\]

and \( \text{START}_3 = \text{START}_2 - \text{START}_1 \ast 2^2 - Z_1 \)

from which we see that \( \text{START}_1 \ast 2^2 - Z_1 \) must be a multiple of 16, hence we create a constraint on possible values of \( \text{START}_1 \) if \( Z_1 > Z_2 \).

Not surprisingly, there were insufficient \( 2^{4+n} \) word boundaries in 64k memory modules to permit large systems to be configured without an "out".

A more realistic requirement of a linking loader for modern systems holding program in ROM would be that it construct module interworking tables so that only these tables need be altered when new program (and hence new modules) are loaded.
4.4.2 Conditional Assembly

The concept of conditional assembly is well known to assembly programmers and those used to language independent macro processors. In general, the symbols used to control the assembly must be local or defined at compilation time. The word "defined" can be used loosely and the definitions may be physically separate from the program itself.

The main reason for this approach is that the use of undefined (or global) symbols will cause the macro-processor to produce all possible expansions of the source. The loader would then use the defined symbols which will need to be available to select the appropriate code. We assume that the compiler will produce "normal" code for all of the generated source together with conditional indicators for the loader.

There are no fundamental problems associated with this approach except that the length of a program is no longer known until it has been scanned at least once. (As we shall see later, this need not present a problem). Of course, this is only true if the global symbols are defined when the program is loaded.

This will only be true if there is a suitable restriction on the use of these global symbols which forces them to be backwards references. There is no real problem if the controlling variables are not run time addresses since the values can be made available prior to loading. However, if variables derived from run time addresses are allowed, and forwards references are allowed, we can create a situation where we will have difficulty selecting the right section of the code.

The linker for the DEC System 10 [MD7] is capable of handling this type of input.

The equivalent parts of L.M. Ericsson's ASA and APS systems also accept as input compiler output containing undecided conditional assembly statements produced in several ways.

As an aside, we may consider for a moment the use of compile time constants in higher level languages.
Quite clearly, the compiler can decide whether or not code should actually be emitted if the controlling expression involves only defined compile time constants. Languages such as ALGOL68, the Burroughs ALGOLs and some others provide for the symbolic naming of constants.

However, if a conditional statement is controlled by an expression which contains some undefined compile time constants, then it will not be able to decide which code should be omitted and must emit it all. This assumes that a suitable mechanism for declaring symbolic constants external is available.
5. **THE IMPACT OF LINKING LOADER REQUIREMENTS UPON SUPPORTING SYSTEMS AND VICE-VERSA**

5.1 **Introduction**

Our previous discussion in fact only raises one factor which would influence the design of a system used to support the Linking Loader. That is the directory structure. We will consider this here in some detail and return to it in Section 9 when automatic call libraries are considered.

5.2 **Directory Structures**

The primary requirement here is that the directory structure should provide a mechanism which enables a module to be located by or through an external symbol.

In fact, practical programs will contain many external symbols, any of which may be referenced. Our directory structure must allow for an "alias" mechanism, since the loader and compiler will not necessarily know which object module is actually required.

Before continuing we can just note that the problem also occurs for source program and macro libraries except in these cases we are accustomed to having only one name for such an entity. If the linking concept described in 4.3.3.2 is used, then the project organisation is simplified since the programmer only needs to control the library against which an external reference is resolved.

We can distinguish between two forms of directory; the "qualified name" mechanism and the sub-file mechanism.

But before doing this, we can point out that access to object modules is not the only reason for providing a means of grouping files but distinguishing between them.

5.2.1 **Qualified Names**

What we are really saying (in admittedly an unnecessarily round-about way) is that we require a mechanism for specifying a group of files, a process (c.f. Burroughs 6700 Work Flow Language [MD13] and IBM JCL [MD14]) which involves
Any library operation can be handled this way provided that the complete operating environment is sufficiently flexible.

Problem programs which are allowed to generate a file name (not possible under OS/360) can read and write any individual program which is part of a group or library.

It should be pointed out that most programming languages use external symbols which are simple identifiers, hence this approach is quite satisfactory. Macro libraries and any kind of source library can obviously be handled.

Problems arise where there are many paths to an item in a library - our multiple external symbol problem.

We note that the use of a simple qualified name fails to meet our requirements since the actual final qualifier used to specify the file may not be the symbol appearing in the external reference - the most likely final qualifier is the program or subroutine name, which will not be known.

One solution is to permit aliases for files. We allow more than one qualified name to specify the same file. The only problem with this is we now create a situation where no program can have the same name as an external symbol - a good idea anyway.

A more serious objection is that this may cause excessive overheads in opening and closing files (this objection applies to the use of qualified names anyway!) and may take up unnecessary directory space.

It also means that the directory must contain a pointer to the true file. The compilers or library maintenance systems will have no difficulty in creating the necessary aliases - if they remember! The B6700 MCP, BINDER and its compilers conspire to ensure that a program cannot be located via any of its external symbols, only by its name!

At least, however, the B6700 (and many other) systems allow the programmer to operate on files whose names are data.
The OS/360 Linkage Editor, however, requires libraries (and all files) to be specified on data definition cards (dd cards) [MD4] which are part of the job step for a link-edit process.

The linkage editor (and any other problem program) can refer to a file only via its dd names.*

This means that library control statements refer to a member name in the file specified on a named data definition card. A user must therefore change his JCL and his control cards if he wishes to change the files to be used, and add control information in two places if access to additional libraries are required.

In my view, this is a distinct disadvantage. The programmer has to learn two different languages in detail when only one is needed.

* The supervisory and Data Management Macros [MD10] provide access to the partitioned data set directories, and PL/I at least supports this also.
6. SUBROUTINE AND EXTERNAL SYMBOL LINKAGE AND THEIR RELATIONSHIP WITH PROGRAMMING LANGUAGES

6.1 Introduction

So far, we have sketched some of the problems created by the linking loader and discussed their solution of what one can call a specification level. The factors causing the problems do so in two ways. Clearly, programs which are communicating in a static fashion have the external symbol/external reference properties.

But what type of symbols? How do they arise and what should be considered? If this were a book, we would probably start at this point. But ’tis not a book!

Interestingly enough, this section will lead us into some questions of language design and implementation. It will not always be possible to propose "solutions". Rather, difficulties will be presented and alternatives suggested when necessary.

6.2 Some Basic Distinctions

We will distinguish conceptually between external references to data and external references to programs. It must be accepted that the distinction may blur - consider an array of references to procedures - but we can make the following quite useful rule.

R6.1 References to external data do not result in a transfer of control to some other program or procedure which is part of the user's problem -solution.

In other words, there may in fact be some system intrinsic executed to make the data available, but the user will not be able to detect this change of control. This will permit "fudges" when necessary.

We will also introduce the concept of mode or type.
R6.2 Symbolic references in a program may or may not have some type. That is, they may or may not have associated with them some property other than their value which enables a translator to determine precisely what code should be emitted for their use.

R6.2 probably differs from generally accepted definitions. Types exist within a semantic and syntactic framework, but on the whole their essential function is as described.

The point of all this is that a loader can be required to check the types of symbols used in external references against those which they match. In the same context, the types of actual parameter strings can be checked with formal parameter strings.

In extreme situations, a loader may have to generate special code to permit the cross reference to occur. That is, some type conversion may be needed.

6.3 Assembly Languages

By far the simplest example is to be found in the Assembly Languages. These do not usually permit any typing of symbolic references. There is generally no restriction upon the use of symbols. A symbolic label may be used in a jump instruction or a data reference. No compile or run-time diagnostics are produced.

To the best of my knowledge, only the L.M. Ericsson APS system provides mechanisms at the assembler level for declaring different types of symbols. All that is required now is a mechanism for declaring external references and global symbols. This is usually done via pseudo instructions which declare what are generally called external and entry symbols.

One minor problem relates to address arithmetic - should address arithmetic expressions include external symbols and if so under what restrictions?
6.3.1 Symbolic Expressions in Assembler Address Fields

Generally, any evaluable expression should be allowed. Strictly speaking, the only expressions which are not evaluable, eventually, are those involving undefined symbols. This implies that any expression involving external references is legal. Barron ([GL4], pp.32, 35, 43, 53) says that expressions involving products and sums of symbols must be classified according to their use of relocatable symbols. However, I would argue that in fact, there is no such thing as a relocatable value, there are only un-relocatable symbols. Upon relocation, these symbols acquire an absolute value. This value is unique for any situation in which any storage allocation specifications determined by symbols use only absolute symbols.

This assumes, of course, that we are prepared to make two passes of the "load", something which must be done in most cases anyway. It also assumes that we will wait until the second pass before evaluating expressions. Having said all this, we can allow any kind of expression which does not make storage allocation dependent upon relocatable symbols, and of course, external symbols can be used.

The only other restriction (obviously) is that no symbol can have a value which depends upon itself. Barron ([GL4] p.53) excludes some cases. Obvious examples of relocatable symbols are program and data names in another program. Absolute symbols can be used for data generation.

It will certainly simplify matters if we introduce the rule that any expressions using external symbols must involve a backwards reference at load-time.

6.3.2 Operand Sizes

Clearly, the result of any external reference (or expression involving it) will have a range. The range will be fixed by the number of bits available for the operand. The loader will need truncation rules for handling those situations where the result exceeds the range, and should produce a warning message. This matter relates strongly to instruction sets and we will deal with it in that section.
6.4 FORTRAN

As remarked earlier, the language FORTRAN is credited by some with the blame (or credit) for stimulating the development of linking loaders. FORTRAN has separately compiled subprograms (of the FUNCTION or SUBROUTINE type) and mechanisms for communicating between them. Generally, symbols are declared external by their use, with one exception; and program entities and labelled common are the only external items.

For example: \[ A(I) = B(I) \]
results in a FUNCTION subprogram reference to B if B is not declared as an array.
Similarly, \[ \text{CALL BA(I)} \]
results in a subroutine subprogram reference to BA.

The common data areas also constitute a form of external symbol. Each common area (any labelled commons and unlabelled common) constitutes a unique area which can be accessed from any subprogram. These data areas are re-mapped according to the local data items declared in them in any subprogram so that the data names are in no way external references! This is despite the fact that the data is external.

Since the difference between a data reference and a subprogram reference can be decided by usage, a subprogram name can be included in a formal parameter list. However, in an actual parameter list, a declaration is required to ensure that a name used only as a parameter will be recognized as subprogram reference.

Parameter transfer is a run-time, not a linkage time problem except that the type of a parameter in both actual and formal strings is known at compile-time and since subprogram calling mechanisms are standard between modules produced by one compiler, a loader could be expected to perform type-checking at loading time. We have already made this remark.

Only the various commons can present difficulties since storage must be allocated for each. It is generally assumed that the first appearance of a common determines its length and subsequent appearances cannot exceed that length.
FORTRAN DATA statements initializing storage in a common should also be checked. The first encountered initialization should be accepted and subsequent attempts should produce an error.

6.5 Block Structured Languages

6.5.1 The Problem of Implicitly Global Variables

Traditional block structured languages based upon ALGOL 60 have what we will call the property of implicitly global references. The normal view of a local variable is that it is either the definition of some new variable, or the re-definition of an existing variable.

It could be said that in this case we view a block (or procedure) from the outside in.

```
looking in
   begin
      real a, b, c, d;
      comment this block-head contains declarations which are global to any enclosed blocks unless new declarations are made;
   begin
      integer x, y, d;
      ;
      ;
      a: = y;
      ;
      comment a, declared outside the block, is assigned a value y from a local variable;
   end;
looking out
```

**FIGURE 6.5.1** (Figure 6.5.1 is a typical situation)
A programmer, however, opening a program listing in such a fashion that he could see only the enclosed block in Figure 6.5.1 would not be able to tell at that point whether or not "a" was undefined or global.

There is no data in the block-head which will tell him what the global variables are, hence, anything which is not local is "implicitly" global; its declaration existing at some outer level or not at all.

6.5.2 Separately Compiled Blocks and Procedures

The important thing about this is its effect upon separately compiled pieces of code. We now have, potentially at least, the possibility of external data items.

Strictly speaking, there is no such thing as an undefined variable or procedure in a separately compiled block or procedure since that item may be used in a program which contains the appropriate declarations.

A similar problem exists for procedures which may be used in different contexts. Some mechanism for "fixing" the scope in which a procedure is considered to be declared is required. Further, separately compiled "segments" need to "know" what the declarations for implicit globals are meant to be. It should be noted that in the case of FORTRAN, all type information is local to the separate entity even if, as in the case of common, it refers to global data.

6.5.3 A Sensible Proposal

The writer's preferred solution involves preventing this "implicit global" feature from having any effect other than that required.

We would introduce what would be an external declaration which would specify the name (and type) of the procedure and its environment, that is, the names of all items within the scope of the external declaration which are actually global to the procedure.
Similarly, separately compiled procedures would include an environment statement which would contain complete declarations of all actual globals.

Example:

begin
  real a, b, c, d;
  
  begin
    integer my, mx, mz;
    boolean plus;
    external procedure crunch
      environment a, my, mz;
      
      crunch(plus, mx);
    end;

    end;

procedure crunch (sign, prod);
  Boolean sign;
  environment begin
    real a;
    
    integer my, mz;
    
    end
    
    begin
      prod: = my x mz;
    end;

    end;
This proposal has some obvious conflicts with the spirit of ALGOL60, but at least enables a loader to ensure that only those globals which are required by the separate code are bound in. Further, we allow a procedure which is compiled separately to be used in the code with correct scope since a new environment specification is required at each point. We would allow several external procedures to be included in one source document or file each with a separate environment declaration. All this means that any undeclared variable is undeclared - and not implicitly global.

6.5.4 The Implications of the Alternative

The possibility of allowing the implicitly global property to be used in separately compiled code creates many problems for the loader. Consider the following separate procedure:

```plaintext
procedure separate (a, array, result);
    integer a;
    integer array array [1:6, 2:10];
    real result;
    begin
        aa: = a + array [i,j];
        result: = aa + bspec;
    end;
```

We can know that:

1) aa is a variable of some type, either real or integer,

2) i,j are at most procedure calls returning a value,

3) similarly for bspec.

It is not to say these problems cannot be solved. However, they require either partial, rather than separate compilation or an approach which requires a subroutine call for each reference to an implicit global. The loader would then generate the code necessary to obtain whatever type-conversion etc. is needed.
For example, for the case involving aa, we can assume the compiler could produce
the following:

CALL STORE REAL* (aa, aa's type)

and aa, and its type can be provided at load time.

Similarly,

result: = aa + bspec;

involves an even more open form:

CALL ADD** (aa, aa's type, bspec, bspec's type)

The main objection to this is that a significant run-time execution overhead
has been added in addition to increasing the general meaninglessness of the
source document.

6.5.5 Two Solutions to the Problem

Both the DEC System 10 and Burroughs B6700 implementations of ALGOL60
have facilities permitting separate compilation.

6.5.5.1 External Procedures In DEC System 10

This implementation (Ref. [MD20] p. 11-10) provides for a procedure to be
declared external. All that is required is that the procedure declaration be
replaced with an external declaration in the appropriate block-head.

"Such an EXTERNAL declaration can be made in any block within the program
and has the same scope as if the procedure appeared at that point."

It should be noted that the procedure name and type only are declared.

The loader documentation (Ref MD7 p.A-3, A-4) describes a mechanism which
allows a symbol to be regarded as either a request for a global or local,
depending upon the existence or absence of a definition for that symbol.

* We assume the result is implied as top of stack, register etc.

** In practice, generalized binary operator routine could be used, and
descriptor addresses.
We presume therefore that a separately combined procedure is enclosed in a null block which contains definitions of all required globals. The compiler then generates a relocatable module of the required form. Also, that all symbols in the scope of the EXTERNAL statement are automatically global to it for loading purposes.

Note that the quoted statement from MD20 specifies that the same procedure may be used in different scopes with different globals - a highly desirable feature. (Ref[ MD20], p.11-10)

6.5.5.2 External Procedures in Burroughs B6700 ALGOL

The B6700/B7700 software support system contains a BINDER which offers quite extensive facilities (Ref.[ MD30], [MD2]), and this has evolved somewhat over the last few years (compare[ MD2] and [MD2G]). Our particular concern is with External Procedures in ALGOL, but it should be noted that the problems raised here reappear in COBOL and in some inter-language linking cases. The latest manual ([MD30], pp.6-1 to 6-4) describes the rules for binding and using external procedures which we discuss here.

A procedure can be declared to be external and the declaration must include parameters. Such a declaration can occur in any block-head, and has the scope of its "external" declaration. It is noted ([MD30] p.6-1) that an external procedure can also access items declared after the procedure declaration in that block-head. This would not be the case if the procedure were textually in line.

The separately compiled procedure indicates its global requirements by listing them (as proper declarations) in square brackets at the start of the source. Note that several procedures may be compiled separately from one source file, but there can be only one "global set". These globals are satisfied with due regard to scope when the procedure is "bound" in. So far so good!

However, it should be noted that the lexical level at which the separate procedure is to be used must be declared in the separate procedure. Hence, a
procedure which is to be used in two different scopes at two different lexical
techniques must be compiled twice. Fortunately, the BINDER provides mechanisms for
controlling the resolution of references originating within nested scopes ([MD30]
p.4-1).

The only reason one can see for this would be that there is no need for
the binder to reset the lexical levels for every local reference.

6.6 A Diversion On Limitations of Data Scopes in Block Structured
Languages (BSL's)

It will be apparent that this writer disagrees with the "natural"
feature of implicitly global names. Fundamental changes are needed to BSL's
to ensure that only those names intended for use can be used in a particular block.
We would not make a formal proposal at this stage, but would suggest that all
blocks be named. A declarative mechanism could then be invented which causes
all declarations in a named block to be valid (and added to the compiler-produced
listing) and global in the block concerned. We might also invent a Record definition
which can be an aggregate of any definable entities, for definition purposes only
it must be*, so that subsets of a block's declaration can be made available by
name.

6.7 More "Advanced" Languages

We have so far dealt (in a limited way) with the more common languages
where we have only simple data types and procedure calls. Our survey has not been
exhaustive as both COBOL and PL/1 have been omitted. It would seem more
appropriate to examine two more "advanced" languages, PASCAL and SIMULA in a
little more detail and to make some observations on ALGOL68.

* We may or may not wish to allow operations on non-elemental parts of a record.
6.7.1 PASCAL

The language PASCAL [GL18] requires no introduction. In place of an
eulogy of the language we will just remark that its use of data types and record
structures constitutes a significant advance on existing languages at the time
of its introduction (1971).

6.7.1.1 PASCAL "Types"

One of the language's most important features is its use of data types.
In most languages, the domain and range of data types are implicit, being determined
by hardware factors such as word size etc. It is therefore possible to test the
value of an external data item in a deterministic fashion even though the code
which creates it is compiled separately.

PASCAL, however, permits the user to declare arbitrary data types with
arbitrary domains and ranges. For example, [GL18 p.34] we can declare

    type colour = (white, red, blue, yellow, purple, green, orange, black);

The declaration has the effect of making the identifiers constants of type colour
and we presume that identifiers can only be declared once.

Naturally, actual machine constants will be used in the place of each
identifier, but it appears that the values chosen may be allocated globally within
a single compilation. We note, however, that if an expression's involving
arbitrary (non-numeric) types are constructed only of variables and constants
of one type, then consecutive integer values can be used, starting at any suitable
data. The only other requirement would be a pointer to a doubly linked list of
the type so that operators indicating successor etc. will work on variables.
type colour = (white, red, blue, yellow, purple, green, orange, black);
var paint: colour;

FIGURE 6.7.1 A possible data structure for PASCAL type
We would require that set declarations generate rings of pointers, and sub-ranges have a range descriptor associated with them. Without pursuing this problem too far— a treatise on the implementation of PASCAL is not in order—it is apparent that the proposal would allow external data types to be accessed.

In fact, code of the form

```pascal
type colour;

external var paint: colour;

if paint = orange then ....
```

will work correctly since the compiler knows that "orange" is a constant of type colour—if we said that undeclared identifiers were assumed to be constants.

However, such an approach is totally unacceptable. It would seem that an external variable would need to be of external type.

While a much simpler scheme than that in Figure 6.7.1, such as the allocation of a set of consecutive constants could be used to represent the basic type, some linked structure is essential for sub-range declarations.

Before passing further judgement, we will look at other features of the language.

6.7.1.2 PASCAL "Case" Statements

PASCAL [GL18 p.31, p.35] requires that the labels on case statements should be constants. We may write, using the previous declarations:

```pascal
case paint of
    red, blue, yellow:S1;
    purple:S2;
    green, orange:S3;
    white, black:S4
end
```
It will clearly be a little more difficult to produce efficient code for the example if paint was an external variable of type colour, and the range and domain of type colour were unknown.

6.7.1.3 PASCAL "Records"

The record is quite a flexible structure which can involve the declaration of new constants or use existing ones [GL18 p.42-49, p.141].

The record concept permits records with variants, and these need not be of equal length.

We could possibly elucidate information about a record from the fact that any operation involving its components will fix their type. The fact that only some components are referenced in a program to which a particular record is external is no major problem since a qualified external reference could be written.

6.7.1.4 PASCAL "Pointers"

Pointers [GL18 p.62-66] are a means of linking instances of one variable to another and/or of accessing dynamic variables (instances of a variable created by "new" - ref. GL18 page 63).

Pointers are bound to a type at declaration time. Dynamic instances of a variable can be accessed only via a pointer. One presumes that the static and dynamic instances can co-exist simultaneously since references to them are distinct. Clearly, a pointer can be external, either in its own right or as a component of a record.

6.7.1.5 Procedures

Procedures in PASCAL follow the normal pattern and [GL18 p.78] functions can return only scalar, sub-range or pointer types.
6.7.1.6 Requirements for External Items in PASCAL

From what we have seen, we could apply the "implicitly global" concept and safely obtain enough information to link to an externally defined (i.e. separately compiled) item. However, this would be a blow against the language designer's intentions.

One would expect therefore that the best way of handling the situation is as previously recommended and including complete definitions for any external items in both separately compiled pieces of source.

We would require, as was implemented in the DEC 10 ALGOL60 (Section 6.4.5.1) that the scope of an external procedure is the scope of its external declaration in the calling source and that an environment list (Section 6.4.3) would specify the variables to which the procedure is external. The environment block is the external procedure would contain complete declarations of all external items.

6.7.2 SIMULA

SIMULA is another programming language rich in data structures and concepts. The comments are based upon Birtwistle et al [GL20], a modern exposé of the language which represents a considerable evolutionary step on earlier versions such as the one implemented on the B6700 [MD21].

SIMULA is primarily a simulation language and as such has provision for timing of events and dynamic creation of many items of one CLASS each of whose local attributes have different values.

CLASSes may also have procedure bodies associated with them, and these "bodies" may, through the use of activation and suspension operations, be in different phases.

The procedures constitute action rules and simplify the maintenance of different states since the next statement to be executed for any particular instance of a class is known to that instance of the class. Without going into the precise structure of declarations, we can nevertheless present the general
concepts.

Local "items" are accessible outside a CLASS.

If A is local to a CLASS body and X is an instance of that CLASS then
X.A specifies the A for that particular instance of the CLASS body.

X will need to be declared as REF (<class>)X
i.e. X is a pointer to entities of <class> ([GL20] pp. 36-37).

CLASSes may have REF mode variables as entities hence if we have REF(CIRCLE)
and the CLASS CIRCLE contains the declaration REF(POINT)P where CLASS POINT has
local attributes X and Y then C.P.X and C.P.Y specify the attributes for the
particular POINT associated with a particular CIRCLE. (GL20 p. 122-).

We already noted that local entities can be procedures, and a procedure
reference can be handled in this way. An example of the situation which can
arise is that of the CLASS LINE which contains a procedure MEETS which is
REF(POINT)MEETS. If we have two pointers to items of CLASS LINE, L1 and L2,
then

L1.MEETS(L2) is the pointer (REF) to the intersection of LINE-

instances pointed to by L1 and L2.

The procedure body in CLASS LINE has access directly to the co-ordinates of L1
and by qualified reference to those of L2.

6.8 On Separate Compilation

Currie [GL21 p. 29] at the IFIP70 conference on the implementation of
ALGOL68 mentioned the need for separate compilation of structures (data definitions)
as well as procedures.

We can see from the previous sections that the declaration of data items
in languages such as PASCAL, SIMULA and you can be sure PL/1, ALGOL68 and COBOL
can be quite lengthy. Hence, the need for separate compilation.
However, it is implied by our discussion that these declarations cannot be "external" in the same sense as a procedure since they determine the syntactic correctness of the program. They cannot therefore be "bound" to later unless the external declaration contains enough information to generate code - and this requires a complete declaration.

It would seem more appropriate to allow for separately "compiled" libraries of declarations (and even procedures in some cases) from which a compiler could extract information.

In its simplest form, a library maintenance processor (for declarations only) would syntax-check only. A more advanced system might produce symbol table input data for the compiler. A suitable "include" directive could be added to the language specification.

I suppose we have a selective prelude in ALGOL68 parlance.

Separate compilation of procedures would be available also.

6.9 The Effect of Global Items at Different Lexical Levels on Loaders

The foregoing discussion covers most of the "modern" languages, but in fact says little about effect of global items in block structured languages on the loader. Section 6.4.5.1 skirts around the technique used in the DEC System 10.

If the proposal in 6.4.3 is used then it would be possible to associate with each lexical level containing an external reference to a particular procedure a sub-symbol list of the items global to the external procedure. Since a procedure may be used at different levels, the external reference itself may consist of a set of pointers to these items.

Figure 6.9.1 shows a manageable relationship. This form is suggested since the global items need never be placed in the main symbol table. Once "crunch" has been found, the linkage is completely specified.

If, however, the DEC system 10 approach is used, then we have a more complex problem. Every declared symbol must be loaded into the symbol table since it is a potential global for all or any external procedures.
begin
real a, b, c, d;
integer my, mz;

external procedure crunch;
  environment a, my, mz;

  crunch (b);

begin
  integer my, mz;
  external procedure crunch;
    environment a, my, mz;

    crunch (c);
end

end

FIGURE 6.9.1 Possible Data Structure for Global Items
A global in the external procedure must then be linked to the symbol with the same name and highest lexical level of similarly named symbols within the scope of the external declaration.

Not so difficult to do, but more than a little messy!

A simpler approach [GL24] makes use of the fact that if a stack-oriented implementation of a block structured language is used, then "linking" for global items is accomplished at compile time if the address couples are known. We can force the compiler to "know" the lexical level of a variable declaring the level number. Once the variables (up to and including the last) variable required by the "separate" procedure are declared, then the correct address couples are available. There can be more than one level if need be.

Our objection would be that the procedure can then only appear at one "scope" in program, an undesirable restriction.

6.10 **Keyword Parameters in Procedure Calls**

Hardgrave [GL26] has suggested that the keyword and default system of parameter specification accepted in advanced macro processors should be used for procedure calls.

His proposal would work ([GL26] p.54) in the following way:

```
CALL S(P_{k1}=Q_{k1}, \ldots, P_{km}=Q_{km});

END

PROCEDURE S(P_1, \ldots, P_n)

DEFAULT P_1, V_1;

\vdots

DEFAULT P_n, V_n;
```

Where for each \( k_i \in \{ k_1, \ldots, k_m \} \), \( P_{k_i} \) is bound to \( Q_{k_i} \). Then \( V_j, 1 \leq j \leq n \): \( j \not\in \{ k_1, \ldots, k_m \} \), \( P_i \) is bound to \( V_j \).
This can be extended as follows:

6.10.1 Defaults need not be provided for all parameters.

6.10.2 Unspecified parameters (i.e. those with no positional or default specification) will produce a load-time error.

And we can add that there should be no run-time overhead involved in this scheme since it involves permuting the parameter list as specified on the calling side to make it correspond with the formal list, a linking-loader function.

The problem of the scope of non-constant defaults of course raises its head.
7. SUBROUTINE AND EXTERNAL SYMBOL LINKAGE AND THEIR RELATIONSHIP WITH TARGET INSTRUCTION SETS

7.1 Introduction

We must also consider the role played by the instruction set in the linkage problem. Implicit in what has been written in the previous chapter is that eventually a reference to an address must be obtained. We also must pay some attention to the parameter transfer problem since the possible mechanisms are related to the instruction set.

7.2 Types of Linkage

7.2.1 Control Transfer

We must be able to jump to the external program.

7.2.2 Data Reference

Data in an external program may need to be accessed.

7.2.3 De-Referenced Data

Immediate constants are possible candidates for external linkage [MD11 p.8 ]

7.2.4 Subroutine Linkage and Parameter Transfer

Mechanisms are required for retention of return addresses and parameter transfer.

7.3 Instruction Set Types

We cannot provide a complete survey of all available instruction sets - this is beyond the scope of this work. However, the following discussion will have more coherence if we describe some general classes of instruction set and separately examine their effect upon the loader.

7.3.1 Non-Relative, Direct Addressing Machines* (Class A)

These machines use an absolute address when referencing memory. Alternatively,

* We really mean "normal mode of use". Most of the machines with the exception of the HP2100 which is without index registers could reference data relative to index registers. Problems arise, as will be seen from Class B machines.
they use an address relative to some programmer inaccessible register. Examples of the former are the CDC 3200, HP2100, VARIAN73 and so on. Examples of the latter are DEC10, UNIVAC 1100 series.

While it is possible to program some of these computers as if they had programmer accessible register relative addressing, it is not generally done. Most importantly however, it is possible to reference some data and program as absolute, and external linkages normally take this form. This point will be dealt with later.

Machines of this type may have indirect addressing. They may also have, for word-length reasons, restricted or several address field sizes. The HP2100 has an address field size of 10 bits or a range of 1023 [MD9 pp.2-3 - 2-5]. References are either to a word in the current "page" or to page 0 (the base page). Indirect addressing in either page provides access to a full address range.

The VARIAN 73, however, provides for direct addressing to the first 2048 words and indirect, indexed to the first 512 words and PC-relative addressing in the positive (forwards) direction [MD22 pp.14-1, 14-3, 15-5].

An extended mode is provided for memory reference instructions in which the word following the instruction contains a 15 bit address field with an indirect bit (bit 15) [MD22 pp.14-1 - 15-6].

The DEC10, CDC3200 and CDC3600 provide one address field size covering the complete addressable memory range. Incidentally, the PDP11 is in this class also.

7.3.2 Register-Relative Machines (Class B) (Displacement plus base)

This class is reserved for those machines in which all addresses are encoded as relative to a programmer addressable register. The most common (and as far as I know the only) modern example is the IBM360-370 instruction set [MD25, GL1, GL9 pp.66-86] and its derivatives. This instruction set combines the worst of all worlds since the address field size is 12 bits. Any absolute address must be obtained by address or displacement constant to a specified
base address to obtain a 24 bit address. There is no provision for indirect addressing, and all jumps use either the displacement plus base mechanism or the contents of a register as an address.

The last two are of less importance - what is important about such a computer is that:

1) any location in physical memory is addressable (subject to memory protection) and the actual address is not mapped in any way which is outside the programmer's control (except for logical-physical mappings.)

2) an address reference within the complete addressing range can only be generated via the base plus displacement mechanism.

3) displacement values of zero are valid and a base added register specification of zero means that no base address is specified - similarly for index registers ([GL9] p.76).

This means that an address held in a word can be loaded into a register and then used since if one of the index or base registers is specified as register zero and a zero displacement used, then the location addressed is specified by the contents of the non-zero register.

7.3.3 "Proper" Stack Oriented Machines (Class C)

Computers which are genuinely stack oriented (hence the word proper - some systems have stacks but not full stack capability with lexical level addressing) and provide some form of lexical level addressing are of significance. We have dealt with the problem of block-structured languages to some extent in Chapter 6. We will assume for the purposes of our discussion that addressing is by address-couple and on-stack descriptor, and that address-couple decoding is transparent to the programmer.

7.4 Memory Organization

We need to consider two cases which may or may not occur simultaneously.
7.4.1 **Direct Address Range Less than Physical Address** (Variant 1)

Some computers do not provide sufficient address bits for accessing the whole of physical memory. Surprisingly enough, the DEC10 model K1 has this property, as have many mini-computers.

7.4.2 **Virtual Memory** (Variant 2)

Some mechanism may exist for mapping contiguous logical addresses onto discontinuous physical addresses, and this may also provide for automatic detection of "faults" i.e. references to non-resident locations.

7.5 **Specifying External References in Object Modules**

We must, at some point in the loading process, be able to locate all instructions which make references to each external symbol. There are two basic methods which can be used.

7.5.1 **Use of Available Instruction Address Space to Specify the External Symbol.**

7.5.1.1 **Linking**

There may be sufficient space in an instruction to enable a link to the next instruction using a particular external reference to be held. This approach is used in the CDC3200 MSOS loader and relocatable decks for example. It requires that address space sufficient for the whole of loadable* memory is available. This approach is also used in the DEC10 and many other systems.

Loaders using this approach were traditionally loading direct to memory, so a table could be constructed of all external references and the linked-lists "threaded" together through the memory load. Upon completion of the subroutine selection process, the "threads" could be followed to insert actual addresses.

This process involves two passes of the "load". One to read it into memory, the other to actually plug the addresses.

* i.e. the volume of memory which can be addressed reasonably by a single program.
7.5.1.2 Pointing

An alternate approach, where the direct addressing space is limited (e.g. HP2100) is to place an external reference symbol index in that space. The HP2100 permits direct addresses with values less than 1024 (10 bits). A single program with 1K of external references could be considered to be a monster, and so the figure is reasonable.

Further, the linkage must, in such a case be indirect through a full word in the current page or this page. [MD6]p.E-4 BCS shows the relocatable deck format.

It should be noted that it would be possible if some form of relative indirect addressing is available, for the language translator to place a full word link at a suitable point in the code, and to use the "linking" technique.

This would be possible with the HP2100 instruction set and the VARIAN 73 for data references.

However, the "pointing" situation implies some form of transfer vector, and this simplifies the work of the loader.

The allocation of the transfer vector addresses requires only a knowledge of required external symbols, and this information can be made available prior to "reading" the object module to obtain actual absolute addresses. Further, a reference to an external symbol can be "plugged" in the calling program without actually knowing the address-value of that symbol. This means that the routines to be loaded do not need to be scanned completely more than once.* The alternative approach involves the construction of a complete memory map prior to the actual linking process.

However, the relevant instruction in the object deck needs to be marked as containing an external reference, complicating the object deck format.

---

* We are not, at this point, making assumptions about the ideal form of object decks and libraries.
FIGURE 7. Basic Organization and Use of Transfer Vector During Linking
7.5.2 Symbol Dictionaries

The approach taken on the IBM360 is different again. The instruction set is such that an external reference can only exist as a full word absolute value, which can be loaded into a register because the full words address can be known at compile time.

Linking would be a suitable mechanism, but the IBM Linkage Editor [GLl p.157] uses a physically separate link record which indicates both the external symbol and the address of the constant within the text. This approach is a little expensive in terms of disk storage, but permits the construction of a readily editable and loadable module.

7.5.3 Relationship with Instruction Set

The symbol dictionary approach can be used anywhere with any instruction set. In fact, the only method which is restricted is that of linking, which requires full memory addressability. We should also note that any transfer vector approach based upon hardware constraints will have the problem that only a certain number of linkages can be provided.

7.5.4 Stack Architecture

7.5.4.1 General

The above examples do not really distinguish between program references and data references. It is possible, with stack architecture that data references will be to the stack and that program references need not be. In 7.3.3 we assume that procedure references are via on-stack descriptors. The addressability problem repeats itself, accept that in this case, the item being linked is the address-couple for the external reference.

7.6 Address Space and its Effect Upon Linked Programs

We remarked in 7.4 some computers cannot directly address the whole of physical memory. To this should be added the fact that there are many machines
which can only address 32K words even though much larger physical spaces are available. The case of the PDPL0, with a logical address space of 256 K words is unremarkable except that completely separate "processes" may require access to common data - a problem on a virtual memory machine.

What follows has been derived in part from discussion with a number of people, in particular P. Herman and Prof. C.S. Wallace.

There are essentially two separate problems:

1) data addressability, and

2) program addressability.

For the former, we can say that except in one case, some form of register relative addressing with displacement of the order of 1024 storage units will suffice. We are assuming here that it is possible to group data into records and that operations on a single record are sufficiently frequent to justify the overhead of loading a register.

The existence of some global constant or individual datum in an external subroutine now leaves us with a clear-cut option. We can provide an extended address (c.f. the V73) mode in this case and allow for it at compile time. This at least overcomes the need to alter the instruction size at linking time, one of our taboos.

For the latter, we could provide PC relative addressing - this would permit logical addresses of more than the address space to be generated without producing additional words at load-time.

However, unless we can actually retain enough bits in all cases, our solutions are illusory. Either we need a larger logical address space or we don't. The solution would seem to lie in part in making address-forming registers large enough to generate the required address range. In particular, the PC should have enough bits to permit addressing of the whole of physical memory, even though address increments may be considerably smaller. This will also be necessary for data-base registers since data may lie anywhere in the (logically) addressable space.
Our argument would seem to lead to a computer design with some registers wider than the word size - I am not aware of any case where this has been done as yet.

Before we consider this question further (it will not be analysed in detail since it is beyond the scope of this project,) I must mention two further points.

Firstly, the above solution (wider PC word and relative addressing) is not satisfactory where single jumps are produced which exceed the jump-range. Additional "steps" must be generated - but not by the loader. An alternative is to allow for a moderately large transfer vector whose entries (necessarily multiple words) are suitable addresses.

Secondly, there would seem to be a problem with the size of the return address, since this will exceed one word. However, PC relative return addresses could always be used (?). This would leave us with the problem of variable sized return addresses to allow for "long" jumps.

Returning to our problem it should be noted that we have asserted that the addressable space must be quite large. Perhaps the term "reachable" space is more appropriate. "Reachability" is less of a problem with record structures than in program jumps.

Practical answers to the "reachability" problem may be achieved by allowing access to a larger mapping table on a virtual machine. That is, the page table for each process may contain enough entries to allow quite a large volume of memory to be addressed, and special instructions provided for "switching" virtual space. Problems can arise here also.

The most significant of these is that logical addresses in the program cannot be altered. This means that either programs use some data in the same logical space, in which case individual programs cannot use the whole logical address space and page table logical - physical mappings must be preserved or the programs must be written in such a way that common data areas in mutually accessible physical space are specified. This latter solution involves mechanisms for allocating
such space (manageable) and correctly loading the page-tables (variously difficult).

These problems are similar to those faced in data and process integrity.

As a matter of interest, the overheads involved in altering page tables are relatively high on a number of systems. The solution involving relative addressing plus a transfer vector seems to have a lot to recommend it.

Higher level translators are capable of detecting intra-program transfers which exceed the range and directing them to the transfer table. Inter-program references could be automatically directed through the transfer vector and the loader allowed to detect relative jumps in these cases to conserve transfer vector locations.

A special transfer vector register could be used to point to the transfer vector start address, and finally, transfer vector entries might be of variable length.

All of this assumes that the instruction size is large enough to allow for a reasonable relative range and transfer vector size.

Experience on machines such as the HP2100 and LME.APZ.130 suggests that transfer vectors of less than 1024 words are too small for a relative jump range of 512 on a 64K word 16 bit computer, hence at least 10 bits are needed for address specification plus one for jump mode (relative or table).

Design, in such cases, is a series of compromises.

It should be noted that the Data General Eclipses provide for two jump instruction sizes (permitting full logical addressing of 32K) [MD26 p.2-23 to 3-25] except for the stack oriented PUSH JUMP instruction (PSHJ) which is of extended length. A conventional subroutine jump instruction with two lengths is also provided. The VARIAN73 has only one jump instruction length - [MD22 pp.16-25 to 16-37] two words.

A paper by the author [MD27 pp.9-12] suggests a technique of using adjacent global page tables, a novel but difficult to administer approach.

It would seem that the problem may be simpler on a stack machine since
the address of any item to be accessed can be constrained to lie on the stack
and have variable size. The stack can act therefore as a special kind of transfer
vector. All "global" addresses could be placed on the stack at some (low level)
and either referenced at that point or copied into the correct scope at procedure
entry.

Concluding, we have tried to treat a major problem and done so incompletely.
Suffice it to say that there are problems in this area.

7.7 Parameter Transfer

This problem is decidedly more straight forward in many respects. There
are two basic techniques for storing parameter lists used on conventional computers.

It should be noted that the techniques used in any particular case are
a matter of convention, depending upon the instruction set to some extent. The
requirements of programming languages also intrude.

The first of these is "in-line" parameter transfer.

In this case, the addresses follow the subroutine calling instruction.

The second case (of which there are two variants) involves a pointer
to a parameter list being provided. This may follow the subroutine call or be
passed through a register.

In any case, the language translator must generate sufficient data to
allow type checking at linking time, and this data can be separate from the code.

As already mentioned, the location of the parameters is always available.
However, since we may have address-range problems of the kind indicated earlier,
we are still in some small difficulty.

We do not need to distinguish between parameters as single entities,
and parameters as fields in one or more records. We must acknowledge that in
fact both types of parameters may be used, and that they will have the same
address-range problems - except that a record address can provide access to
multiple data items whose individual addresses have not been "passed in".
What we really require is the ability to reference a parameter as the nth parameter of the caller directly and allow address evaluation to be done using the specification in the calling program.

For practical application, this approach requires at least two things:
1) a special type of indirect reference which performs a complete address evaluation using the calling programs address space
2) a method of making this indirect reference via the caller's address.

An examination of the PDP10 instruction set [MD24 p.1-11] suggests that this can be done if the calling address were placed in a nominated register. And in fact, the overheads will be no worse than the method used in FORTRAN 10 which involves copying addresses across to a buffer area and making indirect references.

However, our "ideal" machine would have a special register which would contain the parameter string address - and also would access datum registers as loaded at calling time.

But enough of such rambling! All we are doing here is earmarking some problems and not offering proper solutions.

7.7.1 Parameter Transfer Mechanisms - Address Plugging

A technique which was used in the CDC3200 FORTRAN system involved run time linking.

A subroutine was generated with parameter references as dummy addresses. When called, the first thing the subroutine did was to copy the parameter addresses into instructions using an SWA (Store Word Address) instruction which altered only the address field in a word.

There does not seem to be any instruction set feature which would have made this technique preferable to any other. It seems to have been a deliberate attempt to trade parameter reference overheads against subroutine call overheads.
7.7.2 Parameter Transfer Mechanisms - Parameter String Duplication

The technique of parameter string duplication is used as a standard on machines which can and cannot readily modify an address field. The DECl0 FORTRAN10 system uses it and so does the HP2100.

The latter cannot readily modify its address fields while the former can.

This constitutes a different kind of trade off, that is single level indirect addressing as against multiple level and address plugging.

7.7.3 Parameter Transfer Mechanisms - Instruction Sets with No Indirective

As we have pointed out, indirect addressing can provide a suitable means of reducing the overheads associated with parameter transfer to a realistic minimum without making calling overheads too severe.

Not all computers with indirection make use of it in this situation as has been remarked.

The instruction set for the IMB360 which does not have indirect addressing, creates an interesting problem for the programming convention inventor.

From the FORTRAN User's Guide we find that ([MD29] pp.146-155) the parameter address is passed in register 1 and the nominal return address in register 14.

Calls are via a BALR or BAL instruction, and these will load a register with the address of the next instruction (actually, current PC value which is already incremented). If the parameter string is "out of line" (the FORTRAN standard) then we actually have the return address in GPR14. Otherwise it must be stepped over the string of address which follow the BAL[R].

The addresses in the parameter string are readily obtained by

\[ L \quad R_1, D_{P1}(1) \]

where \( D_{P1} \) is the displacement to parameter 1 and \( R_1 \) is the desired register.

The parameter itself (if a scalar) can now be loaded to another register by

\[ L \quad R_2, 0(R_1) \]
and can be moved to local storage directly

\[ MVC \quad VAR(B),0(R_1) \]

This means that:

1) Either the **actual** parameter is copied across to local storage (and back if altered) so that "normal" address calculation can be performed.

OR

2) Two registers and two instructions are used to obtain every parameter reference.

The second possibility may not be much slower than single level indirect addressing, but it requires an additional instruction for each parameter reference.

Modifying the called code presents problems since all addressing is base register relative and by convention, the same GPR is used as base register in all subroutines.

However, the problems are less severe when references to a "passed" array are involved, since arithmetic must be performed upon the address anyway.

The problem will be less significant when accessing parameterized data structures of a more general kind when comparing with non-parameterized ones.
8. MATHEMATICAL ANALYSIS OF PROGRAM PRODUCTION

8.1 Preliminary Analysis

Let the total source volume in a suite of programs be \( L \) lines.
Let the compiler resource utilization be \( u \) units per line.

We can assume that the total number of compilations needed to obtain
a working system is proportional to the total volume so that we can assume that
there are \( c \) compilations required per source line.

... the total resources used to complete the system if it is a single
program will be:

\[
U_S = u c L^2
\]  

1.1

If the system is partitioned into \( n \) programs, then the production "cost" (in
terms of resources) is:

\[
U_{MC} = u c \sum_{i=1}^{n} L_i^2
\]  

1.2

assuming

\[
L = \sum_{i=1}^{n} L_i
\]  

1.3

Initially we will assume that all programs are of equal size so that:

\[
U_{MC} = \frac{u c L^2}{n}
\]  

1.4

since

\[
L_i = \frac{L}{n}
\]  

1.5

But the programs must be linked together, and this will be done for each
compilation of each subprogram.

Let the resources used by a single link operation be \( U_L \), then \( U_{ML} \), the
total linking effort is:
\[ U_{ML} = (\text{total number of compilations}) \times U_L \]  

\[ n = \sum_{i=1}^{\infty} cL_i U_L \]  

which in any case is just:

\[ U_{ML} = cL U_L \]  

The linking process itself will be considered as having two parts:

Processing the intermediate or relocatable files;

and

Handling external references.

If we assume that, allowing for compression of data produced by compilation etc., there is a simple ratio between the "speed" of the compiler and the loader, we can say that for each source line, the loader will use:

\[ \frac{u}{M} \text{ resource units per source line} \]  

Now

\[ U_L = U_{LS} + U_{LR} \]  

where \( U_{LS} \) is the "source" component and

\( U_{LR} \) is the "external reference" component.

\[ U_{LS} = \frac{n \sum_{i=1}^{n} uL_i}{M} = \frac{uL}{M} \]  

which is distribution independent.

\( U_{LR} \) however, presents some more complex problems.
The total number of cross-references can reasonably be assumed to be a function of the number of modules and the total source length involved, i.e. \( R(n, L) \). Adhering to our concept of a source line as a unit of work, we can say that each cross-reference presents a load of \( K \) source lines so that:

\[
U_{LR} = \frac{uKR(n, L)}{M} \tag{1.12}
\]

since the work done for each source line is in compiler units. Selecting a function for \( R \) is more difficult, but it seems to me that the assumption of proportionality to the product \( nL \) is extremely pessimistic, so we shall use it.

Absorbing the constant of proportionality and \( K \) we obtain:

\[
U_{LR} = \frac{urnL}{M} \tag{1.13}
\]

where \( r \) is the equivalent number of source lines per module per line of total volume which are produced for all cross references. Of course, we can write

\[
r = e^K \_r
\]

where \( e^K \_r \) is the equivalent source lines/xref and \( K \) is the number of xrefs/module/line.

Now we have:

\[
U_L = \frac{uL}{M} + \frac{urnL}{M} \tag{1.15}
\]

so that we can write:

\[
U_{ML} = \frac{ucL^2}{M} \left( 1 + rn \right) \tag{1.16}
\]

and

\[
U_M = ucL^2 \left( \frac{1}{n} + \frac{1 + rn}{M} \right) \tag{1.17}
\]

The ratio

\[
\frac{U_M}{U_S} = \frac{1}{n} + \frac{1}{M} \left( 1 + rn \right) \tag{1.18}
\]
If we assume $r << \frac{1}{n}$ then

$$\frac{U_M}{U_S} = \frac{1}{n} + \frac{1}{M}$$

so that if $n$ is large we have:

$$\frac{U_M}{U_S}_{Lt\rightarrow\infty} = \frac{1}{M}$$

or

$$\frac{U_M}{U_S}_{Lt\rightarrow\infty} = \frac{1}{n}$$

which is even nicer!

8.2 Conclusion

The preliminary analysis shows that the ratio of the cost of compilation with the use of a linking editor to that without is:

$$\frac{U_M}{U_S} = \frac{n + M}{nM}$$

where $n$ is the number of source modules and $M$ is the ratio of compiler to loader speed. If $n$ is large then the cost ratio is $\frac{1}{n}$ i.e. it will cost $n$ times as much if a loader is not used. If the speed ratio is large, then the cost ratio approaches the speed ratio.

In any case, for reasonable values of $n$ and $M$, significant savings result.

This analysis does not consider the effect of library referencing, and this is a suitable area for extensions.
OVERLAY PROCESSING AND ITS RELATIONSHIP WITH TARGET INSTRUCTION SET

9.1 Introduction

(A complete Chapter has been allocated to this subject, but it will be short).

The fundamental problems of Overlay Processing have already been mentioned. Further, in discussing the parameter transfer and "reachability" problems, we have described part of the problem.

However, while 4.3.6 contains quite a detailed discussion of the general question, the problem of parameter transfer was left out.

9.2 Parameter Transfer and Overlays

It should be noted that in the "natural" course of events, parameters can only be passed "out" in overlay invoking calls. For example, in the overlay structured shown below, only certain calling sequences can involve parameter transfer in most systems.

![Diagram of a simple overlay structure]

FIGURE 9.1 A Simple Overlay Structure
Table 9.1 shows the calls which would include parameter transfer.

<table>
<thead>
<tr>
<th>CALLED ROUTINE</th>
<th>SUB01</th>
<th>SUB11</th>
<th>SUB12</th>
<th>SUB13</th>
<th>SUB02</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLING ROUTINE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAIN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SUB01</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>SUB11</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SUB12</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SUB13</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>SUB02</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Legend**

✓ parameters transferrable
x parameters not transferrable

**TABLE 9.1 Parameter Transfer Calls in an Overlay Structure**

We could argue further that this problem cannot actually be handled satisfactorily for the following reasons:

1) Any parameter list is an arbitrary subset of the totality of data-names "known" at the point of the call.

2) Following from this, an arbitrary re-arrangement of storage is needed for all possible calls to be allowed.

Such a re-arrangement could be considered to be beyond the scope of a reasonable linking loader design.

However, what we really require is a technique for making parameters "virtual" so that their "logical" addresses and physical realization are preserved while all other data space is overlaid (and saved for reinstatement on return). The problem here is that the "logical" addresses so used are no longer accessible to the overlaying block of code - which presumably needs all the space it can get.
We can copy the parameter list into an appropriate root segment without difficulty, but the parameters themselves?

What we are in fact doing is to combine virtual memory with overlaying, a step which can be necessary if programs which exceed logically addressable memory are produced, or, as mentioned in 4.3.6 if logical address space limitations are placed upon individual programs.

We really require, however, that logical addresses outside the current allowable address range can be generated, and that these can be assigned the physical addresses which are those of the parameters. This has the effect of increasing the actual logical space used by the amount of data in the parameters.

The proposal made in Chapter 7 at least allows for the necessary address range.

We should remark at this point that overlaying solves the problem of inadequate logical space while virtual memory solves the problem of mapping logical to physical addresses.

We keep returning to a computer architecture involving two mappings, one to obtain a logical address of an address, and the second to obtain the actual physical address. The number of named objects is going to be significantly less than any address range - a "named" object is one for which a compile time constant address is required. Further, the number of named objects passed as parameters will be a small subset of all "named" objects.

If we could separate the parameter-lists from the program itself - say onto a "stack" or linear address-space of some kind, and were to allow logical addresses larger than physical memory (or at least as large) in the form of a descriptor of the B6700 variety, which allows for individual data items* to be present or absent from memory.

* primarily arrays
We can suggest then that the parameter list for an overlay invoking call which will destroy the calling program, can be located in a non-overlayable area at load-time, that all writable data local to the overlayable area be saved, and that the descriptor mechanism handle access to the parameters (which may have been saved) as part of its normal memory management function.

We would further allow, for reasons of economy, for named commons to be handled as one entity - and force the compilers to mark them as not needed if it can be discovered that they are not accessed in the current program.

It should be noted that the above discussion has some relationship to conventional techniques for handling block-structured languages. But it is not the same.

This approach would place all parameter strings which were the subject of overlays in a non-overlayable area. A proper stack is not needed.

Of course, the whole approach can be taken one step further. Let the parameter list be a single segment with a descriptor (in the B6700 sense), then all that needs to be resident is the descriptor. The parameter list is properly a property of the program not the run-time stack, even though call-by-name operations may force references to the stack.

The descriptor can either be placed on the stack or compiled into the program. Again, the memory management mechanism can be allowed to handle the situation as best it can.

9.3 Conclusion

There does not seem to be a great deal more which one can add at this time. However, Chapter 4 discusses the question of overlaying and remark upon some design considerations independent of the instruction set.
10. AN EXAMINATION OF AUTOMATIC CALL LIBRARY ORGANIZATION AND
   CREATION ON SOME EXISTING SYSTEMS

10.1 Introduction

Chapter 5 does make some reference to the problem of automatic call
libraries and their organization. However, the problem is examined only from
the point of view of the ability of file handling systems to support this function.

We will look again at the problem from two points of view. Firstly,
the overall question of structure and secondly, factors which influence performance
other than the library-directory structure. Reference will be made to one or
two existing systems.

10.2 Call Library Organization on Existing Systems

10.2.1 IBM360/370 OS

The call libraries on the IBM360/370 OS are just standard partitioned
data sets which, as we have pointed out, have a directory attached to them. Entries
in this directory contain the sub-file name and its location. Names may be
ALIAS's, that is, they may point to the same entry. The BLDL macro [(MD10) p.25-26]
is provided for multiple reads of the directory. A list of names are provided,
and the directory-data extracted by BLDL. The macro STOW [(MD10) p.231-233]
allows for the creation of new directory entries.

However, the directory is a simple unordered list of names, and the
macros provided do not allow for a sensible search strategy - of course
they may not be used by the linkage editor anyway.

Of more significance to our argument is the fact that the compilers
do not update the PDS directories when producing object modules. The naming
of a member is done in the JCL card specifying the object dataset. This means
that ALIAS entries in the PDS directory, necessary if multiple entry points are
to be handled, must be added by another utility. In fact the normal mode
of use of the IBM linkage editor is such that automatic call resolution would
play a small part in linking. The user would normally specify the names of members
to be included and the dataset (via a dd name) which contains it.

10.2.2 Burroughs B6700

As pointed out in Chapter 5, the B6700 software will create suitable
qualified names but does not allow for multiple entry points.

10.2.3 HP2100 DOS

The library on the HP2100 just consists of the subroutines themselves,
no directory structure is provided.

10.2.4 DEC System 10

The DEC10 linking loader, LINK-10 ([MD7] p.A-8) has a simple library
structure in which all the entry points for a given program are collected together
(one presumes at the start of the library file) and followed by a pointer to the
individual module which contains them.

Libraries are created by package called FUDGE2 which combines relocatable
decks.

10.3 The "Search the Module" Syndrome

All of the above library organizations can only be used in a particular
fashion. Once an external reference has been found, the loader can locate it in
the directory and find its "owner". However, the external references produced
by that module can only be found by examining the module itself. Having done
this, the loader can now search the directory anew, hoping for a "resolution".

In most cases, time will be lost due to head-movement and latency delays
without allowing for the problems of data read.

Also, efficiency of linking on the B6700 becomes a function of the
directory look-up technique.

We could conclude that none of these approaches are optimal, but there
are additional factors to consider before leaping in.
It should be noted, however, that if all the entry points and external references for each program were collected in a directory, then the programs to be loaded at least could be determined completely without reference to the load modules.

If we further adhere to our restriction that program volume may not be altered at load-time in such a way that the value of global symbols is altered, then we could also determine the value of such symbols without reading the program. This is not the case if a transfer-vector approach of some kind is used, since the "linking" address is independent of its actual value.

However, we now place a significant demand upon language translators, that is, they ought to be able to create and update such directories.

Alternatively, special library building software can be created.

10.4 Some Comments Upon Directory Structure

10.4.1 "In Store" vs Direct Access

The external symbol resolution problem can be stated as follows:

After having "loaded" primary input, we have a list of say e external references which will be resolved completely by P₁ programs in the library.

Necessarily, P₁ ≤ e. However, P₁ + P₂ programs may be loaded since any of the P₁ first level routines may add new externals to the list as well as deleting them.

Our model of the process involves actually "resolving" a program against the list at the point of discovering that it contains a required external.

Simple directories which contain both what could be called the program skeleton (name, address of program, entry point list, external symbol list) may fit in core. A simple sequential search, performed cyclically will not in this case require access to external storage.

The directory structure can be simply extended to the following form so that unnecessary searching is avoided:
FIGURE 10.1 A Simple Directory Structure

This organization would increase storage requirements marginally but would reduce cpu usage significantly. However, the directory is essentially minimal in size - there is little redundant data.

We have not at this stage considered the problem of holding the transitive closure of the reference set for each entry point (to be considered later).

However, we can examine the effect of any design decision which would cause one directory look-up for each of the \( P_1 + P_2 \) routines loaded and require data to be held on backing storage.

10.4.1.1 Holding an Entry Point List in Core

If the ENTRY point list is in core and the program skeleton is on a random access device then at most \( P_1 + P_2 \) probes (actual disk accesses) will be needed to locate the skeletons.

10.4.1.2 Location of Entry Points Requiring Access to Disk

If we assume that there is some way of configuring the directory so that
each program skeleton can be located by using its entry points as keys, then we can estimate approximate performance from the literature.

Random access files will obtain a record with an average of less than 1.15 probes for blocking factors and loading factors of the kind we might consider appropriate (Montgomery [RR5] p.9). This is lower than an earlier figure ([RR7] p.148) of 1.3

Relating this to index sequential files is difficult for a number of reasons, not the least of which is that our reference ([RR6]) does not use the same timing measures.

Pure random access files produce good average access figures, but may perform badly for certain non-optimal sets of keys. And there is no way in practice of preventing this from happening.

Chained techniques at least have the limiting property that the maximum number of probes cannot exceed the length of the longest chain.

The writer's experience with internal hashing in an experiment where the standard deviation of accesses as well as the mean was measured suggests that even reasonably good techniques could degenerate.

Tree structured files at least offer a bounded upper limit for accesses. Balanced binary trees never have a depth greater than \( \log_2 N \). Multiway trees are more promising.

An average access (number of actual reads or probes) of

\[
E_R = n - \frac{(B_R^n - 1) - n(B_R - 1)}{(B_R^n - 1)(B_R - 1)}
\]

is quoted in [RR8 p.10]. \( n \) is the depth and \( B_R \) is the branching factor.

Knuth ([RR3] pp.473-479) describes a more sophisticated self organizing structure called a B-tree which has very large (up to 128) branching factors. These can be combined with software paging (as can other techniques) and he reports an experiment ([RR3] p.479) in which the average number of probes to file was .786 for a retrieval - the number of levels was 2.
If we were to presume that in fact it would not be possible to access skeleton data directly, and that our EFT directory cannot be held in memory, then we will accumulate disk accesses at a rather rapid rate as will be seen in 10.4.6 where some comparisons are made.

We will consider the likely optimal design at that point.

It should be obvious that the total size of the library and the number of entry points will be the factor determining the approach to be taken. Unfortunately, we do not have a great deal of data on this, except that libraries such as those on the HP2100 and CDC 3200 contain hundreds of programs.
More Elaborate Directory Structures

Re-stating the Problem

That our design should be optimal for a multi-program system has been pointed out by Peter Herman. The in-store solution from 10.4.1 will be extremely fast, but could consume large amounts of storage.*

We will consider an initial comparison of directory structures based on worst case performance in a later section.

What we wish to do now is to consider a more elaborate directory structure, examine techniques for constructing it and consider its performance.

We have already said that loading primary data will lead to a list of e external references which will be satisfied by external symbols in \( P_1 \) programs in the library. These may in turn generate further external references which may be satisfied in the library. Providing the list of all programs referenced directly and indirectly (the referenced set) by each of the primary programs means that the list of external references cannot increase after each "resolution" against the library. We can, if sufficient data is provided to satisfy all external references in e which match external symbols in a given referenced set, reduce the number of accesses to the directory to at most \( P_1 \) where each of the \( P_1 \) are peripheral programs.

What follows involves consideration of basic properties of the library, so we shall consider these first and then elaborate.

In fact, our referenced sets are just the transitive closures of the individual \( P_1 \) routines.

Now, every program in the library will have a referenced set, and the originating program and its referenced set will form a directed graph. The process of forming the referenced set for each program in the library will create a set

* Such judgements are of course relative. However, the design should be such that program library size is not a limiting factor.
of graphs.*

This set of graphs will be unique in form but will have nodes which appear in several graphs. Further, graphs may be sub-graphs of each other so that there is significant redundancy in their representations.

Reference Graphs

However, a different situation arises if we form graphs which consist of all programs which reference each other and are referenced by each other.

These graphs are unique by definition.

They have the property that the referenced set for any vertex is the transitive closure of that vertex on the graph.

They also have the property that referenced sets can be defined in terms of referenced sets.

We will define peripheral nodes to be those with zero in degree ([RR10] p.109). i.e. those with no predecessor. Terminal nodes are those with no out degree.

We would also note that maximally connected sub-graphs (i.e. sub-graphs in which every vertex has a path to itself) may exist and these can be loaded when a reference to any vertex is detected.

We have already remarked that a referenced set completely satisfies a reference to its head in the sense that no new reference to the library can exist. The referenced set is the transitive closure of the vertex referenced.

We can at this stage introduce a design restriction intended to minimize head movements in the directory:

DR1/ Referenced sets will be represented so that any link out of a physical record is in the direction of increasing relative record number.

The effect of this will depend upon the length of referenced sets and we have not at this stage considered what data should be held in referenced sets.

* We mean directed graphs.
10.4.2.2 An Example

Figure 10.2 shows a reference graph, and Figure 10.3 shows the referenced sets for each vertex in the graph.

Note that C, F, H, E form a strongly subgraph and hence their order of appearance is irrelevant in the referenced sets. We can construct a list for this graph which has property DRI and write it down simply as follows:

1) the symbol ] is a "stop". Sequential scanning cannot proceed beyond ].
2) an arrow above a letter indicates that there is a means of getting to the corresponding program (via its entry points).

FIGURE 10.2 A Possible Reference Graph
3) A chain forwards is shown by a directed loop between two letters.

4) Maximally connected programs appear as a sequence with an arrow to the first in the sequence. It is assumed that entry points map to the referenced set accordingly.

5) The lists (called referenced set lists) are read only from left to right.

The RSL for Figure 10.2 is as Figure 10.4

```
+  +  +  +  +  +
B ] A D G ] E C F H G ]
```

FIGURE 10.4 Referenced Set List for Figure 10.2
An algorithm which could be used to obtain the referenced sets appears in Berztiss ([RR10], p. 220-223) and one would expect to find more suitable algorithms elsewhere (Berztiss' algorithm is not applicable for more than 50 vertices), e.g. [RR9].

The algorithm for reading a list simply takes an external reference and obtains the appropriate entry to the reference list from a table.

The list is read forwards from the entry and forward chains are accumulated in a (sorted) list. When a "stop" is found, we chain forwards to the next (nearest) member. The search terminates when we have a "stop" and an empty chain-forward-list.

The precise action and detail of data will be considered on another occasion. We assert that conceptually at least this form of directory has advantages.

However, let us consider some other properties of libraries before examining the possibilities.

10.4.3 Clustering of References

An obvious point which must be noted relates to the clustering of references. For example, related mathematical subroutines are likely to be referenced in a single loading operation, service routines for I/O type conversion, multiple-precision etc. may form groups which, while not strongly connected or even in a reference graph, will be loaded together with high probability. There are two aspects of this problem.

10.4.3.1 Grouping Clusters

We require a technique for grouping the program skeleton data so that we can match a group of programs against e in an attempt to reduce total access time. Our buffer-sizes may be such that a number of program skeletons can be in memory at one time. Grouping could be achieved by sub-directories based upon some compiler generated information. We note that the B6700 binder already allows for this, but its efficiency is unknown. Grouping could also be achieved through
an intermediate directory structure, the entry point names being mapped to related groups.

10.4.3.2 Relationship between the Grouping Properties of the Library and other Directory Structure

The possibility of combining the grouping and referenced set properties needs to be considered, since the two requirements could conflict. We begin, in effect, to talk about groups of reference graphs, and we need now ask whether the reference list structure is required.

10.4.4 The Information Needed in Reference Lists

10.4.4.1 Minimum Information

We can consider the amount of data which needs to be stored in reference lists - do we in fact need complete program skeletons of entry point symbols and external references? The answer is no. External references are required only if an out of library reference exists since the reference sets are "closed".

Similarly, entry points are not required.

Consider the following arrangement:

![Diagram](image)

**FIGURE 10.5 Data Structure for Reference List and Directory**
Our strategy is to load the skeleton data for the conglomerate, which, for the inter-conglomerate links contains the pointers to the separate programs in the library, as well as the pointer to the conglomerate. The separate program addresses are held as the "correct" ones in the cross reference table in storage, except in cases such as (b) where a reference to D forces the loading of E and F. When a reference to any program in such a conglomerate is found after the conglomerate is "detected", the program address is set to the conglomerate address.

Prior to actual loading, the loader must detect those groupings in which all routines were not required and set their states accordingly. There are a number of ways of doing this.

10.4.6 Worst Case Comparison of Loader Resolution Performance

10.4.6.1 General

We can examine the performance of a loader which builds a list of subroutines to be loaded from a call library considering the efficiency with which the list is constructed. A detailed analysis is beyond the scope of this work (and the author's present ken) but is a possible source of research problems.

Our worst case analysis will consider primarily external device access times and transfer rates - no allowance will be made for cpu utilization.

10.4.6.2 Directory Structures

We have three models of directory structure:

10.4.6.2.1 Compact Program Skeletons

A directory which consists entirely and only of the external references and external symbols for each program together with its address in the library proper.

10.4.6.2.2 Direct Access Using Unresolved External References as Keys

We provide a directory access table which uses the external symbols as keys to locate blocks in the skeleton file.

10.4.6.2.3 Minimal Information Data Structure

This uses the referenced set list structure and a directory to obtain the referenced set lists (c.f. 10.4)
10.4.6.3  **Block-Search Strategy**

In all cases, the loader will search the complete block (or blocks) in the skeleton directory starting at the first complete entry and resolve all references in and within that block. Strategies of clustering without conglomeration can then have some effect (This strategy would be a subject for a further investigation).

10.4.6.4  **Basic Parameters**

Our basic parameters are as follows:

Loading primary data constructs a list of external references e of length |e| which will be resolved by |P₁| programs in the skeleton list. These will be called primary programs. References from these to the library will cause an additional |P₂| secondary program to be loaded.

Disk access time is assumed to be τ seconds, transfer rate is Tᵣ units/second where the units are those used in the block sizes Bᵣ.

Directory lengths are to lᵣ units.

The * is one of C, D, M for the three cases respectively.

10.4.6.5  **The Calculations**

10.4.6.5.1  **The Compact Program Skeletons**

In this case, the loader cycles through the directory until no new routines are added to the loading list.

The time taken is:

\[ t_c = R(P_1) \frac{1}{B_c} (\tau + T_B) \text{ secs} \]  \hspace{1cm} 10.1

R(P₁) is the number of cycles through the directory, and is a function only of the set P₁ and the library.
10.4.6.5.2 Direct Access Directory*

In this case, there will be access to the external symbol directory which is a function of the set \( P_1 \) needed to satisfy the set \( e \). Assuming optimal P.A. performance we have:

\[
  t_D = A_D(P_1)(2.15\tau + B_D R_x)
\]

In the worst case \( A_D(P_1) = |P_1| + |P_2| \)
i.e. one access to the external symbol directory is needed for each program loaded. Note, a block is read for each external symbol directory access.

\[
  t^* = (|P_1| + |P_2|)(2.15 + B_D R_x)
\]

10.4.6.5.3 Minimal Information Data Structure

In this case, there will be access to the external symbol directory to obtain the set \( P_1 \), but no secondary references will occur.

\[
  t_M = A_M(P_1)(2.15\tau + B_M R_x)
\]

In the worst case, \( A_M(P_1) = |P_1| \)

\[
  t^* = |P_1|(2.15\tau + B_M R_x)
\]

10.4.6.6 Comments

There does not seem to be much point in examining these equations in detail at this stage. Too little is known about the form of the functions on \( P_1 \).

We could perhaps allow ourselves a little licence just for the hell of it, however.

* We assume that directory blocksizes are small enough to be ignored.
1. $R(P_1)$ is not likely to exceed 10.

2. We can assume all block sizes equal at $B$.

3. The sum $|P_1| + |P_2|$ can be assumed to be (say 150) with $|P_1| = 20$.

We now have:

$$t_c = \frac{1}{B} (T_X + T_B)$$

$$t_D = 150 (2.15T_X + T_B)$$

$$t_M = 20 (2.15T_X + T_B)$$

which leaves us with the comparison between the first and the last techniques as being worth while.

Of course, such a conclusion cannot be made since we need to consider actual values of $A_D$ and $A_M$ for given sets $P_1$.

Nevertheless, the simplest case (10.4.6.5.1) cannot be discounted.

10.4.7 Other Factors Affecting Directory Design

We have remarked that more than one library may be used. There are two cases to be considered. If the two libraries can reference each other, then we want to be able to treat them as a whole. This requirement tends to invalidate directory strategies where access technique is affected by individual library size e.g. hash coding. Otherwise, there is no problem - provided we can make this assumption.

Proper call libraries will not reference each other often. However, libraries of programs constructed in the process of implementing a large system may as has been remarked.

We are not in a position to comment, except to say that the proposal presented in Appendix A-1 allows the user to control this to some extent.
Further, any group of subroutines compiled in one operation are potentially a library. Directory construction needs to be within the "power" of compiles and should not cause running costs to rise. The Burroughs approach is good from this point of view.
11. EXPERIMENTAL RESULTS

11.1 Introduction

It has been argued (in Chapter 10 and elsewhere) that the process of constructing the "partial loads" can have a significant effect upon the performance of linking loaders. It was decided that adding some form of directory to the program library used by the loader on the HP2100 system at Monash University's Computer Science Department, would be a suitable practical exercise for this project. The value of such an improvement could then be gauged.

11.2 Original Concept

The original proposal involved a sophisticated cross-reference directory capable of enabling complete partial loads to be identified in one access for each subroutine in the primary load (Ref. 10.3).

Subsequent investigations and the passage of time, resulted in a simple linear directory being constructed. Chapter 10, Part 3, includes arguments which suggest that this is sufficient if the symbol resolution phase of loading is a significant overhead. It is argued further (10.3) that any directory which can be made memory resident, or which can be "swept" in a small number of disk accesses will be faster than even optimal random-access techniques where the number of subroutines loaded is large (20 or more).

11.3 Final Choice

It was decided therefore to construct a directory consisting of all relevant information from the program library; in other words, the record types used by the loader in determining which subroutines were to be loaded. As will be seen, this directory is about 20% larger than it need be.

The existing loader was modified by Appleby in 1973 and has I/O buffers occupying some 19000 words of main memory. The proposed directory would fit quite easily.
In principal at least, the modification was simple since much of the existing library searching code could be used. The final program contained only about 220 new lines of code, however, diagnostic versions contained about four times as much.

11.4 Original LOADR Library Search Strategy (After Appelby)

The original loader library search mechanism was extremely simple. A loop was provided which caused the library to be searched while there were undefined references which had been created by previous library searches. This can be determined as:

"undefined references created by library search if routines loaded by library search and there are undefined references".

The actual library search consisted of saving the address of each subroutine and examining its ENT records to see if any matched undefined externals in the list of external symbols which was being constructed.

If a match was found for any ENT symbol then the subroutine in question must be loaded. The saved address is then used to re-read the subroutine, adding its name and address to a load-list and its ENT and EXT symbols to the symbol list. A check is then made to see if any external references remain undefined, and if they do, the scan is continued.

Points of interest include the interlocking and compact code in the library search. Students of classic structured programming will find little to enthuse them since the same code is used differently in each phase of the operation.

Library data is read via DISKI which uses Appleby's DEXEC to buffer I/O. However, the system itself works from a 128 word buffer. A considerable amount of effort is spent in the original loader in administering this 128 word buffer, notably, subroutines are provided for stepping a pointer to the current word and for skipping the remainder of a record (see Figure 4.2 for library record formats). The latter
Fig 11.4.1 Control Loop For Library Scan

Programs
Loaded count
set to zero

LODLL

Open Directory

Directory

Directory NOT PRESENT

PRESENT

Initialize Buffers

LODLR

Any Programs Loaded?

NO

YES

Any Undefined Ext.?

NO

YES

Clear Program Count

Scan Library

YES

Programs Loaded?

NO

Reset Buffers

Reset Buffers

LODLR

LI.Math
Fig 14.2 Summary of Directory Records (Ref [MD5 E-4 BC5])
calls the former until the record is cleared. In fact, arithmetic should be used to calculate the location of the next useful word.

In the main library, each subroutine starts on a new sector boundary - a wasteful practice.

11.5 Directory Structure and Generation

11.5.1 Directory

As already stated, the directory is a simple linear affair containing all library data except for program information records (DBL records), packed (as they should be). The directory contained about 104 sectors of data as against 750 odd for the library itself.

The directory is not optimum since:

11.5.1.1 unused words in the NAM record are included,

11.5.1.2 END records are included.

It is estimated that space overhead of about 20% results from this. The modifications needed to both the new loader and the directory builder are noted elsewhere.

Word 3 of the NAM record is used to hold the subroutines actual address in the library.

11.5.2 Directory Generation

The directory is generated by a rather large and probably unnecessary program. (The loader contains all the code needed to analyse the library, and DEXEC, the buffering routines needed to read and write - hind sight is a marvelous thing.) However, it produces a nice listing of the subroutines in the library including entry point and external symbols and some statistics. Simultaneously it writes the required data to a disk file. The programs are listed in Appendix 3, and operating instructions are included. The directory is terminated with a special end record of type 7.
11.6 Modifications to the Loader

11.6.1 Modifications to Library Search Control Loop

The library search control loop was modified so that on its first entry, DEXEC is called to alter the buffer allocation so that there is one large buffer. On exit, DEXEC is called to reset the buffer allocation to eight smaller ones (this reset was omitted at one stage and the result was a much slower loader).

11.6.2 Modifications to Library Search

The modifications can be summarised as follows:

11.6.2.1 All access to data in the 128 word buffer must be via the pointer stepping routine. The original made use of the fact that the NAM record started in the first word of the buffer.

11.6.2.2 The record detection mechanism was altered to recognize a type 7 record as the end of the directory.

11.6.2.3 The track, sector and word address of the NAM record is saved to permit back tracking (N.B. the NAM records no longer start on sector boundaries).

11.6.2.4 Correspondingly, the actual word address of the NAM record must be calculated when re-scanning a required program's directory entry.

11.6.2.5 The pointer stepping routine also must step the record length counter so that records are skipped correctly. This was necessary because the original code assumed the pointer was always pointing at the first word in a record when the decision to skip was made.

11.6.2.6 The flow was somewhat simplified.

11.6.2.7 Word three of the NAM record was stored in the load list.

Otherwise the two mechanisms are the same.
Flow Chart of Modified Library Search (with Directory)

Control Loop

The number of routines loaded is set non-zero.

LODLL

- Locate directory (EXEC)

- Re-initialize buffers (DEXEC)

LODLR

- Any routines loaded?
  - NO
  - Any undefined externals?
    - YES
    - Set routines loaded to zero
    - Search library and update lists (LIBS)
      - YES
      - Any routines loaded?
        - YES
        - Reset buffers (BUFRE)
        - LMATN
        - NO
      - NO

- NO

- Reset buffers (BUFRE)
Flow chart of Modified Library Search (With Directory)

Library Search

1. ENTER
2. ENTER?
   YES → ENTER
   NO → add to LIST (SELST)
3. SET TO undef. EXT.
4. ENTER
5. SET ordinal
6. EXNBD
7. Searching?
   YES → ENTER
   NO → set name
8. Clear BP
9. Link
10. ENTER
11.7 Flow Chart of Modified Loader

The flow chart shows major items only. The existing subroutines are not flow charted.

The flow chart should be considered in conjunction with the listing of the library search.

The directory is "read" via subroutine DISKL.

DISKL sets a word in the base page via a call to SETCR, enabling DISKI, the routine which actually performs the reads, to read from user disk. DISKI resets this word.

11.8 Modifications to Buffer Management System

Appleby's modifications included the introduction of a buffer management system which was responsible for a significant improvement in the original system.

The original subroutines included the buffer size and number of buffers as Fortran DATA items. It was necessary to add an operation to DEXEC calls which flushed the existing buffers (with their current size and number), reset the buffer size and number (in our case to one buffer of maximum size) from the parameter list. In fact, the buffer number and length in words were "passed in".

The buffer descriptors were then passed as parameters to the subroutine REJCT which performs buffer rejection the strategy.

The new buffer size was made available to the loader in a word in the common area. (Improvements include placing all buffer constants in common and initializing them from DEXEC to ensure that these constants are declared in a logical place vis. with the buffer maintenance mechanism.)

The modified code is item 3 in Appendix 2.

11.9 Possible Further Improvements

11.9.1 Improving the Efficiency of the Lib-Search
11.9.1.1 The pointer used to access the directory could be directed at the I/O buffer itself, and a count kept to detect the end of the buffer being reached. This would save the unnecessary copying from the I/O buffer to the loader's working buffer.

11.9.1.2 A count could be kept of the number of unresolved symbols remaining in the symbol list. At present, the list is scanned after each program is processed to see if any unresolved symbols remain.

11.9.2 Improving the Directory Access

11.9.2.1 The directory can be contracted as noted in earlier parts. The END record can be omitted, saving four words per program. The NAM record is actually 17 words long. Of these, a total of 8 words are needed. These are:

<table>
<thead>
<tr>
<th>WORD NO.</th>
<th>CONTENTS*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Record Length</td>
</tr>
<tr>
<td>2</td>
<td>Record Type</td>
</tr>
<tr>
<td>3</td>
<td>Library Address</td>
</tr>
<tr>
<td>4-6</td>
<td>Program Name</td>
</tr>
<tr>
<td>9</td>
<td>Length of Common</td>
</tr>
<tr>
<td>10</td>
<td>Program Type</td>
</tr>
</tbody>
</table>

TABLE 11.1 NAM Record Usage

We should note that record length and type could be combined with ease. This would give a saving of 10 words for each NAM record. Combining the record length and type saves one word per record of all types, but we do not know how many records there actually are.

*In head-modified loader.
The relocatable address for each ENT symbol is not used either, so we save one word per ENT symbol.

<table>
<thead>
<tr>
<th>ITEM</th>
<th>SAVING PER ITEM</th>
<th>NO. ITEMS</th>
<th>NO. WORDS SAVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAM Record</td>
<td>10</td>
<td>296</td>
<td>2960</td>
</tr>
<tr>
<td>ENT Symbol</td>
<td>1</td>
<td>406</td>
<td>406</td>
</tr>
<tr>
<td>EXT Symbol</td>
<td>4</td>
<td>296</td>
<td>1184</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TOTAL SAVING</strong></td>
<td></td>
<td></td>
<td><strong>4550 words</strong></td>
</tr>
</tbody>
</table>

**TABLE 11.2 Total Savings In Directory Space**

The total saving exceeds 4550 words or about 36 sectors. Total directory length is about 104 sectors, so the saving is considerable.

11.9.2.2. A hash coded access table containing indexes to the program and ENT symbol would speed up the library search further. See Figure 11.

**FIGURE 11.** Hash Table for Access to Directory.

This would require a complete re-write of the library search process.
11.9.3 **Comment Upon the Improvements**

The savings achieved by adding the directory were quite considerable, and approach the maximum achievable. Reducing the size of directory is worth while and fairly simple - but speeding up the library search does not seem worth while. Inefficiencies elsewhere in the system are now more significant.

11.10 **Timing Experiments**

11.10.1 **Estimated Maximum Savings**

The library is 752 sectors long at the time of writing, and is therefore 15 tracks long. Appleby's load used buffers which were 16 sectors long i.e. one-third of a track.

A total of 45 reads are required then to scan the library.

If we assume that the process is I/O bound, then the total read time is:

\[ T_{RA} = [47(\frac{T_L}{3} + T_L) + 15T_{H_{min}}]P \]

since one full rotation of \( T_L \) seconds must occur before the next read, and \( T_{H_{min}} \) records is the time to move the head one track.

\( P \) is the number of passes.

The values of the variables for the disk on the HP2100 are:

\[ T_L = 0.025 \text{ sec} \]

\[ T_{H_{min}} = 0.010 \text{ sec.} \]

\[ \therefore T_{RA} = 1.7 \text{ P secs.} \]

The savings actually achieved suggest that the number of complete scans being made is occasionally quite high, perhaps as high as 6 (see table 11.4), but this has not been verified.
11.10.2 Timing Experiments

A total of four loader operations were chosen to verify the gains expected. The loads were run with and without map requests, and the standard timing print outs used.

Table 11.3 shows the details of the loads involved.

<table>
<thead>
<tr>
<th>JOB TITLE</th>
<th>NO. OF FILES SPECIFIED</th>
<th>JBIN USED</th>
<th>NO. OF ROUTINES LOADED</th>
<th>NO. OF LIBRARY ROUTINES LOADED</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMLO</td>
<td>3</td>
<td>No</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>SFCRD</td>
<td>10</td>
<td>No</td>
<td>39</td>
<td>29</td>
</tr>
<tr>
<td>LTEll</td>
<td>1</td>
<td>Yes</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 11.3 Examples Chosen for Testing**

Table 11.4 shows the elapsed times obtained. It will be seen that the savings depend, as would be expected, upon the amount of library searching performed, however, job LTEll was a typical Computer Science student exercise.

<table>
<thead>
<tr>
<th>JOB</th>
<th>ELAPSED TIMES (SECS)</th>
<th>WITH MAP</th>
<th>WITHOUT MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FASTL</td>
<td>LOADR</td>
<td>SAVING</td>
</tr>
<tr>
<td></td>
<td>Secs</td>
<td></td>
<td>%</td>
</tr>
<tr>
<td>LMLO</td>
<td>16</td>
<td>23</td>
<td>7</td>
</tr>
<tr>
<td>SFCRD</td>
<td>23</td>
<td>28</td>
<td>5</td>
</tr>
<tr>
<td>LTEll</td>
<td>24</td>
<td>36</td>
<td>12</td>
</tr>
</tbody>
</table>

**TABLE 11.4 Timing Experiment Results**

These experiments show that a significant saving in loader overhead has been achieved.
11.11 Incorporating the New Loader in the System

11.11.1 Remarks

The new loader needs to be incorporated in the system, and this requires some minor changes to the directory builder (DIRBL) and to the loader itself. These changes can be summarised as:

(a) permitting the directory to be part of the write protected system area on disk.
(b) having the loader locate the directory on system-disk and read it,
(c) having DIRBL write the directory onto the system disk.

11.11.2 Proposed Implementation

11.11.2.1 Directory-Space

The actual directory space will be a "program" which a system generation time will consist of 32K of zero words. This can be done using the assembler pseudo REPL.

11.11.2.2 Writing Directory

DIRBL will attempt to create a directory if the directory has zeros in its first sector. DIRBL will need to be modified so that it locates this "program" and writes to it.

11.11.2.3 Use of Directory

The loader will be modified to
(a) seek the directory on user disk
(b) read from user disk.

The latter change involves only omitting the call to SETCR from subroutine DISKL, the former a change in the EXEC call at LODLL.

11.12 Diagnostics

A moderate amount of effort was expended in providing diagnostics. These were built on Appleby's LISOT, a format-less I/O routine.
The extensions were:

11.12.1 To provide a routine (LISOP) capable of building a "line", and carrying out administration of line counters. A simple-minded approach was adopted which worked quite well.

11.12.2 To provide a routine which functions as CHKAB does, using LISOP, so that the formatter is not needed.

A "diagnostic" version of LOADR existed at one time, using these routines. Both this listing and LISOX (the source file containing the routines) appear in Appendix II.
12. FACTORS TO BE CONSIDERED IN THE DESIGN OF A GENERALIZED LINKING LOADER

12.1 Introduction

In an investigation of this kind, whose emphasis has shifted from the practical to the theoretical, and from the specific to the general, it is inevitable that the question of generality of results should arise. The author, has the view that a software implementation as a solution of a real world problem ought only to be done once.

Accordingly, there exists that embryonic concept of the generalized linking loader.

Such a system in its wildest and most fanciful realization would be capable of accepting input in a standard (data) format but which would contain pre-compiled output from any language translator for "loading" (or rather linking) for any machine.

The above specification is probably not realisable but the examination of the problems involved, which, incidentally has occurred to an extent in the preceding sections, may at least throw some light on how the problem might be tackled.

With this in mind we can restate the problem and indicate what might be a realistic specification for such a problem, and proceed, in this section to comment.

12.2 Project Specification - A Generalized Linking Loader

12.2.1 To specify a linking loader capable of handling the linking problems arising from current and projected languages on current and projected systems.

12.2.2 To propose a design which will be efficient irrespective of host system.

12.2.3 To examine the impact of these two requirements, and those of linking upon hardware design.
To provide, in the implementation, a set of standard and defined interfaces for access to symbol tables, object code building and formulating. Of course, it is the substance of these interfaces which will be significant.

In other words, a system can be constructed which would be largely re-usable in the sense that while the precise code may not actually service in its entirety, the basic structure will. Further, the mechanisms necessary for later modification will exist. Lastly, we will get a glimpse of the effect of these requirements on hardware.

12.3 Scope for this Paper

We propose only to discuss these problems, drawing on previous material. The limited project of 12.2 is beyond the scope of our current work, but could prove to be a sequel.

12.4 The Factors to be Considered

12.4.1 Linking Mechanism

A linking mechanism needs to be selected. This must allow for efficient "plugging" of addresses in the load module.

12.4.2 Resource Allocation

The various types of resources to be allocated must be allowed for. These resources will be determined by target computer architecture and by the overlay scheme to be used. Primitives need to be available for controlling new types of target computer resources.

12.4.3 Relocation or Binding

Mechanisms for actually relocating the executable code for various computer types must be provided. We must include in this handling both the generation of memory mapping tables and address space problems as already described.
12.4.4 Symbol Table Processing

While not actually part of the generality of the loader, symbol table processing will have critical effect upon its performance.

12.4.5 Source Language Related Problems

Our generalized loader needs to allow for the problems of binding between and to various languages.

12.4.6 Object Module Format

A suitable object module format needs to be chosen and must allow for point 12.3.4.

12.4.7 File Handling System

For the loader to be transportable, it must contain a file-handling system capable of storing object and load modules, and various directories.

12.4.8 Linker Command Analysis

There must be a suitable set of commands for the linker. These must permit:

12.4.8.1 specification and control of library usage,
12.4.8.2 specification of overlay structures where appropriate
12.4.8.3 specification of modules to be included.

Chapter 4 of this report contains the basic requirements.
13. **THE CONCEPT OF A GENERALIZED LINKING LOADER**

13.1 **Introduction**

When this work was commenced, the author did not appreciate some of the problems involved. The loader's function was perceived primarily in the traditional form of linking simple external references in a simple address space. (See Chp. 12).

Within this context, which holds for a significant number of modern computers and languages, we can consider generality in the sense of universality. That is, we can postulate the existence of a loader capable of producing executable object modules for more than one machine.

We would face two problems; firstly defining a suitable relocatable object module format, and a technique for handling different instruction sets. Some obvious and simple resources could be allocated also, and some address length cases handled. The comments in Chapter 12 are essentially addressed to this aspect of the problem.

However, when considering more advanced data structures and programming languages (Chapter 6) it becomes apparent that true generality that is the ability to handle inter and intra-language linking for existing and unforeseen languages, presents difficulties. To this we can add the problems of various, as yet unknown forms of computer architecture beyond those considered in Chapter 7.

In fact, we are re-inventing a form of the UNCOL problem which is a well known source of difficulties.

The goal of generalization can be considered in another light as Prof. Wallace has pointed out. A possible approach to the problem of binding to undeclared data items discussed in Chapter 6 is a form of partial compilation in which the source language statements affected are only partially translated. The loader would complete the job. We will, in this chapter, consider the two cases in a little more detail under the headings Structure and Philosophy. In both, some material mentioned in earlier chapters will be extended, in particular, the question of resource allocation will be examined in the light of some real and hypothetical cases.
13.2 Structural Problems in a Generalized Linking Loader

13.2.1 A Loader Writing Language

Some of the structural problems could conceivably be handled by a Loader Writing Language (a LWL). It should be pointed out that there is a subtle but important distinction between the semantic problems faced by a LWL and an UNCOL. The LWL does not need to generate arbitrary sequences of instructions from a source code, in fact the input to the LWL program (i.e. the loader) is an object module and as such consists of instructions executable on the target computer.

The LWL must allow the loader writer to describe and fill variable length bit fields in instruction words. This of course raises a problem. Setting arbitrary width bit fields is notoriously inefficient on most cpu's in fact, many cpu's have fractional word operations which, by some strange coincidence, correspond to their address fields.

We would also need to declare tables and operate upon them. A table is a collection of identical items with access and modification operations based upon both a key and a position. The access techniques could be defined as part of the language. We would need to distinguish between tables held in core and those which may spill onto external storage.

We would want to define our operations in terms of functions required by the loader-writer.

The format of the various record-types in the object module is also a subject for inclusion in a language but it would seem that the mechanisms in PASCAL for example would be adequate.

Data declarations permitting multiple relocation registers to be created and evaluation operations permitting their use would also be needed. However, relocation registers would need to be available as values also.

The symbol table facilities can be used to handle the problems associated with representing external references - but the format of object decks will be fixed by compilers.
We should note that the loader writing language needs to be transportable, but that loaders do not.

It could be argued that the frequency with which loaders are written is sufficiently small to give such a project a low priority in research activity, however, it could permit users to rewrite loaders and hence raise their performance. The author is no longer convinced that this project has a great deal of merit. However, it would be a suitable step beyond the existing work.

13.2.2 Resource Allocation

Amongst the resources which a loader might need to allocate are storages of various speeds. While caches will not usually be programmer accessible, the user may have at his disposal large amounts of main-memory of varying speeds. He may wish, at load time, to nominate the location of various routines.

We have already mentioned the transfer vector problem. An interesting problem can arise in addition to those already discussed in Chapter 7.

A study made of 64K words of program on the AP2130 by Holmén and Reed (unpublished) showed that about one half of the entries in the transfer vector were referenced by only one instruction. This meant that the available transfer vector space could be increased considerably if some other position could be found for the address.

An obvious alternative is a two word jump instruction.

The Computer Science Department has considered another solution involving in-page links on the HP2100. Both of these solutions create the problem of altering the range of jumps across the "expanded" source as explained elsewhere, however, the loader could be reasonably expected to perform such optimization in critical cases when the vector is full.

Again, there may be some as yet unknown possibility lurking in the dark recesses of computer designers' minds, and this jeopardizes our generalized loader plan - but, since the LML would of needs, contain a usable programming
language, these cases could be handled.

13.2.3 Stack Architectures.

The last word on stack architectures for cpu's has not yet been uttered. Generality will present difficulties in this area.

13.2.4 Conclusion

We would argue that the LWL is a suitable research topic from several points of view, although we are not sure of its use. The proposal for a generalized linker, however, seems to have little merit unless restricted to a common class of computers and languages.

13.3 Philosophical Problems in a Generalized Linking Loader

13.3.1 Integration of Program Construction

Leaving aside the practical problems for a moment, we can pursue Prof. Wallace's proposal a little further and try to relate it back to the author's experience with the L.M. Ericsson APS language support system which went part of the way towards implementing the "compiling at binding time" approach.

The proposition is basically that since references to undefined data items of the kind one might find in PASCAL etc. make the production of efficient object impossible, one might consider a form of partial compilation which produced an intermediate code which could be translated at load time, when data definitions were available.

It is worth remarking that almost all existing systems produce an object deck which is "nearly" executable. The only additional information is that necessary to handle external references and relocation.

With few exceptions, the output from various compilers will be structurally similar, the special needs of block structured languages being handled by devices not requiring loader action. An exception is the PDP10 linker [MD7] (Burroughs reverse the problem and make non-block structured programs special cases of block structured).
We can consider that the compilation process is therefore almost completely closed, and that code production is integrated with "translation."

The proposal would reduce the extent of this "closure" and provide for code generation to be integrated with the loader.

Examining a program which contained external references, one would expect to find that it could be sectioned into regions in which these were of no relevance to the emitted code, except that they will affect the range of "jumps" across, to and from them. This remark will be true for any sub-set of the variables and for labels defined in a program when selected according to any criteria. Of course, this is quite different from considering the effect of using such variables.

The object code for a single statement could be translated into a standard form for a particular computer - we have no desire to re-invoke Uncol problems. The loader could then access the data definitions to complete the code.

This approach is perferable to any form of incremental compilation involving retention of the original source code.

In the absence of any optimization, the above approach is sufficient.

The effect of optimization, as described variously by Gries ([RRll] Chapter 18) is to physically move the points of reference of items so that they do not necessarily occur in their original place. However, some part of the original expression must be retained, and it is the smallest sub-expressions (in semantic sense which includes all "operations") involving the external reference which will appear in the intermediate form. There does not appear to be any reason for being more specific than "A is added to B" irrespective of operand typing, although optimization may be affected by the complexity of the operation, which is not known. However, an operation may involve two elements of complicated data structures, both of which may be external, or which are indexed by another external item of arbitrary type.
For some languages, the local data structure definition may need to be retained. An example is some assignment to an item in PLI or COBOL which is itself composed of sub-items. Hill ([RRL2]) discusses the problems of run time organization for ALGOL 68 and associated de-referencing and coercion problems. However, should one have a language in which access to a data structure is controlled by a procedure which is declared as part of that structure or associated with it ([GL27], and Simula [GL20]) then the picture becomes more complex since the assignment rule is not known either—it is also external.*

It is possible that these problems can be overcome, but a considerable part of the compilation process is being moved to the loader.

Further investigation of this question would seem warranted as part of a more general study of linking "exotic" languages.

It should be noted, however, that we are now approaching, from a different viewpoint, the question of separation of data from programs which also arises in data base considerations.

We are in fact arguing that the meaning of any operation is completely defined by the data structures upon which it operates, which is obvious.

However, the rules for generating the operation are only partly related to the language in which it is stated. If the structures are compiled separately, then we must produce sufficient information relating to the special effects of the operator and the data structure to allow code to be emitted. This can be in a standard form.

Such arguments are not new, but I am unable to provide references at this time.

It should be noted, however, that we can now introduce the possibility of type-less operations at the machine level where the need for de-referencing is detected and performed automatically by code (or micro-code) produced by the loader.

* This may not be as serious as it appears at first glance since once any item is external the de-referencing process is unknown and must be derived from other data.
13.2.2 The System APS

The author is only aware of one system which left object code production until binding * stage. This was L.M. Ericsson's APS, the software support system for the AKE13 systems.

Very briefly, the compilers produced what was called a target code which was related to the object or code in that it contained the necessary data needed to produce object code, even though this may include reference to as yet undefined symbols.

As has been already mentioned, the compiler output also contained unresolved macro expansions dependent upon as yet undefined symbols.

This system was not really a success, largely because of overheads resulting from design decisions which were thought to result from the need to access symbols undefined at compile time. The author was also able to show that in practice, the majority of symbols would actually be defined at compile time.

However, the proposal being made differs from APS in major respects. Only that code which cannot be evaluated is held in an intermediate form. Further, we are talking about a level of abstraction at which generalized operations have some meaning - APS was an assembly language system.

13.3 Conclusion

As already remarked, this seems a likely area of investigation.

* Actually, the HP2100 holds its relocatable code in an expanded form also.
TECHNIQUES FOR SPEEDING UP LINKING LOADERS

14.1 Introduction

The performance of linking loaders must be such that their use is justified. Frequently, especially on small computers, the user will feel that he would be better off compiling all the relevant source. Subroutine libraries of several hundred programs occupying a megabyte or more of disk space are quite common, so the design of mechanical parts of the loader should be taken seriously. Chapter 10, Part 3, deals briefly with the question of directory structure. Other "mechanical" details will be dealt with here. The actual "linking" mechanism will not be dealt with since it requires alterations to all language processors and/or a major re-write of the loader even if a post-processor is produced.

14.2 Mechanical Factors Affecting Speed

14.2.1 Blocking Factors on Sequential Files

The blocking factors on sequential files should be raised to a point where rotational delays are minimized. This will usually involve (c.f. IMB360) altering the file specification for compiler output. It will not be possible in some systems (e.g. DECI0 and some mini computers) since the maximum block which is likely to be allocated contiguously is limited. Some systems may not allow for blocking anyway. In such cases, modifying the operating system is worthy of consideration. Adequate storage must be available, of course.

14.2.2 Buffer Swapping

Control of the number of buffers in use during the loading is important. The buffers for each active file should be distinct and core resident to avoid excessive disk access. Each case needs to be examined individually, but problems can occur in this area where software is written to run with extremely small amounts of storage.

14.2.3 Adding a Directory where one does not Exist

A suitable directory can be added to library files if one does not exist.
Refer to chapter 10, part 3. The directory must contain enough information to at least resolve a load completely without reference to the library itself. Loader modifications may be extensive, but the improvements can be significant.

14.2.4 Control of Symbol Table Space

The user should discover whether or not his loader is overlaying its symbol table - if it is, he should endeavour to prevent it from doing so whenever practical.

14.3 Restricting the External Symbol Fan Out

The fundamental reason for poor performance of loaders is the fact that new external references can be generated in attempting to resolve a given external reference. Unless we use the "perfect" directory structure described in Chapter 10, we cannot prevent either multiple passes on or accesses to the directory (at least).

In practice, however, the majority of the routines required for a particular class of user will be known in advance. It is possible, in such cases, to "integrate" the appropriate source modules by a program which converts them to a single source program, and to compile this to produce an object module which satisfies all the users externals.

The user can, if the loader allows it, then specify the file containing this module as part of his input with the result that no library searches will occur.

Failing this, the separate modules can be placed into a short, special library or into a file to be loaded as described above. The gains would not be as great as in our first suggestion, but would be useful on machines with a poorly designed loader.

Examples are common run-time routines for a higher level language such as FORTRAN where most programs will require mode conversion, array indexing, I/O and some mathematical routines. Such a suite may cause more storage to be
used than optimal, but the time saved during the debugging phase may be significant.

Of course, the above approach requires little or no changes to the loader.

A more ambitious approach is reported by Barron ([GL4] pp.60-61). The approach outlined above was considerably extended. It was assumed that the "resolving suite" of programs would be loaded into store in a fixed location so that the module can be load module format.

These would be loaded first, so that the entry points automatically satisfy externals in modules loaded subsequently. Barron's report does not make it clear whether or not the system (FASTLINK) loaded to core or not. If it did, then the gains could be very significant since a special symbol table segment could be provided followed by executable code. Hence only the problem program would need to be processed.

The gains reported were significant - from a minimum of 25 seconds to a maximum of 5.

14.4 The Library Partition Solution

As has been shown in Chapter 10, a subroutine call library does not consist of n independent subroutines, it consists of r & n independent partial loads each of which contains only entry points.

This means that the library need only contain the r "partial loads" and hence there will never be cross linking within the library. As we have shown, the cross linking in a static library produces partial loads.

We must however, retain a list of all entry points. We would not be able to get at the partial loads unless we do.

14.5 Conclusion

Good data processing practice in the area of directory design seems to be critical. None of the systems examined seem to make even cursory efforts in this direction.
Beyond this, the structure of object modules, which must be affected to some extent by instruction set; and overall library search strategies (a function of loader specification) will affect the efficiency of the loader.

The points made here are obvious but must be listed.
15. CONCLUSIONS

15.1 This Particular Project

The results of this work are a major survey of the literature on linking loaders together with an analysis of the problems of their specification, and implementation with due regard to questions arising from linking separately compiled programs involving involved data structures (e.g. PASCAL, SIMULA, ALGOL68).

The relationship between subroutine transfer (and "global" references) mechanisms and computer architecture is discussed, with some suggestions being made which reduce the parameter transfer load and simplify the problem of "distant" references in systems with limited address ranges.

The problems of entry point directory structure are considered in some detail, and experiment conducted in which a simple linear directory was added to the HP2100 loader (after Appleby) at Monash University's Computer Science Department. This confirmed the view that a simple linear in core directory was sufficient as stated in the analysis of directories.

Other techniques for improving loader performance are noted and the question of generalized linking loaders viewed from several view points.

A specification for a practical loader is included in Appendix I and appropriate listings in Appendix II.

An attempt is made to justify the existence of linking loaders on economic grounds. The mathematics used is elementary, but has not been done before according to the author's knowledge.

Perhaps the only major omissions, as far as a complete being treatise on linking loaders, are a detailed consideration of address binding and the practical question of object code emission.

The sections on the generalized loader could also be strengthened, and the experimental work was minor, although it contributed to most of the delay in completing this work.
15.2 Future Work

The work which has been done on the HP2100 loader by this author and by Appleby (1973) has resulted in speed increases overall of about a factor of four, most of this being due to Appleby. The critical section of this work points out several deficiencies in loader design and implementation particularly in the area of overlay processing and directory usage.

However, there would appear to be room for further work in several areas. These include:

15.2.1 Collection of data upon the frequency of occurrence of external references in typical program environments.

15.2.2 Verification of the mathematical results obtained using information from 15.2.1 and actual statistics from projects involving large numbers of modules.

15.2.3 Development of an algorithm for constructing optimal directories.

15.2.4 Analysis of the behaviour of loaders which permit load module editing.

There seems to have been little work performed in these areas.
APPENDIX 1

SPECIFICATION OF A LINKING LOADER FOR A NEW MACHINE
3. The process of linking programs together is assumed to use time as function \( L \) of the total number of programs, external references and total source volume.

\[ i.e. \quad T_1 = L(n_p, n_e, \Delta) \]

where \( n_p \) is the number of programs
\( n_e \) is the number of external references
\( \Delta \) is average source statement volume

4. Assume that a complete system has been divided into \( n_p \) programs each of size \( \Delta_i \), \( 1 \leq i \leq n_p \).

Then \( T = k_1 k_2 \sum \Delta_i^2 \)

which is clearly much smaller than

\[ T_c = \sum \Delta_i \text{ is large and } \]

the \( \Delta_i \) are approximately equal.

In particular, assume that all programs are equal.

Then \( \Delta_{\text{avg}} \) is

\[ \Delta_{\text{avg}} \sum_{i=1}^{n_p} \Delta_i = \frac{\Delta}{n_p} \]

the \( \sum \Delta_i = n_p \Delta \)

\[ \therefore T_c = k_1 k_2 n_p \Delta^2 = k_1 k_2 \frac{\Delta^2}{n_p} \]

and \( T_c = k_1 k_2 \Delta^2 \)

\[ \therefore \frac{T_c'}{T_c} = \frac{1}{n_p} \]
5. Let us assume further that \( n_e = k_3 n_p \)

and that the cost per source line for linking is \( k_4 \). We can assume:

\[
T_1 = k_3 k_4 n_p^2 \bar{\Delta} \quad 1.10
\]

\[
T_c^1 + T_1 = k_1 k_2 n_p \bar{\Delta}^2 + k_3 k_4 n_p^2 \bar{\Delta} = np \bar{\Delta} (k_1 k_2 \bar{\Delta} + k_3 k_4 n_p) \quad 1.11
\]

and without

\[
T_c = k_1 k_2 n_p^2 \bar{\Delta}^2 \quad 1.12
\]

The ratio \( T_c^1 + T_1 = (k_1 \bar{\Delta} + \frac{k_3 n_p}{\bar{\Delta}}) / k_1 n_p \bar{\Delta} \)

\[
= \frac{k_1 \bar{\Delta} + k_3 n_p}{\beta k_4 n_p \bar{\Delta}}
\]

where \( \beta = k_1 / k_4 \), the ratio of linking speed to compilation speed.

Let us assume a system with:

\( \bar{\Delta} = 150 \) lines

\( k_1 = 5 \) compilations/100 lines = .05

\( np = 100 \) programs

\( \beta = 10 \) i.e. linking is 10 times faster than compiling

\( k_3 = 10 \) external references per program

\[
T_c = \frac{0.05 \times 150 \times 10 + 10 \times 10}{0.05 \times 10 \times 100 \times 150} = \frac{175}{7500} \approx 0.023
\]

Assuming \( \beta = 1 \)

\[
T_c = \frac{0.05 \times 150 \times 1 + 10 \times 10}{0.05 \times 1 \times 100 \times 150} = \frac{107.5}{750} \approx 0.143
\]
The above analysis is a severe simplification of the actual situation. The author knows of no formal analysis of this problem.

However, the prospective savings can be verified by the following example:

APS is a system consisting of some 600 modules totaling about 150,000 lines of source program in Fortran and Assembler.

The compilers involved averaged about 10,000 lines/hour on a 360/50, so it would have taken around 10-15 hours to recompile the complete system.

However, the whole system could be "linked" together from scratch in about 20 minutes.

In particular, this meant that recompilation of a particular program of say 1000 lines could be altered and a new version of the system obtained for about 30* minutes.

* In fact, the IBM linkage editor operates in such a way that such an operation took about 10 minutes.
2. SOURCE LANGUAGE FEATURES IN EPASS II

2.1 External References and Global Symbols

We must, if we are going to have a system split into several programs, provide an interworking mechanism. At some point in time, one program will transfer control to another.

EPASS II will use some of the conventions from FD 1818, and we describe these briefly below:

2.1.1 External References

Any symbol used as a label* in may infact, specify an address in another program. This is done by writing an

\[
\langle\text{External Reference} \rangle : = \langle\text{Programme} \rangle. \langle\text{label} \rangle
\]

where \( \langle\text{programme} \rangle \) is the name of a program containing the label \( \langle\text{label} \rangle \)

* Except in SETPC, RESERVE or on a BEGIN. Some restrictions may be necessary for ASSIGN.
EX 1

JSUBR ERRORPROC.IOERROR

causes a jump to the label IOERROR in the subroutine called ERRORPROC

EX 2

CMP (CALLINGDEVICE),DEVCLASS.FDR6
JEQ FDR6CALLED

causes a local variable (CALLINGDEVICE) to be compared with the value of the label FDR6 which is in a program called DEVCLASS, which might be

BEGIN DEVCLASS
GLOBAL FDR5, FDR6, FURT, FIRT

This program is a table of Device Types

FDR5: ASSIGN 1
FDR6: ASSIGN 2
FURT: ASSIGN 3
FIRT: ASSIGN 4
END DEVCLASS

External references will be allowed for any immediate constant or address.
2.1.2 Global Symbols

A global symbol is one which can be referenced as an external reference (see 2.1.1). An external reference does not have the desired effect unless there is a symbol corresponding to label in the program <programname> which is GLOBAL.

We introduce the pseudo instruction GLOBAL which has the syntax.

<global declaration>GLOBAL<label>

where <label> is some label defined within the program.

2.2 Effect of BEGIN - New Definition

The pseudo BEGIN will be interpreted differently in EPASS II.

2.2.1 Complete Program

If a loading address is specified and there are no external references then a loadable binary file will be produced by the EPASS II Assembler in addition to linker input file.

2.2.2 Program Name

A program will have a Name <programname> which shall be the symbol used in the BEGIN statement. This name is for use in external references and loader directives.

2.2.3 Default Global Symbol <programname>

By default, the programname is a global symbol whose value will be the loading address of the program.

This means that an external reference of the form <programname>, <programname> has as its value the loading address of the program <programname>.
3. **LINKER FUNCTIONS**

The linker will provide the following functions.

3.1 **Program Link List**

Link a specified list of programs from a specified list of files. Only those programs in the Program Link list are to be considered unless otherwise indicated by the Library List.

3.2 **Library List**

The Library List is a list of files in which programs referred to by external references within programs in the Link List are to be found. Note that files in the program Link List may also appear in the Library List.

3.3 **Specification of Program Loading Addresses**

The loading address of particular programs or groups of programs and there loading order may be specified.

3.4 **Specification of Start Address**

A start address may be specified to the Linker. If this is done, then that start address will be used and all others ignored. Alternatively, the first start address encountered by the linker in any program will be start address used.

**N.B.** The start address specified to the linker will be an external reference!
3.5 Linker Storage Map

A map of the storage will be produced when specified. This will provide a cross-reference mechanism to enable the program interworking to be clearly perceived.

3.6 Loadable Binary File

A Loadable Binary File will be produced in a specified file.

3.7 Usage of Symbol Tables Produced by an Existing Linker Run

The linker will be able to save the symbol table produced by a given run in a form suitable for linker input so that programs may be added externally to an existing store load.

This facility will also enable individual programs or parts of programs to be over-written without re-linking the whole package.

4. FORMAL SPECIFICATION OF LINKER (Preliminary Only)

4.1 A Load Specification

Loader directives are broken up into Load Specifications.

A Load Specification contains the following information (not necessarily in the order specified, and some optionally).

4.1.1 Output File specification for the resultant program and/or its symbol tables.
4.1.2 Load Address List
4.1.3 Program Link List
4.1.4 Library List
4.1.5 Start Address specification
4.1.6 A symbol table specification

<load specification> -> <symbol table file specification> -> <load address list> ->
<program link list> ->
4.2 Symbol Table File Specification

The name of the file containing the symbol table is specified as follows:

USE TABLES IN <filename>

4.3 Load Address List

<Load Address List> LOAD to <programme> AT to <addressconstant> <eos>

All symbols have obvious meanings except that addressconstant is a hexadecimal, decimal or octal constant.

4.4 Program Link List

The precise format of this is to be fixed.

<program link list> LINK <program list> IN <file list> <eos>

where

<program list> <programme>

and

<file list> <file specification>

The individual LINK directives are self contained and handled separately.
The programs in the programlist will be sought in the filelist and a list made of their unresolved external references and of their global symbols.

Note that a file may appear in more than one filelist.

4.5 Library List

Library lists may be specified in three ways:

4.5.1 RESOLVE* - AGAINST

It may be desirable to resolve the external references originating from specified programs against the programs in specified files.

EX 3 If the programs SYSTEM, APPLICATION and MAINTENANCE reference programs EXEC, SUPERVISOR, DEVICE and ERROR, and ERROREM which are in the files F1, F2 as well as PLIB, and the copies in F1 are needed we could write:

RESOLVE SYSTEM, APPLICATION, MAINTENANCE AGAINST F1

* This function may not be implemented.
4.5.2 USE IN

It may be desirable to use the copies of programs in particular file or files as those against which the external references are to be resolved.

\[ \text{<usage spec>} \rightarrow \text{USE} \rightarrow \text{programlist} \rightarrow \text{IN} \rightarrow \text{filelist} \rightarrow \text{<cos>} \]

4.5.3 LIBRARY

It will be necessary to specify the files in which the program satisfying any remaining external references are to be found. This is done with a

\[ \text{<library specification>} \rightarrow \text{LIBRARY} \rightarrow \text{programlist} \rightarrow \text{<cos>} \]

4.5.4 General Semantics

The order in which the library directives appear is immaterial. However, only those references not resolved by at the point where a particular directive is encountered can be resolved by that directive (or sub-directive).

DIRECTIVES ARE PROCESSED STRICTLY IN THE ORDER ENCOUNTERED.
4.6 Output File Specifications

4.6.1 Symbol Tables

The Symbol Tables are saved in the named file.

SAVE TABLES IN <filename>

The information saved is sufficient to enable another program to reference any global symbol in the load module created by this linking operation.

4.6.2 Program

The program is saved in the named file

SAVE PROGRAM IN <filename>

The program will be in a loadable binary format as needed by the binary loader.
5. DESIGN PROBLEMS

5.1 Introduction

Linkers present a series of design problems to the implementer. None of them are insurmountable, however, we are aiming at extremely high speed – so the correct choice of alternatives is important.

This project has limited manpower, so implementation must not yield problems not foreseen in the design.

The strictly sequential processing of commands is intended to simplify the implementation.

A number of features have been omitted – deliberately. In particular, the ability to "edit" a linked module is not provided – this is available in the IBM Linkage Editor, a truly wonderous beast. Overlay processing capability is not planned for an initial implementation.

5.2 Address Modes and Sizes

A major decision relates to operand Address modes and sizes.

5.2.1 Address Modes

The various addresses modes which will result in calculation to an actual value, must be identified and associated with a calculation rule.

The APN 162 has only two such address modes: PC relative and direct.

5.2.1.1 The PC relative address must be computed based upon the actual location of the address being resolved.

5.2.1.2 The direct address involves substitution of the actual address value.
3.2.2. Address Sizes and Location

It is possible for an address to have a limited width and odd starting position within the instruction.

The Linker must, therefore, be able to obtain this information for the particular usage of the address in question. This may be in addition to the mode.

5.3 Locating the Positions at which Address Usages Occur

The traditional method of finding the actual word in the program in which the external reference or address usage occurs is by linking them together.

If the space for the address is insufficient to permit a large enough link, then some other method is employed.

For example, if an address field occupies 8 bits, then it cannot possibly be used to link to any item further than 256 words (or bytes away). The precise method to be used has not yet been chosen.

5.4 Organization of Multiple Program Files

The organization the output from EPASS runs in which more than one program was processed is significant, since these will constitute possible libraries.

They must be organized in such a way that the required programs can be found without continuous scans of the file. Some form of directory is needed for each such file. Further, we are implying direct access files.

5.5 Allowing for Empty Programs

Example 2 in section 2 shows a program which produces no instructions. The design chosen must allow such programs.
5.6 Format of the Storage Map

The storage map is a major item in the design and its requirements will effect the overall design. In particular, the source line at which the GLOBAL symbol is defined and the lines at which it is referenced are of interest. This information should be provided if possible*.

5.7 Error Messages

Error messages should list the symbol referenced and the program from which the reference was made when an external reference cannot be found. In practice, this will be the last action taken. More specific errors, i.e., an external symbol which does not fit its address size, should produce a line reference*.

5.8 Internal Table Organization

The internal table organization is highly significant, since the combined symbol table can be arbitrarily large. The system should, therefore, be able to switch from a completely core-resident table to a virtual storage table which necessary. This will involve dynamic table re-organization.

* This may not be provided.
6.2 Handling of Various Operand Types

A final decision has not been made on the range of operand types to be allowed.

6.3 General Technique for Linking

Refer Section 4.1.

6.3.1 Load Address List

Firstly, the Load Address List will be used to construct an embryonic symbol table for later use. The name of each program and its start address will be held initially. A storage allocation map will be constructed also.

6.3.2 Program Link List

Each file in the Program Link Lists (there must be at least one entry for the PLL) will be opened and the specified programs found from the respective program Name Directories (PND's). The program length and supplied or implied start address will be checked against the storage allocation map to ensure that no overlap occurs.

The data in the Global Symbol Dictionary and External Reference Dictionary will be added to the symbol table. Each program will then be copied to an output file - as it is copied.

6.3.2.1 Any ER's for which a value exists in the symbol table will be correctly set.

6.3.2.2 Any External Reference useages for which a link-chain exists will be linked into that chain.
6.3.2.3 Absolute addresses will be computed for any operands which must be relocated.

N.B. If no start address was specified by a program, then the next available address is used.

6.3.2.4 Any ER's directed to a program which is not found will produce an error.

6.3.4 Library Processing

Each library directive is processed as it is encountered.

6.3.4.1 **RESOLVE AGAINST**

If a RESOLVE AGAINST directive is encountered, then each program named in the directive is located in the symbol table and the programs in its ERD are sought in the list of files specified in this directive. Action taken are as in 6.3.2 - 6.3.2.4. Error messages will be produced if the programs have already been resolved, but not if more than one copy of a program existed in the files.

6.3.4.2 **USE IN**

If a USE IN directive is encountered, then a check is made that these programs are referenced in the existing symbol table. If they are not, then an error message indicating fact is produced.

If the program is still to be loaded, i.e. has ERS pointing to it, then it is sought in the files specified in this directive. If found, the action taken is that in 6.3.2.2 - 6.3.2.4.

No error message is produced if it is not found.
All tables will then be cleared and the input stream examined for more data.

6.4 General Errors

The following general classes of errors will be logged.

6.4.1 Program not found
A program referenced was not found.

6.4.2 Unresolved External Reference
The label referenced was not in the program.

6.4.3 Program Over writes Existing Program
The program will not fit; it will not be linked.

6.4.4 Loop in ASSIGN statements
A loop is formed by ASSIGN statements.

6.4.5 Violation of Operand Useages
Operands are too big, too small etc.

6.5 Format of Loadable Output

The format of the loadable output will conform to the standard APN 162 loader format.

6.6 MAP

The precise map format is to be decided.
7. MODIFICATIONS TO EPASS I

The following modifications are necessary:

7.1 Syntax of Operands

Label references must permit ‹name›•‹label› in all places, subject to existing rules.

7.2 New Directive

The directive GLOBAL must be added.

7.3 Production of Linkable Files

The output from EPASS must be in the form described elsewhere in this document.

8. CONCLUSIONS

It will be seen that an extremely flexible system has been specified, which will meet all requirements except for overlay processing. We have not allowed for COMMON in the specifications, although this could easily be added.

Overlay processing is not included. This would involve a significantly different output file format and additional capabilities in the Linker.
APPENDIX 2

CODE AND RESULTS FROM HP2100 MODIFICATIONS
APPENDIX 2 - PART 1  Written Notes

A2.1.1  Explanation

This Appendix is split into two parts, these Written Notes and a
collection of Computer Listings.

A2.1.2  Index to the Listings

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Listings of the directory builder (DIRBL) and sample run.</td>
</tr>
<tr>
<td>2</td>
<td>Abbreviated listing of LMODD, the modified loader.</td>
</tr>
<tr>
<td>3</td>
<td>Listing of DEXEG, the modified version of the buffer management routines (Fortran).</td>
</tr>
<tr>
<td>4</td>
<td>Listing of DEXEA, the modified version of the buffer management routines (Assembler)</td>
</tr>
<tr>
<td>5</td>
<td>Trial Results - Timing tests</td>
</tr>
<tr>
<td>6</td>
<td>Complete listing of LMODD</td>
</tr>
<tr>
<td>7</td>
<td>Abbreviated listing of LMODD with diagnostics</td>
</tr>
<tr>
<td>8</td>
<td>LISOX, the modified formatless I/O routine</td>
</tr>
<tr>
<td>9</td>
<td>PRIHE, a simple heading lister</td>
</tr>
<tr>
<td>10</td>
<td>Listings of the compiler-calling programs FINCO, ASMCO.</td>
</tr>
</tbody>
</table>
A2.1.3 Notes on DIRBL

DIRBL consists of nine subroutines and a main program.

A2.1.3.1 Subroutine GELNC (File KDLNC)

Locates a user file in system area given its name.

Provides start address and length. Calls EXEC.

A2.1.3.2 Integer Function GERLT (File KBRLT)

Obtains the next record heading data from a relocatable deck.

Provides INDENT type, record length and number of entries. Writes the three word header to the directory (via PACKB) in all cases except for DBL records (nothing written) and for a NAM record when two words are written. Calls INR to read the subroutine library.

A2.1.3.3 Integer Function GESYM (File KBSYM)

Obtains the actual symbolic name and any associated data for a NAM,
ENT and EXT element. Sundry data for listing purposes returned.

Three words written to the directory for NAM and ENT elements; four for EXT via PACKB. INR called.
A2.1.3.4 Integer Function GEPRD (File KBPRD)

Extracts the four items Program Type, Main Program Length, Base Page Length, Segment Common Length from the NAM record. Writes the three words involved unaltered to the directory via PACKB. INR called.

A2.1.3.5 Subroutine POINT (File KBPOI)

POINT maintains the relative address of next word to be read from the subroutine library. It is a general purpose routine which makes use of a file descriptor array FIED. Relative sector, and word in sector are maintained. The new relative address is computed from the current in one of the following ways:

1) sector relative to current sector, absolute word.
2) absolute sector, absolute word
3) ignore sector, relative word to current address.

A2.1.3.6 Subroutine DIAL (File KDDIA)

Produces named diagnostic listings.

A2.1.3.7 Integer Function ICON (File KDICO)

Calculates the control word for exec calls. N.B. is not suitable for all possible control words since these are always positive.

A2.1.3.8 Subroutine PACKB (File KFPBO)

Packs variable numbers of words into a buffer and writes them when full. Can (in this version) create (open) a named file and write to it and close it (empty buffer). Contains unnecessary diagnostics associated with some bug which was never found. These are by-passed if IPAG.EQ.1.

A2.1.3.9 Program LILIS (File KFLIS)

This program builds the directory. It has two data cards. The first is the directory name in 3A2 format. The second is control data in I1, I6 format as follows:
Item 1 (I1) if 0 print all library data
   if 1 print routine names to statistics
   if 2 print only statistics
Item 2 (I6) is the number of routines to be processed
   if 0 all will be processed.

After reading the filename, PACKB creates the file. The program creates a field descriptor (FIED) and a set of pointers in common. These are used by INR and POINT for file control. The array used by POINT for file control. The array used by POINT (POIN) is initialized. GELNC is called to locate the library. At label 1000, GERLT is called to obtain the record type and length for the first (next) record. Switching on record type allows processing to continue as follows:

NAM Record (IDENT is 1) (Label 1000)

The absolute track/sector address for this program is calculated and written via PACKB to form the third word of the NAM record (refer note on GERLT). GESYM is called to extract the NAM symbol and other data, and a call to GEPRD completes the process. The NAM data is then printed (if required), and the remaining words in the NAM record copied to the directory.

GO TO 10000

ENT Record (IDENT IS 2) (Label 2000)

Successive ENT elements are fetched until the record is emptied (do loop on NENT). Data is listed if requested. Note that GESYM writes the data to the directory. When all elements of the record are processed,

GO TO 10000

DBL Record (IDENT is 3) (Label 3000)

DBL records contain relocatable program data and are not processed. POINT is used to skip the record.

GO TO 1000
EXT Record (IDENT is 4) (Label 4000)
EXT records are processed as for ENT records.

GO TO 10000

END Record (IDENT is 5) (Label 5000)
The last word in the END record is copied to the buffer, statistics are accumulated

GO TO 10000

The program is terminated by non-zero returns from the various subroutine calls which call INR which detects the end of file. A dummy END record with an ID of 6 is written to the directory so that the loader can detect the end of useful information.

A2.1.3.10 Subroutine INR (File KBINR)
This routine reads variable numbers of words from the library file, maintaining a valid address of the next word to be read, and permits random access via POINT.

A2.1.4 Modifications to Existing Programs
The modifications to the various programs are shown with a black line.

A2.1.5 The Program FINCO and ASMCO
These two programs make use of the facilities for saving the current program after it has passed a command string to the executive.
They perform basically as follows:
if filenamed is "/E" then stop;
if filenamed does not exist then print error and restart
else if qualifier is ",NEW" then
    if relocatable file exists print error and restart
else
    else purge relocatable file;
    compile filenamed and store its relocatable; restart

The programs greatly simplify the task of maintaining a suite of programs, and were used continuously throughout this project.
APPENDIX 3

REFERENCES
REFERENCES

GENERAL LITERATURE

GL1 PESSER & WHITE
"Linkers and Loaders" ACMCS Vol.4 No.3 pp.149-167

GL2 KNUUTH
"The Art of Computer Programming Vol.1 Fundamental Algorithms"

GL3 MC CARTHY, et al.
"The Linking Segment Subprogram Language and Linking Loader"
CACM Vol.6 No.7, July 1963.

GL4 BARRON
"Assemblers & Loaders"

GL5 CURTIS, A.R. and PYLE, I.C.
"A Proposed Target Language for Compilers on Atlas"
Com. J. pp.100-106 Vol.5

GL6 HEISING, W.P. & LARNER, R.A.
"A Semi-Automatic Storage Allocation System at Loading Time"

GL7 FLORES, I.

GL8 WEGNER, P.
"Programming Languages, Information Structures and Machine Organization"

GL9 GEAR, W.C.
"Computer Organization and Programming"

GL10 NOBLE, A.S. et al
"Design of an Integrated Programming and Operating System - Part III
The expanded Function of the Loader"

GL11 LANZANO, B.C.
"Loader Standardization for Overlay Programs"

GL12 MOORE, D.P.
"Library Loading with Alternate Routine Selection"
Letter to CACM Nov. 61, Vol.4, p.496.

GL13 MOCK, O.R.
"Logical Organization of the PACT I Compiler"

GL14 FRIED, F.C.
"An Ideal Computer Support Program and a Specific IBM System"
GL15  TSICHRITZIS, D.C. & BERNSKIN, P.A.
"Operating Systems"

GL16  WALLACE, C.S.
Informal discussions with author.

GL17  PANKHURST,
"Program Overlay Techniques"

GL18  JENSEN & WIRTH
"PASCAL User Manual and Report"

GL19  WIRTH, N.
"The Design of a PASCAL Compiler"

GL20  BIRTWISTLE, DAHL, MYHRHAUG & NYGAARD
"SIMULA BEGIN"
Student literature/Auerbach Publishers Inc. 1973, Second Printing.

GL21  PECK, Ed.
"Algol 68 Implementation"
Proc. of the IFIP Working Conference on Algol 68 Implementation,

GL22  "An Informal Introduction to Algol 68"

GL23  BOTTENBRUCH
"Structure and Use of Algol 60"

GL24  STOCKS & KRISHNASWAMY"
On A transportable Language for Mini computers"
ACM - SIGPLAN Notices Vol.11, No.4, April 1976, "Proceedings
of the ACM SIGMINI/SIGPLAN Interface Meeting on Programming
Systems in the Small Processor Environment" pp.138-143.

GL25  FLORES
"Computer Organization"

GL26  HARDGRAVE, W.T.
"Positional Versus Keyword Parameter Communication in Programming
Languages" SIGPLAN Notices Vol.11, No.5, May 1976, pp.54-58.

GL27  KOSTER, C.H.A.
"Beating the Global"
M.O.L. Bulletin No.64, First Meeting of the Working Group on Machines
Oriented Languages, May 1974, IFIP WG 2.4 pp.109-115.

GL28  SCHUMAN, S.A.
"Toward Modular Programming In High-Level Languages"
M.O.L. Bulletin No.64, (See GL27) pp.117-128.
RR1  SEVERANCE
"Identifier Search Mechanisms"
ACM-CS Vol.6, No.3 pp.175-194.

RR2  NIEVERGELT
"Binary Search Trees and File Organization"
ACM-CS Vol.6, No.3, pp.195-207

RR3  KNUTH
"The Art of Computer Programming Vol.3 Sorting and Searching"

RR4  LEFKOVITZ
"File Structures for On-line Systems"
Hayden Book Co. 1969.

RR5  MONTGOMERY, A.Y.
"Algorithms and Performance Evaluation of a New Type of Random
Access File Organization"
ACJ Vol.6, No.1, March 1976, pp.3-11

RR6  MC DONELL, K.J. & MONTGOMERY, A.Y.
"The Design of Indexed Sequential Files"

RR7  MONTGOMERY, A.Y. & WALLACE, C.S.
"Evaluation and Design of Random Access Files"
ACS pp.142-150

RR8  MONTGOMERY, A.Y.
"Evaluation and Design of Tree Structured Files"
Lecture Notes, Department of Computer Science, Monash University.

RR9  KNUTH
"The Art of Computer Programming, Vol.1 Fundamental Algorithms"
Addison-Wesley, World Student Series, Ed 1972.

RR10  BERZTIS, A.T.
"Data Structures, Theory and Practice"

RR11  GRIES, D.
"Compiler Construction for Digital Computers"

RR12  HILL, U.
"Special Run-Time Organization Techniques for Algol 68".
in "Compiler Construction, An Advanced Course" Eds. Gross and
Hartmanis, pp.222-252, Springer-Verlag, Lecture Notes in Computer
Science No.21, 1974.
MANUALS AND DOCUMENTATION

MD1  "MSOS Loader"
     CDC 3200 Operating System Maintenance Documentation pp.2-240, 2-279.

MD2  "Program Binder, B6700"
     Burroughs Corp. Form No. 5000045, Nov. 1971.

MD3  "Subroutines Common to Exec Subtasks"
     CDC 3200 Operating System Maintenance Documentation, pp.2-379.

MD4  "IBM System/360 Operating System Linkage Editor and Loader"
     IBM Systems Reference Library File No.S360-31, Order No. GC28-6538-8

MD5  "Hewlett Packard 2100A Moving Head Disc Operating System Users' Guide"
     Department of Computer Science, Monash University, February 1975.

MD6  "A Pocket Guide to the Hewlett-Packard 2100A Computer"
     Hewlett Packard

MD7  Digital Equipment Corporation.
     "DECSYSTEM 10 Link-10 Programmer's Reference Manual"

MD8  CDC MSOS Loader

MD9  HP System Manual

MD10 "IBM System/360 Operating System Supervisor and Data Management
      Macro Instructions"

MD11 REED, K.
      "Preliminary Specification for a Linkage"
      Document P/Yx -PD2283, L.M. Ericsson Pty. Ltd. (Australia), 11/6/75,
      (See Appendices)

MD12 REED, K. & WARD, C.F.
      "Source Language Specifications"
      Document P/Yx - PD1818, L.M. Ericsson Pty. Ltd., (Australia) 30/10/73.

MD13 "B6700 Workflow Management User's Guide"
      Burroughs Corporation, 1973, Form 5000714.

MD14 "IBM System/360 Operating System: Job Control Language Reference
      Manual" Order No. GC28-6704

MD15 DECll Linker (RTll)

MD16 "B6700/B7700 ALGOL Language Reference Manual"
      Burroughs Corporation, Form 5000649, 1974.

MD17 "B6700/B7700 COBOL Reference Manual"
      Burroughs Corporation, Form 5000656-001 Rev.1 April, 1974.

MD18 IBM Systems Reference Library,
      IBM System 360 Operating System.
MD19  IBM System/360 Operating System
     IBM Systems Reference Library, Form C28-6594
     "PL/I(P) Programmer's Guide"

MD20  DECSYSTEM10 "ALGOL"
     Form DEC-10-LALMA-A-D
     Digital Equipment.

MD21  "SIMULA Users' Manual" Preliminary Edition
     Burroughs Corporation, 1972, No.AA181435.

MD22  VARIAN 73, System Handbook
     Vairan Data Machines, Calif. 1972, Form No.98A9906010

MD23  PDP11/20/15/120 Processor Handbook
     Digital Equipment Corp. 1971.

MD24  DEC SYSTEM-10 System Reference Manual, Order No. DEC-10-HGAD-D

MD25  "IBM360 Principles of Operation"
     IBM Systems Reference Library Form AA22-6821-5, File No.S360-01

MD26  "Programmer's Reference Manual, ECLIPSE LINE OF COMPUTERS"
     015-000024 Rev.4.

MD27  REED, K.
     "Jumptables - A different Approach"
     L.M. Ericsson Pty. Ltd.
     X/Mx186 1970.

MD28  "PDP11/70 Processor Handbook"
     Digital Equipment Corp. 1975.

MD29  IBM System/360 Operating System
     FORTRAN IV (G & H) Programmer's Guide
     IBM Systems Reference Library File No. S360-25, Order No. GL28-6817-3

MD30  B6700/B7700 BINDER Reference Manual (Relative to Mark 11.7
     Release) Burroughs Corporation 5000912, 9-75.