Burroughs

Reference Manual

B 20 Systems Linker/Librarian

Priced Item
Printed in U.S.A.
June 1982

1148681

Burroughs

Reference
Manual

B 20 Systems Linker/Librarian

Priced Item
Printed in U.S.A.
June 1982

1148681

This edition also includes the information released under the following:

PCN 1148681-001 (January, 1983)
PCN 1148681-002 (August, 1983)
PCN 1148681-003 (April, 1984)

LIST OF EFFECTIVE PAGES

Page	Issue
iii	PCN-003
iv	PCN-003
v thru vii	Original
viii	Blank
1-1 thru 1-4	Original
1-5	PCN-002
1-6 thru 1-10	Original
1-11	PCN-003
1-12 thru 1-15	Original
1-16	Blank
2-1 thru 2-6	Original
A-1	Original
A-2	Blank
1 thru 3	Original
4	Blank

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

INTRODUCTION

This manual provides descriptive and operational information for the Linker/Librarian utility used in program development applications for the B 20 Micro-Computer Systems. The information is presented in two sections: Section 1 for the Linker utility, and Section 2 for the Librarian utility. A glossary of terms is provided in Appendix A.

The following technical manuals are referred to for additional information:

B 20 System Executive Reference Manual, form 1144474

B 20 Operating System Reference Manual, form 1148657

B 20 System Programmers Guide, Part 1, form 1148699

B 20 System Programmers Guide, Part 2, form 1144466

B 20 Systems Debugger Reference Manual, form 1148665

B 20 Systems Forms Reference Manual, form 1148715

SECTION 1
LINKER

INTRODUCTION

The Linker is a program development utility. It combines object modules (files produced by compilers and assemblers) into run files. Run files are memory images of tasks (in ready-to-run form) linked into the standard format required by the Operating System loader.

The Linker has these features:

- It resolves references from one object module to variables and entry points of other object modules.
- It builds a run file that can be efficiently loaded. The run file that is constructed consists of a memory image plus some directory and relocation data. It is organized so that the Operating System can, in most cases, load it with a single disk access and data transfer.
- It does not require specification of the eventual memory address of the task. Instead, the Linker computes and places into the run file information that the Operating System loader uses to relocate the task, when loaded, to any desired memory location. In this way, a single run file can be used with various memory configurations or as one of the tasks of a multitask application system.
- It can search through libraries of object modules and select exactly those modules required by a particular application.
- It can construct run files containing overlays for use with the virtual code segment management facility.

Two-Pass Linker

This is a two-pass linker. On the first pass, the Linker reads all object modules, extracting external and public symbol information, and builds a symbol table. It looks at this symbol table to see if there are unresolved external references. If there are, it repeatedly searches the list of libraries, specified when the Linker is invoked, for object modules whose public symbols resolve the external references.

On the second pass, the Linker assigns relative addresses, relocating as necessary, to all data in all object modules, and links the object modules, constructing a run file ready for the Operating System loader.

INVOKING THE LINKER

To invoke the Linker from the Executive, type "Link" in the command field of the command form. (See the B 20 System Executive Reference Manual, form 1144474.) Then press the RETURN key. The form illustrated below then appears.

```
Link
  Object modules             ______________________
  Run file                   ______________________
  [List file]                ______________________
  [Publics?]                 ______________________
  [Line numbers?]            ______________________
  [Stack size]               ______________________
  [Max memory array size]    ______________________
  [Min memory array size]    ______________________
  [System build?]            ______________________
  [Version]                  ______________________
  [Libraries]                ______________________
  [DS allocation?]           ______________________
  [Symbol file]              ______________________
```

Fill in the various fields. You are normally concerned with only a few fields: "Object modules", "Run file", and "[List file]". All fields after the second ("Run file") are optional. You can default optional fields by leaving them blank. You will usually default all fields after the third.

FIELD DESCRIPTIONS

Object Modules

Fill in the "Object modules" field with a list of the names of one or more object module or library files. Separate the names with spaces, not commas. The Linker combines these object files to form a run file.

Example

To build a run file from the three object modules A.obj, B.obj, and C.obj, fill in the "Object modules" field of the Linker form this way:

```
Object modules A.obj B.obj C.obj
```

The Linker combines the three modules to form a run file.

You can mix the specification of ordinary object module files and object modules to be explicitly extracted from libraries. Specify library extraction with this form:

```
LibraryName (module1 module2 ...)
```

where module1, module2, etc., are the names of object modules to be extracted. (Note that these module names are separated by spaces, not commas.)

Example

Assume that Z.lib contains the object modules V, W, and X. To build a run file consisting of object modules A, B, W, X, and C, fill in the "Object modules" field of the Linker form this way:

```
Object modules A.obj B.obj Z.lib(W X) C.obj
```

The run file is the same that results if the original W.obj and X.obj are specified in the "Object modules" field as:

```
Object modules A.obj B.obj W.obj X.obj C.obj
```

Virtual Code Segment Management

You can construct a task containing code that is not permanently memory-resident. The virtual code segment management facility allows you to divide your task into resident and nonresident portions. (See the B 20 Operating System Reference Manual, form 1148657.) The resident portion includes data and code segments; the nonresident portion includes only code segments. When linking, you can specify one or more overlays. An overlay is a code segment, made up of the code from one or more object modules. An overlay is loaded into memory as a unit.

To construct a run file using overlays, append "/O" to the name of the first module in each overlay.

Example

To build a run file consisting of a resident portion with the code from A, and a nonresident portion, consisting of two overlays, one with the code from B, W, and X, and a second with the code from D, fill in the "Object modules" field of the Linker form this way:

```
Object modules A.obj B.obj/O Z.lib(W X) D.obj/O
```

Run File

Fill in the "Run file" field with the name of the run file to be built.

[List File]

Fill in the "[List file]" field with the name of the list file.

The default is to derive the name of the list file from the name of the run file. If you do not specify a list file, a default list file is chosen as follows: the run file name is treated as a character string, any final suffix beginning with the character period (.) is removed, and the characters ".map" added. The result is the name of the list file. For example, if the run file is:

```
  Prog.run
```

then the default list file is:

```
  Prog.map
```

If the run file is:

```
  [Dev]<Jones>Main
```

then the default list file is:

```
  [Dev]<Jones>Main.map
```

If you default the remaining ten fields of the Linker command form, the Linker generates a short list file that contains an entry for each module or segment and shows the relative address and length of the module or segment in the memory image.

Figure 1-1 shows a sample list file. The starting addresses are offsets, not absolute addresses. The offsets are relative to the base memory address, determined at run time, at which the Operating System loads the run file.

```
Start    Stop     Length  Name            Class
00000h   00639h   063Ah   BNDTRN  CODE    CODE
0063Ah   00C8Bh   0652h   BNDTP1  CODE    CODE
00CBCh   01890h   0C05h   BNDMAP  CODE    CODE
01892h   01CD6h   0445h   BNDSYM  CODE    CODE
01CD8h   02565h   088Eh   BNDTP2  CODE    CODE
02566h   0299Fh   043Ah   BNDOUT  CODE    CODE
029A0h   02E0Dh   046Eh   BNDUTL  CODE    CODE
02E0Eh   034BDh   06B0h   BNDPRI  CODE    CODE
034BEh   03930h   0473h   BNDPAR  CODE    CODE
03932h   04136h   0805h   CONST           CONST
04138h   04572h   043Bh   DATA            DATA
04574h   04E1Ch   08A9h   STACK           STACK
04E1Eh   04E1Eh   0000h   MEMORY          MEMORY
```

Figure 1-1. Sample List File

[Publics?]

Fill in the "[Publics?]" field with YES to include the values of all public symbols in the listing, sorted first alphabetically, and then numerically by value.

The default is NO; that is, responding with NO and entering no response both produce no listing of public symbols.

Figure 1-2 is an excerpt from such a list file, with the values of all public symbols sorted first alphabetically, and then numerically.

The Address column contains the notation XXXX:YYYYh, where _h_ means hexadecimal; this is the standard processor segment-plus-offset addressing structure. For more details about this, see the processor.

The Overlay column contains "Res" if the symbol is in the resident portion of your task, an integer (_n_) if it is in the _n_th overlay, and "Abs" if it is absolute. An absolute symbol is one with a specified place in memory (for example, an address within the Operating System).

```
Publics by name                Address     Overlay

BSRUNFILE                      076B:3630h    Res
BSVIDCLEARMARK                 0593:0682h    Res
BSVIDEO                        076B:36CCh    Res
BSVIDMARK                      0593:0604h    Res
BSVIDTURNOFFCURSOR             0593:06E5h    Res
BSVIDTURNONCURSOR              0593:06A4h    Res
CBREC                          076B:376Ch    Res

Publics by value               Address     Overlay

BSVIDMARK                      0593:0604h    Res
BSVIDCLEARMARK                 0593:0682h    Res
BSVIDTURNONCURSOR              0593:06A4h    Res
BSVIDTURNOFFCURSOR             0593:06E5h    Res
BSRUNFILE                      076B:3630h    Res
BSVIDEO                        076B:36CCh    Res
CBREC                          076B:376Ch    Res
```

Figure 1-2. Sample Symbol Table Listing

[Line Numbers?]

Fill in the "[Line numbers?]" field with YES to include in the list file a list of line numbers and addresses of all source statements in your task. This list of line numbers comes after the listing of public symbols. Not all compilers produce object modules containing line number information.

The default is NO; that is, responding with NO and entering no response both produce no list of line numbers and addresses in the list file.

Figure 1-3 is an excerpt from a list file with line numbers.

```
105 0000:0000h  106 0000:0003h  107 0000:0023h
108 0000:00B2h  109 0000:0092h  110 0000:00AFh
111 0000:00EBh  112 0000:00EDh  113 0000:00F2h
114 0000:0114h  115 0000:011Eh  116 0000:0123h
```

Figure 1-3. Sample List File with Line Numbers

[Stack Size]

Fill in the "[Stack size]" field to change the stack size from the compiler's estimate.

All compilers produce information in object modules from which the Linker can compute the size of the required stack segment. This information is usually conservative (that is, it calls for a stack that is typically larger than the actual requirements). However, in rare cases, the compiler can supply information that causes the Linker to undercompute the required stack size. An example is a task with many recursive procedures.

The default is the compiler's estimate of the correct stack size.

[Max Memory Array Size]
[Min Memory Array Size]

Fill in the "[Max memory array size]" and "[Min memory array size]" fields to leave data space above the highest memory address of the task. The specifications must be in decimal notation.

The defaults are 0 for both of these fields.

If you specify too large a minimum (that is, if you don't leave enough room for the task), then an error code and error message are output when the Operating System fails to load the task. If you want the task to always load low, that is, with maximum data space above the task, set the minimum to 0 and the maximum to 1000000.

Figure 1-4 shows the normal memory configuration when a run file is loaded.

Figure 1-5 shows the memory configuration when a run file is loaded and the memory array size was specified.

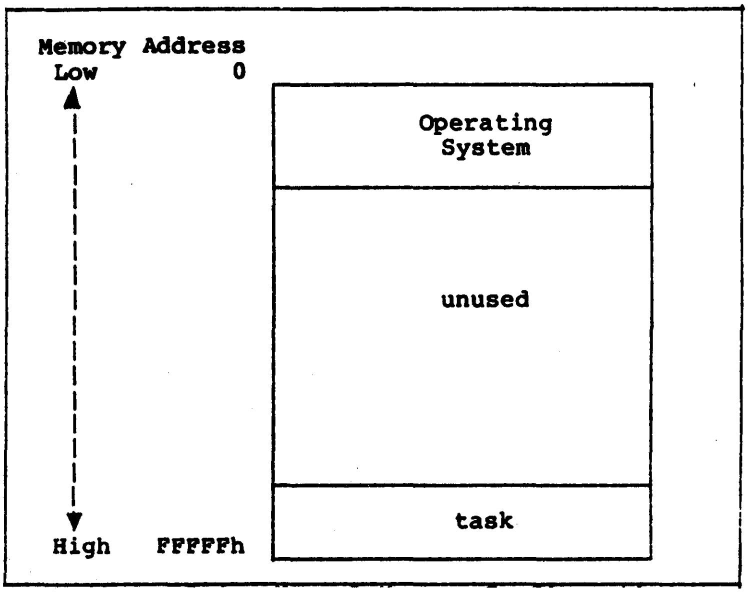

Figure 1-4. Normal Memory Configuration

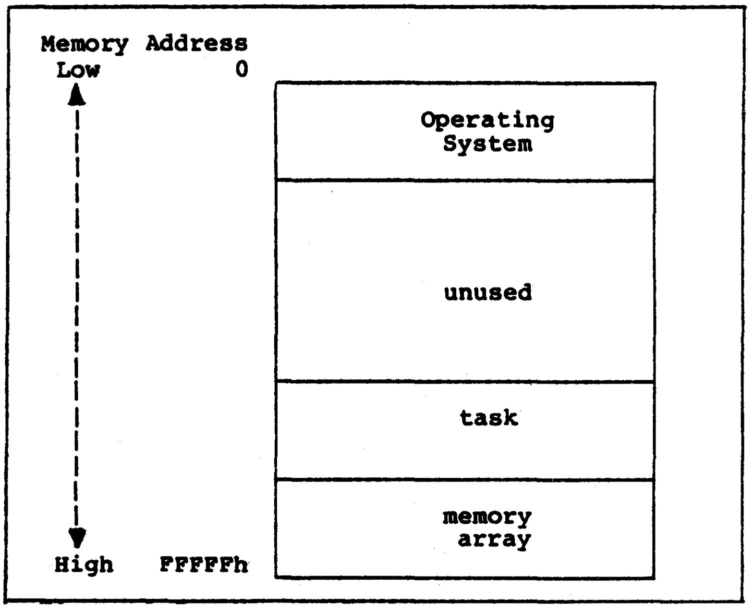

Figure 1-5. Memory Configuration with Memory Array Size Specified

[System Build?]

Fill in the " [System build?] " field with YES only to build special versions of the Burroughs Operating System. Do this only by carefully following the system build procedure described in the B 20 System Programmers and Assembler Reference Manual (Part 1), form 1148699.

The default is NO; that is, responding with NO and entering no response both produce no system build.

[Version]

Fill in the "[Version]" field to add a version specification (in the form of an alphanumeric string) to the header of the run file. (As usual in command forms, if this parameter has embedded spaces, then the parameter must be surrounded with single quote characters.) The string, preceded by the characters "VER " (the letters V, E, R, and a space), is placed in the first sector of the run file. Also, the public variable sbVerRun is automatically defined in the run file: it consists of the string preceded by a single byte containing the length of that string.

The default is none; that is, entering no response produces no version specification.

For example, if the string is 1.0, then the run file header sector contains the text "VER 1.0" and the variable sbVerRun is defined as the following four bytes: the number 3, and the ASCII characters "1", ".", and "0".

[Libraries]

Fill in the "[Libraries]" field with a list of library files. Separate the names with spaces, not commas. The library files must have been created by the Librarian utility.

The Linker searches, and automatically selects object modules from, a list of library files to satisfy unresolved external references. The Linker then treats the selected object modules as if they had been specified in the "Object modules" field, that is, they are linked with the resident portion of a task that uses overlays. To link object modules obtained from libraries into overlays, the explicit library extraction facility discussed above in "Object modules" must be used.

The Linker always searches the file [Sys]<Sys>CTOS.LIB, if it exists, for unresolved external references.

[DS Allocation?]

Fill in the "[DS allocation?]" field with YES (the default) to offset all references to group DGroup to minimize the run-time value of DS (the data segment register). This field is meaningful only with tasks that use a single value in DS during their entire execution, and include the group DGroup with DS equal to DGroup.

Responding with NO produces no offsetting of DGroup references.

(Object module procedures and tasks produced by the Pascal compiler use a single value in DS during their entire execution, and include the group DGroup with DS equal to DGroup; indeed, this feature must be used for linking Pascal tasks that make use of the Pascal heap.) If a task using this feature is loaded at the high end of memory, the space below the task can be conveniently used as a dynamically allocatable area containing data referenced relative to DS.

[Symbol File]

Fill in the "[Symbol file]" field with the name of a file to which the Linker writes a symbol table of the run file. (See the B 20 Systems Debugger Reference Manual, form 1148665, for an explanation of the use of this file.)

The default is to derive the name of the symbol file from the name of the run file. That is, if you do not specify a symbol file, a default symbol file is chosen as follows: the run file name is treated as a character string, any final suffix beginning with the character period (.) is removed, and the characters ".sym" added. The result is the name of the symbol file. For example, if the run file is:

```
Prog.run
```

then the default symbol file is:

```
Prog.sym
```

If the run file is:

```
[Dev]<Jones>Main
```

then the default symbol file is:

```
[Dev]<Jones>Main.sym
```

To avoid creating a symbol file, fill in this field with [NUL].

LIMITS

There may be at most 1024 public symbols and 256 segments in a run file, and 256 PUBLICs and 256 EXTERNs in a module.

ERROR MESSAGES

At the end of the list file, the Linker issues error messages. An _unresolved external reference_ is a public symbol that is used by some object module, but not defined by any of the modules being linked. If unresolved external references still exist after linking, the Linker issues the nonfatal error message, "Unresolved externals:", and lists them in this format:

```
nnn  in [Sys]<Sys>Module.obj
```

where _nnn_ is the name of the unresolved external reference.

ARRANGING MEMORY

An object module may contain any (or all) of the following: code, constants, variable data. The Linker arranges the contents of a set of object modules into a memory image, typically with all code together, all constants together, and all variable data together. (This arrangement makes optimal use of the addressing structures of the processor.) Although the Linker produces such arrangements automatically, the programmer will occasionally want to exercise explicit control. This section explains the concept involved and the facilities used to arrange memory.

Units of Relocation and Linkage

The fundamental units of relocation and linkage are segment elements, linker segments, class names, and groups.

Segment Elements

An object module is a sequence of segment elements. Each segment element has a segment name. An object module might consist of segment elements whose names are B, C, and D.

Linker Segments

The Linker combines all segment elements with the same segment name from all object modules into a single entity called a linker segment. A linker segment forms a contiguous block of memory in the run-time memory image of the task. For example, you might use the Linker to link these two object modules:

```
object module 1
  containing segment elements B, C, D

object module 2
  containing segment elements C, D, E
```

Linkage produces these four linker segments:

```
linker segment B consisting of element B1
linker segment C consisting of elements C1, C2
linker segment D consisting of elements D1, D2
linker segment E consisting of element E2
```

(In each of these cases, xi denotes the segment element x in module i.)

Class Names

The ordering of the various linker segments is determined by class names. (A class name is an arbitrary symbol used to designate a class.) All the linker segments with a common class name and segment name go together in memory. For example, if B1, D1, and E2 have class names "Red," while C1 has a class name "Blue," then the ordering of linker segments in memory is:

B, D, E, C

If you look inside the linker segments, you see that the segment elements are arranged in this order:

B1, D1, D2, E2, C1, C2

(If two segment elements have different class names, then they are considered unrelated for purposes of these algorithms, even though they have the same segment name.)

As you can see from this, segment names and class names together determine the ordering of segment elements in the final memory image.

Groups

The next step for the Linker is to establish how hardware segment registers address these segment elements at run time.

A group is a named collection of linker segments that is addressed at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

Several linker segments may be combined into a group. For example, if B and C were combined into a group, then a single hardware segment register could be used to address segment elements B1, C1, and C2.

Naming

Segment, class, and group names may be assigned explicitly in assembler modules using appropriate assembler directives (as described in the B 20 System Programmers and Assembler Reference Manual (Part 2), form 1144466). Most compiled languages assign these names automatically. (See the individual language manuals for details.)

Alignment and Combination

Segments have alignment and combination attributes. In assembly language programs, these attributes are assigned explicitly using appropriate assembler directives. (See the B 20 System Programmers and Assembler Reference Manual (Part 2), form 1144466 for details.) Most compiled languages assign these attributes automatically. (See the individual language manuals for details.) A segment can have one of several alignment attributes. These are:

- byte (a segment that can be located at any address),
- word (a segment that can be located at an address that is a multiple of two), and
- paragraph (a segment that can only be located at an address that is a multiple of 16).

The Linker packs segments containing data and code end-to-end. Alignment characteristics can cause a gap between the segments. The Linker adjusts the relative addresses in the segments accordingly.

Segments with the combine type STACK are a special case. When stack segments are combined, they are overlaid (with high addresses superimposed), but their lengths are added together. When the Linker has combined all stack segments, it forces the total length of the aggregate stack segment to a multiple of 16 bytes.

Segments with the combine type COMMON are also special. When COMMON segments are combined, they are overlaid (with low addresses superimposed), and the length is that of the largest element.

Compilers construct stack segments automatically. However, if your entire program is written in assembly language, you have to define an explicit stack segment. See the B 20 System Programmers and Assembler Reference Manual (Part 2), form 1144466 for details.

SECTION 2
LIBRARIAN

INTRODUCTION

The Librarian is a program development utility that creates and maintains libraries of object modules. A library has these uses:

- o A library can be a parameter in the "[Libraries]" field of the Linker command form, to specify that the Linker should search the library for object modules that satisfy unresolved external references. (See the Linker section of this Manual for details.)

- o A library is a convenient unit in which to collect several object modules and distribute them as a single file, so that the extraction facility described below, and also available in the Linker, can be used to extract specific modules from the unit.

- o A library is a convenient unit in which to collect several forms created with the Forms Editor. (See the B 20 Systems Forms Reference Manual, form 1148715.)

In the first use, above, of the Librarian, you do not have to know the names of the object modules composing a library, since the Linker's library search algorithm automatically selects from the library exactly the required modules. In the second and third uses, above, the desired object module or form name must be specified to extract it from the library.

The Librarian builds or manipulates libraries in these ways:

- o The Librarian builds a new library, when given the name of a new library file and the object modules that are to compose it.

- o The Librarian modifies an existing library, when given the names of object modules to be added to or deleted from it. (This includes the case in which a module in a library is to be replaced by a new module with the same name.)

- The Librarian extracts from a library one or more object modules, when given the names of the desired object module files.
- The Librarian produces a sorted cross-reference listing of the object modules and public symbols in the library.

INVOKING THE LIBRARIAN

To invoke the Librarian from the Executive, type "Librarian" (or as many letters as are required from the word "Librarian" to make the command unique; see the B 20 System Executive Reference Manual, form 1144474) in the command field of the command form Press RETURN. The form illustrated below then appears.

```
Librarian
  Library file              ____________________
  [Files to add]            ____________________
  [Modules to delete]       ____________________
  [Modules to extract]      ____________________
  [Cross-reference file]    ____________________
  [Suppress confirmation?]  ____________________
```

Fill in the various fields. You can default optional fields, those listed above in square brackets, by leaving them blank. All fields after the first are optional.

FIELD DESCRIPTIONS

Library File

Fill in the "Library file" field with the file name of the object module library (typically of the form LibraryName.lib).

If the specified file already exists, it is the starting point for any library to be built. The contents of the file are preserved intact in a file whose name is the original name plus the suffix "-old". However, if a cross-reference listing is requested but no modifications are made, then the input library is not modified and no "-old" file is generated. If modifications are requested, then the updated library is named as specified by "Library file".

If the specified file does not exist, you are prompted to confirm the creation of a new library file. You may suppress this request for confirmation by filling in YES in the field "[Suppress confirmation?]".

[Files to Add]

Fill in the "[Files to add]" field with a list of files containing object modules to be added to the library. Separate the names with spaces, not commas.

The default is none; that is, entering no response produces no adding of files.

You are prompted for confirmation if an object module that is to be added has the same name as an object module already in the library. If you confirm the replacement, the file containing the module with the same name replaces the existing object module. Since the Linker must sometimes search a library for a module defining a given public symbol, it is unusual to create a library in which two object modules define the same public symbol. (This kind of duplicate definition might reasonably occur in a library intended just as a convenient unit in which to collect object modules and not for automatic search.)

The name of the added module within the library is derived from the name of the added object file. All leading volume, directory, and subdirectory specifications are dropped. Any final extension beginning with a period is dropped. For example, if the file name is [Sys]<Jones>-Sort.obj, then the module name is Sort, and, if the file name is <Jones>Working>Sort, then the module name is Sort.

You are also prompted for confirmation if a public symbol declared in a module that is to be added conflicts with a public symbol already in the library. If you confirm the duplication, the module containing the duplicate definition is added, but the public symbols (both old and new) are removed from the index of symbols searched by the Linker. You may suppress these requests for confirmation by filling in YES in the field "[Suppress confirmation?]".

[Modules to Delete]

Fill in the "[Modules to delete]" field with a list of modules to be deleted from the library. Separate the names with spaces, not commas.

The default is none; that is, entering no response produces no deleting of modules.

[Modules to Extract]

Fill in the "[Modules to extract]" field with a specification of the object modules of an existing library that are to be extracted to form individual object module files. Separate the names with spaces, not commas. The specification is a list of entries of either the form:

```
ModuleName
```

or:

```
FileName(ModuleName)
```

If the first form is used, files containing the specified object modules are created with names of the form ModuleName.obj. If the second form is used, the file name may be specified explicitly.

When the Librarian is used to modify a library, which is the most common occurrence, the "[Modules to extract]" field is not used.

The default is none; that is, entering no response produces no extraction of modules.

[Cross-Reference File]

Fill in the " Cross-reference file " field with the name of a file to which the Librarian is to write a cross-reference listing of public symbols and object module names. A cross-reference listing has two parts:

o The first part lists public symbols in alphabetical order and, for each public symbol, the name of the module that defines it.

o The second part lists module names in alphabetical order and, for each module, the names of the public symbols it defines.

The default is none; that is, entering no response produces no cross-reference listing.

[Suppress Confirmation?]

Fill in the " Suppress confirmation? " field with YES if you do not wish prompts for confirmation when creating new library files (with the "Library file" field) or replacing existing object modules (with the " Files to add " field).

The default is NO; that is, responding with NO and entering no response both cause the Librarian to issue the prompts for confirmation.

SAMPLE LISTING

Figure 2-1 shows a sample listing produced using the "[Cross-reference file]" option.

```
    Burroughs Librarian, Version 2.0

COMPACTDATETIME.............CMPDT   EXPANDDATETIME........EXPDT
FILLFRAME.....................VAM   POSFRAMECURSOR........VAM
PUTFRAMEATTRS.................VAM   PUTFRAMECHARS.........VAM
QUERYFRAMECHAR................VAM   RESETFRAM.............VAM
SCROLLFRAM....................VAM

CMPDT (Length 0177h bytes)

   COMPACTDATETIME

EXPDT (Length 014Ch bytes)

   EXPANDDATETIME

VAM (Length 09B8h bytes)

   FILLFRAME          POSFRAMECURSOR      PUTFRAMEATTRS
   PUTFRAMECHARS      QUERYFRAMECHAR      RESETFRAM
   SCROLLFRAM
```

Figure 2-1. Sample Listing

APPENDIX A
GLOSSARY

Absolute. An absolute symbol is a symbol that has a specified place in memory (as, for example, an address within the Operating System).

Class Name. A class name is an arbitrary symbol used to designate a class.

Group. A group is a named collection of linker segments that is addressed at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

Librarian. The Librarian is a program development utility that creates and maintains libraries of object modules. The Linker can search automatically in such libraries to select just those object modules referred to by a program.

Linker. The Linker is a program development utility that combines object modules (files produced by compilers and assemblers) into run files.

Linker Segment. A linker segment is a single entity consisting of all segment elements with the same segment name.

Object Module. An object module is the result of a single compilation or assembly. A single object module is contained in an object module file (.obj), while many object modules can be contained in a library file (.lib).

Overlay. An overlay is a code segment, made up of the code from one or more object modules. An overlay is loaded into memory as a unit and is not permanently memory-resident.

Run File. A run file is a memory image of a task (in ready-to-run form) linked into the standard format required by the Operating System loader.

Segment Element. A segment element is a section of an object module. Each segment element has a segment name.

Unresolved External Reference. An unresolved external reference is a public symbol that is used by some module, but not defined by any of the modules being linked.

INDEX

Documentation Evaluation Form

Title: B 20 Systems Linker/Librarian Reference Manual

Form No: 1148681

Date: June, 1982

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Comment/Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error ☐ Other

Comments:

From:

Name

Title

Company

Address

Phone Number Date

Remove form and mail to:

Burroughs Corporation
Corporate Documentation
Planning, East
209 W. Lancaster Ave.
Paoli, PA 19301, U.S.A.