S-1 INTERMEDIATE LOADER FORMAT
and
S-1 LINKER

Arthur Keller and Gio Wiederhold

TECHNICAL NOTE NO. 147

November 1978

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

This work has been supported by the Department of the Navy via ONR Order Numbers N00014-77-F-0023 and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which is operated for the U.S. Department of Energy under Contract No. W-7405-Eng-48) and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation). This work was also partially supported by a National Science Foundation Graduate Fellowship. This work has been performed under Contract No. LLL P09083403, Principal Investigator, Professor Gio Wiederhold.

S-1 INTERMEDIATE LOADER FORMAT
and
S-1 LINKER

Arthur Keller and Gio Wiederhold

TECHNICAL NOTE NO. 147

November 1978

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

ABSTRACT

The loader format for the S-1 and the Linker for the S-1 using the format are described. The loader format was designed to be transportable, easily read by humans, and editable on available text editors. For these reasons it uses only a minimal subset of the printable ASCII character set. It is used for both the linker input and output, so that linking can proceed in stages.

The Linker is part of the S-1 programming system. It was designed to support linking modules from compilers of different languages. In order to make it transportable, it was written in PASCAL. The output can be relocatable or absolute, but it is usually absolute.

INDEX TERMS: Loader, linker, S-1, PASCAL, transportable, module, object code, relocation

Arthur Keller
Gio Wiederhold

27 November 1978

Objective

Output Files from Compilers and Assemblers which support the S-1 have to be transportable to the simulator and to the S-1 system for testing purposes. Limitations are imposed by limited wordlength and limited input/output capabilities of support systems.

The format described here is intended to be

- Transportable and usable on 360, 6600, LSI11, etc., equipment and their compilers.
- Readable by real human beings to facilitate checkout.
- Editable on available text-editors.
- Used for both the linker input and output.

Record Format

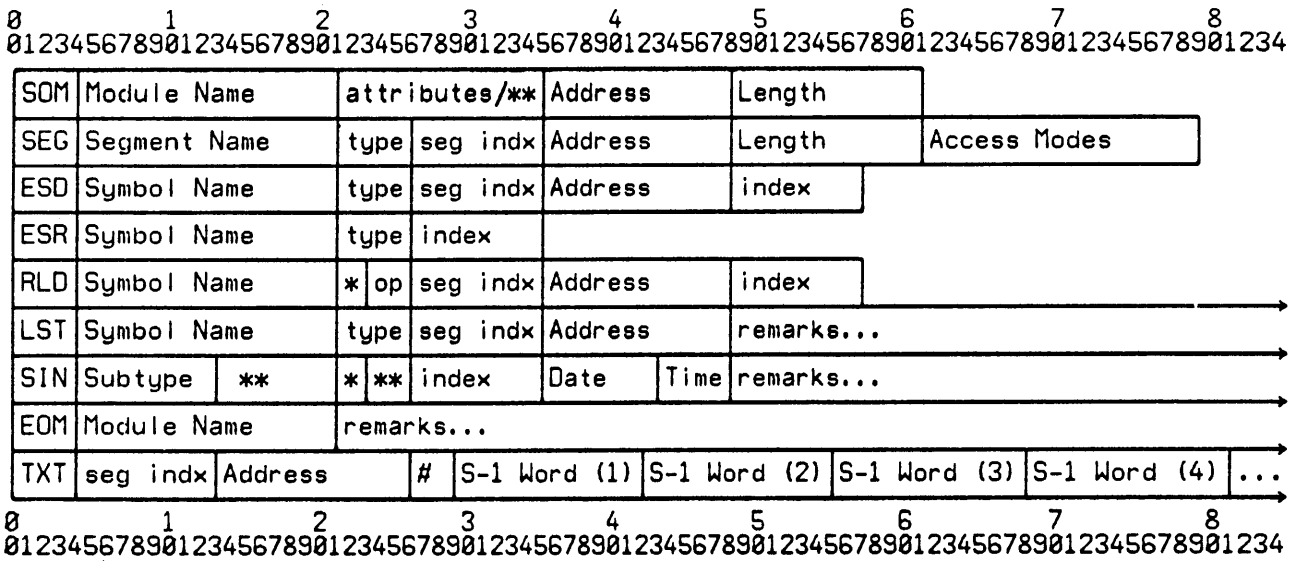
General:

Records will use a minimal subset of standard printable ASCII characters for their contents. They will have a maximum length of 1024 characters, but will often be limited to printer line length (132 characters). They will be identifiable using record positions 1-3. Records will be separated by ASCII CR and LF (octal 15 and 12) or by ASCII FF (octal 14). Each record type has its own fixed field format with blanks separating the fields. Trailing blanks may follow the defined portion of the record. However, extraneous non-blank information is not allowed. Blank records are ignored.

Since the S-1 word is 36 bits, 12 octal digits are used to denote addresses. Eight decimal digits will be used to denote segment index numbers and index numbers. Leading blanks are allowed in both formats instead of zeros. The format of other fields is described explicitly. However, no field other than comment fields, *e.g.*, on the LST, SIN, and EOM records, may have imbedded blanks.

All addresses specified to the LINKER are 36 bit byte addresses. All 36 bits will be maintained by the LINKER. For memory allocation, only the low order 30 bits are significant. The currently defined RLD operations affect only the low order 30 bits of the relocated word, leaving the high order 6 bits unchanged.

Record Format Summary



* source of index number (ESD, ESR, SEG).

** reserved for future use.

\# number of S-1 Words following.

The fields labelled `remarks...` contain unformatted information that is terminated by the end of the record. The linker does not verify the contents or format of these fields.

Detailed information on the format of each record type may be found in the section describing the particular record type. Detailed information on record order may be found in the section on record order.

Start of Module Record (SOM)

Format

```
Chars.  1-3     Record type.
                        Characters 'SOM'.
        5-20    Module Name.
                (Left justified, padded with blanks).
        22      Relocatability attribute.
                        'A' (Absolute) if module does not contain ESR or RLD
                            records and may not be relocated
                        'N' (Non-relocatable) if module contains ESR and/or
                            RLD records, but no segment may be relocated
                        'R' (Relocatable) if module may be relocated
        23      Order attribute.
                        'S' if module is in standard sorted order
                        'N' if module is not in standard sorted order
        24-34   Other attributes.
                (Reserved for future use).
        36-47   Lowest assigned address within module.
                (12 Octal digits).
        49-60   Length of module.
                (12 Octal digits).  Highest address less lowest address plus 1.
```

Description

This record is required for all modules, whether linked or not, and it must be the first record of the module.

The module name is used by the linker only for identification on messages but it must match the name specified on the EOM record. Duplicate module names for several modules input to the linker are allowed.

The relocatability attribute specifies whether the segments may be relocated and whether a relocation dictionary is present. An absolute module contains only segments that may not be relocated and may not contain a relocation dictionary, and hence may not contain ESR records. Only absolute modules are acceptable input to a non-relocating loader. A non-relocatable module contains only segments that may not be relocated. RLD and ESR records are allowed. A relocatable module contains only segments that may be relocated and contains a complete relocation dictionary and may contain ESR records. All modules with the absolute or non-relocatable attributes must precede modules with the relocatable attribute in the input to the linker. Note that the linker processes absolute modules the same as non-relocatable modules, except that a diagnostic message may be issued if RLD or ESR records are present. The distinction between absolute and non-relocatable modules is necessary for the absolute loader, which does not accept a relocation dictionary.

For information on relocation and on order, see the appropriate section in this document following the description of the record types.

Segment Definition Records (SEG)

Format

Chars. 1-3 Record type.
Characters 'SEG'.

5-20 Segment Name.
(Left justified, padded with blanks). Name of segment, except blank for unnamed segments.

22-25 Segment type.
(Left justified, padded with blanks).
'IS ' for instruction segment
'DS ' for data segment
'CM ' for common area

27-34 Segment index number.

36-47 Address assigned to start of segment (relative to start of module).
(12 Octal digits).

49-60 Length of segment.
(12 Octal digits).

62-63, 65-66, 68-69, 71-72, 74-75, 77-78
Memory access mode.
(6 fields of 2 characters each, unused fields blank and following all used fields). One or more of the following, in any order:
'IN' for instructions
'DA' for data
'RA' for read-allocate
'WA' for write-allocate
'WO' for write-only
'WT' for write-through

Description

Each module consists of one or more segments, which are identified by SEG records. No address may be contained within more than one segment, *i.e.*, segments may not overlap. Segments must be aligned on word boundaries, and may have arbitrary lengths, which are rounded up to a multiple of 4 (number of bytes in an S-1 word). Segments need not be assigned adjacent memory locations, *i.e.*, not all locations less than the highest assigned memory location need be assigned. Segment type IS is intended to be used to identify segments containing instructions. Segment type DS is intended to be used to identify segments containing data. Segment type CM is intended to be used to identify a segment of data not initialized at compile-time which is common to several segments, *e.g.*, FORTRAN common.

SEG records may be in any order subject to the following rules. A SEG record must appear before

any record referring to that segment, *e.g.*, ESD record for the same segment, TXT records for that segment, and RLD records for that segment. SEG records may be interspersed with other record types, but SEG records should precede all other records other than the module header record, SOM, if any.

Segment names must be unique within a module. However, unnamed segments do not conflict with each other even though they have blank segment name fields, except that at most one unnamed common segment is allowed per module. If multiple modules are linked together, depending on linker options, conflicting segment names may be handled by deleting all duplicate segments from relocatable modules with any particular segment name. Duplicate segments from absolute or non-relocatable modules will not be deleted automatically and will be handled as described in the section of this document on Relocation. All ESD symbols defined within a deleted segment and associated TXT and RLD records are also deleted. The exception to this rule is that conflicts among common segments with the same name are resolved by using the common segment with the largest length for each segment name and deleting all others, and similarly for unnamed common.

Segment index numbers are used to uniquely identify segments within the module. Therefore, these numbers must be unique. All of these numbers must be non-zero. It is preferred, but not required, that SEG records appear in the order of ascending segment index number. It is also preferred that segment index numbers be assigned sequential numbers starting at one. Linkers and loaders may have restrictions on the number of segments and on the maximum segment index number. Segment index numbers are assigned by compilers and assemblers and may be altered by linkers.

The memory access mode is used by the loader to set the appropriate bits in the page map. The linker ensures that IN is set for segment type IS and DA is set for segment types DS and CM. Segments with differing access modes may not share the same memory page (one page equals 512 words) in the linker output. Uncorrectable sharing, *e.g.*, in non-relocatable modules, will result in a warning to the user. Other restrictions may be enforced by the loader or by the hardware. Definitions and additional information may be found in **S-1 Multiprocessor Architecture** (SMA-*n*).

Segments from non-relocatable modules will retain their allocated addresses. Segments from relocatable modules will be given arbitrary addresses so that segments with different memory access modes will be allocated on separate memory pages. Upon request, the linker will provide the user with a map indicating the location of segments and symbols in memory.

External Symbol Dictionary Records (ESD)

Format

Chars.	1-3	Record type. Characters 'ESD'.
	5-20	Symbolic Name. (Left justified, padded with blanks). External symbol name, except blank for Start Symbols.
	22-25	ESD type. (Left justified, padded with blanks). 'ST ' for start symbols 'IN ' for instruction address name 'DN ' for data address name 'AN ' for absolute address name
	27-34	Segment index number of the containing segment.
	36-47	Address assigned within module (relative to start of module). (12 Octal digits).
	49-56	Index number assigned to this defined IN, DN, or AN; zero for ST.

Description

Each module consists of one or more segments, which may contain symbols specified on ESD records. Specific locations in segments may be identified through the use of ESD records. Symbolic names are correlated with the segments in which they are contained by the segment index number. The address of the symbolic name need not be contained within the segment, but it will always remain a constant offset from the start of the segment. ESD type ST is used to indicate the starting execution address of the main program. ESD type IN is intended to be used to identify the starting address of a section of code in a segment, *e.g.*, a subroutine entry point. ESD type DN is intended to be used to identify variables contained in one segment and referred to in another segment. ESD type AN is intended to identify absolute locations in memory or constants. Conventions regarding the use of ESD symbols (especially for multi-word variables) are the responsibilities of the compiler or language designer or the assembler language programmer. It is recommended that these conventions be clearly documented in compiler reference manuals to facilitate efficient programming.

Non-relocatable ESD symbols may indicate a segment index number of zero. This special segment index number indicates that the linker is not to alter the address of the symbol if any relocation should be done. ESD symbols with type AN must have a segment index number of zero. In relocatable modules, ESD symbols of type other than AN may not have a segment index number of zero. This feature may be used to facilitate creation of modules from core images or by an absolute assembler as well.

ESD records may be in any order subject to the following rules. An ESD record identifying a symbol must appear before any record referring to that symbol, *e.g.*, RLD records for that segment.

ESD records may be interspersed with other record types, but ESD records should follow all of the SEG records and precede all other records.

Symbolic names must be unique within a module. If multiple modules are linked together, conflicting symbolic names remaining after segment deletion are invalid, but may be handled by renaming all but the first occurrence of the symbolic name. Only one ESD type ST entry is allowed per module. Conflicts among ESD type ST entries from different modules are resolved by using only the first entry encountered that corresponds to a segment that is not being deleted. If no ESD type ST entry appears within a module, main program execution will start at the start of the first segment encountered with SEG type IS that is not being deleted.

Index numbers are used to uniquely identify symbols within the ESD. Therefore, these numbers must be unique. Certain rules apply to the assignment of these numbers. All of these numbers must be non-zero. However, no reference may be made to an ESD type ST by the ESR or RLD, so ESD type ST is given an index number of zero. It is preferred, but not required, that ESD records appear in the order of ascending index number. It is also preferred that index numbers be assigned sequential numbers starting at one, as they may duplicate segment index numbers. Linkers and loaders may have restrictions on the number of symbolic names and on the maximum index number. Index numbers are assigned by compilers and assemblers and may be altered by linkers.

External Symbol References (ESR)

Format

```
Chars.  1-3     Record type.
                        Characters 'ESR'.
        5-20    Symbolic Name of requested symbol.
                (Left justified, padded with blanks).  External symbol name
                to match a symbolic name of the ESD of another module.
        22-25   ESR type.
                (Left justified, padded with blanks).
                        'IR  ' for instruction address reference
                        'DR  ' for data address reference
                        'AR  ' for absolute address reference
                        'XR  ' for reference of unspecified type
        27-34   Index number in ESR assigned to this symbol.
```

Description

ESR records are used to indicate external references to locations not contained within this module, but defined in the ESD of another module. The symbolic name is used to correlate the ESR entry with the ESD entry that satisfies it. Symbolic names in the ESR must be unique and must not duplicate symbolic names in the ESD. The ESR records defining any particular symbolic name must precede all references to that symbolic name by the RLD. Unique, non-zero index numbers are assigned to each symbolic name in the ESR by compilers and assemblers and these numbers may be altered by the linker. These numbers are used to correlate the ESR entries with the RLD entries that refer to them. It is recommended that index numbers in the ESR be assigned consecutive numbers starting at one. Linkers and loaders may restrict the number of symbolic names in the ESR as well as the largest index number of an ESR.

ESR type IR may only be used to refer to externally located code, *i.e.*, ESD type IN; ESR type DR may only be used to refer to externally located data, *i.e.*, ESD type DN; ESR type AR may only be used to refer to an externally defined absolute address, *i.e.*, ESD type AN. ESR type XR may refer to an ESD entry of any type.

In order to resolve external references indicated by ESR records, several techniques are commonly used. Control commands supplied by the user are first used to obtain and edit modules. If an external symbol requested by an ESR is found in the ESD of another module, all records referring to the ESR, *e.g.*, are altered to refer to the matching ESD record. If any unresolved external references remain, the linker will usually then search automatic call libraries for the referenced symbolic names. Linker options are usually used to indicate the names of the automatic call libraries (usually language dependent), whether these libraries are to be searched, and whether execution is to be attempted if unresolved external references remain. Further information should be in the manuals describing the linker or loader being used or in the manual describing the compiler and/or compiler subroutine library.

ESR records may appear in any order. It is recommended that ESR records appear in ascending order and follow all ESD records within a module. ESR records must precede RLD records that refer to them.

Text Records (TXT)

Format

```
Chars.  1-3      Record type.
                        Characters 'TXT'.
        5-12     Segment index number of segment containing text.
        14-25    Initial loading address for the words in this record.
                 (Octal, right justified, leading blanks allowed).
        27-28    Number of S-1 words contained in this record.
                 (Decimal, right justified, leading blanks allowed).
                        4 or less for 80 column card format records
                        7 for printer format using 119 characters
                        8 for printer format using 132 characters
        30-41    First S-1 word.
                 (Octal, right justified, leading blanks allowed).
        43-54, 56-67, 69-80              Words 2, 3, and 4 in the same format.
        82-93, 95-106, 108-119, 121-...  Similar, always using 13 positions.
```

Description

TXT records are used to define the initial state of storage for instructions and data. TXT records may be in any order but must appear after all SEG records defining the segment containing the text and must precede RLD records relocating an S-1 word appearing on the TXT record. But it is preferred that the TXT records within a module appear in ascending order by starting address, which must be on a word boundary. In the case of conflicting TXT records, the last TXT record specifying the contents of each S-1 word is used to determine the actual contents to be loaded. TXT records are correlated with the segment containing the text by the segment index number. TXT records may not span segments. TXT records may not be used to initialize common segments.

TXT records in absolute modules may specify a segment index number of zero. This special segment index number indicates that the linker is not to check that the text lies completely within the bounds of any particular segment, but only that it lies within the bounds of the module specified on the SOM record. The user is cautioned against using a segment index number of zero to initialize memory that is not within the bounds of any segment, as doing so may cause unpredictable results during loading or execution. This feature is intended to facilitate generation of modules from core images, but may be used by an absolute assembler as well.

Words within a segment which are not initialized by TXT records will have unpredictable values at the start of execution. In particular, common segments will not be initialized.

Relocation Dictionary (RLD)

Format

Chars.	1-3	Record type. Characters 'RLD'.
	5-20	Symbolic name or segment name for relocation value. (Left justified, padded with blanks).
	22	Relocation source. 'D' if relocation value is address of ESD symbol 'R' if relocation value is address of ESR symbol 'S' if relocation value is starting address of segment
	24-25	Operation. (Left justified, padded with blanks). '+ ' for 30 bit add '- ' for 30 bit subtract
	27-34	Segment index number of segment containing the word to be relocated.
	36-47	Address (relative to start of module) of S-1 word to be relocated. (12 octal digits).
	49-56	Index number of ESD or ESR symbol (or segment index of segment).

Description

The RLD describes all words whose contents is location dependent. This includes words which are used to allow one segment to refer to another as well as address constants that allow one segment to refer absolutely to locations within that segment. For example, segment A may contain a word, ADCON, which should have the address of symbol B (or segment B). Then, there would be an RLD entry for the address of ADCON with the segment index of A and the index number of symbolic name B. The segment index in the RLD record is the segment index associated with the segment (types IS and DS only) containing the word to be relocated, *e.g.*, the segment index of A in the example above. The index number in the RLD record is the index number of the symbolic name in the ESD or ESR (or segment index number of segment B) whose address is to added to or subtracted from the contents of the word specified, *e.g.*, the index number of B in the example above. Words relocated through RLD records must be on word boundaries. Only the low order 30 bits of the word is altered (for the operations 30 bit add and 30 bit subtract. Other operations that may be defined in the future may have different rules). The high order 6 bits remain unchanged. Overflow over 30 bits in the computation is lost and has no effect.

The relocation source and index number are used to determine the relocation value. The relocation source determines whether the symbolic name or segment name appears in a ESD, ESR, or SEG record. The index number is used to determine which record of the specified type is being referenced. The symbolic name on the ESD or ESR record or the segment name on the SEG record must match the name specified on the RLD record.

The ordering of RLD records is restricted by the following rules. An RLD record must follow the last TXT record that defines the contents of the word to be relocated, *e.g.*, ADCON in the example above. An RLD record must also follow the ESD, ESR, or SEG record defining the symbolic name whose address is to added to or subtracted from the contents of the word, *e.g.*, in the example above, the RLD record must follow the ESD, ESR, or SEG defining the symbolic name B. If the symbolic name is an unresolved external reference, no relocation is done for that symbol. The word to be relocated will instead have the contents specified on the last TXT record, possibly relocated by other RLD records. For efficient loading, RLD records should immediately follow the last TXT record initializing the word to be relocated.

Every word to be relocated, *e.g.*, ADCON, must be initialized by a TXT record. Failure to do this will cause unpredictable results. In particular, words in common segments cannot be relocated.

Local Symbol Table Records (LST)

Format

Chars.	1-3	Record type. Characters 'LST'.
	5-20	Local Symbol Name. (Left justified, truncated to 16 characters or padded with blanks).
	22-25	Symbol type. (Left justified, padded with blanks). As defined by compiler or assembler.
	27-34	Segment index number of the containing segment.
	36-47	Address assigned within module relative to start of module. (12 Octal digits).
	49-	Comments. (Undefined format, unchanged by linker).

Description

Each module consists of one or more segments, which may contain local symbols specified on LST records. Specific locations in segments may be identified through the use of LST records. Local symbols are correlated with the segments in which they are contained by the segment index number. The address of the local symbol need not be contained within the segment, but it will always remain a constant offset from the start of the segment. The actual names of the local symbols are ignored by the linker. Local symbols are intended to be used by symbolic debuggers and may be used to refer to locations of variables appearing in the source. Conventions regarding the use of LST records (especially for multi-word variables) are the responsibilities of the compiler or language designer or the assembler language programmer and the debugging package designer.

Non-relocatable LST symbols may indicate a segment index number of zero. This special segment index number indicates that the linker is not to alter the address of the symbol if any relocation should be done. In absolute or non-relocatable modules, all LST symbols are non-relocatable and may have a segment index number of zero; in relocatable modules, only LST symbols that refer to absolute locations may have a segment index of zero. This feature may be used to facilitate creation of modules from core images or by an absolute assembler as well.

LST records may be in any order subject to the following rules. An LST record identifying a symbol must appear after the SEG record defining the containing segment. LST records may be interspersed with other record types, but LST records should follow all of the SEG, ESD, ESR, TXT, and RLD records and precede all of the SIN records.

Segment Information Records (SIN)

Format

Chars. 1-3 Record type.
Characters 'SIN'.

5-12 Record subtype.
'COMPILER' if record contains information about the compilation or assembly of segment
'LINKER ' if record contains information about the linking of this module
'USER ' if record contains information supplied by the user about a segment through commands during the linking of this module
'CLOBBER ' if record contains information supplied when TXT records were altered after compilation by a byte-altering program or a text editor
'VERSION ' if record contains information about the version of the segment or module

14-20 (Reserved).

22 Symbol source.
'S' if information is about a segment name in SEG
'D' if information is about a symbolic name in ESD
'R' if information is about a symbolic name in ESR
' ' (blank) if information is about the module

24-25 (Reserved).

27-34 Index number in ESR or ESD or segment index number in SEG identifying the symbol to which the information relates.
Zero if information is about the module and not a particular segment.

36-42 Date of Compilation/Assembly/Linking/Clobbering or Date Information Supplied.
(Standard S-1 Date format: ddmmmyy, mmm is one of JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, and DEC).

44-47 Time of Compilation/Assembly/Linking/Clobbering or Time Information Supplied.
(Standard 24-hour clock time format: hhmm).

49- Descriptive information.
Any character string for User and Clobber formats.
Following format for Compiler and Linker formats:

49-56 Program Name of Compiler/Assembler/Linker.

58-59 Version Number of Compiler/Assembler/Linker, Segment or Module. (Two decimal digits).

60-61 Modification Level Number of Compiler/Assembler/Linker,

```
                Segment or Module.
                (Two decimal digits).
        63-     Optional Compiler/Assembler/Linker-Dependent Information.
```

Description

One compiler type SIN record should appear for each segment and is supplied by the compiler or assembler that produced the segment. (Compiler type SIN records are also created by programs that convert a core image to a module.) Information contained on the Linker type SIN record is supplied automatically by the Linker during linking. During linking, previous Linker type SIN records are deleted. User and Version type SIN records are created during linking from user supplied information. At most one User and at most one Version type SIN record appears for each segment and for each symbolic name in the ESD and ESR, and newer User and Version type SIN records take precedence over older records. Information supplied to byte-altering programs is copied into Clobber type SIN records each time records are altered. If TXT or other records are altered manually, *e.g.*, through interactive text editors, it is strongly recommended that Clobber type SIN records be manually created to document the change.

The linker will not verify information in date, time, or descriptive information fields, but will copy such information intact on records not deleted. Records created by the linker will have the specified format and it is assumed that programs will also generate records of the specified format.

End of Module Record (EOM)

Format

Chars.	1-3	Record type. Characters 'EOM'.
	5-20	Module Name (to match that on SOM record, if any). (Left justified, padded with blanks).
	22-	Remarks. (English, free format).

Description

The EOM record must physically be the last record of the module. For modules that have not been linked, it contains compiler/assembler dependent information. For modules that have been linked, it contains linker-dependent information. The module name must match that on the SOM record, if any. Otherwise, the format is not documented here, and should be located in logic manuals for component producing the record. Unless documented here, it should not be relied upon to contain any specific format or value.

Relocation

Relocation consists of two parts: assigning absolute memory locations and merging the relocation dictionary into the text records. A relocatable module becomes non-relocatable when absolute storage locations have been assigned to the segments in the module. A module becomes absolute when the relocation dictionary has been merged into the text records. A description of the semantics of the linking process follows. The actual implementation should not be inferred from the description. The related process of resolving external references is described in the section on the ESR record.

The modules input to the linker may be a combination of absolute, non-relocatable, and relocatable. All absolute or non-relocatable modules must precede any of the relocatable modules. The linker processes absolute and non-relocatable modules similarly, except that it may verify that absolute modules don't contain ESR or RLD records.

When several absolute or non-relocatable modules are linked together, the segments in different modules may overlap. Since no segments in absolute or non-relocatable modules may be relocated, the overlap may not be resolved by relocation. Since segments from relocatable modules may be relocated, the following discussion will be concerned only with absolute and non-relocatable segments and not with relocatable segments, which will be treated in another paragraph. Segments that do not overlap with other segments are copied into the linker output. If a non-overlapping segment duplicates the name of another segment, it must be explicitly deleted or renamed. Each group of overlapping segments is treated separately and will be combined into a single segment. Each group must have the same segment type and memory access modes. The resultant segment will have that segment type and those memory access modes. The name of the resultant segment will be the name of the first segment encountered in the group. The resultant segment will be assigned all of the addresses assigned to the original segments. External symbols (ESD's) and local symbols (LST's) belonging to segments in the group will belong to the resultant segment. TXT and other records will be processed as if they were from a single segment and in the order encountered. (See also the section in this document on the specific type of record.) Deletion of absolute or non-relocatable segments is not advisable since the TXT records in other absolute or non-relocatable segments may already contain the address of the segment to be deleted. The linker will not automatically delete absolute or non-relocatable segments. If the linker allows requests for deleting absolute or non-relocatable segments, it is the responsibility of the user to ensure that errors do not result from such requests.

When relocatable modules are linked together or with absolute or non-relocatable modules, the segments contained in these modules must be assigned non-overlapping memory locations. These locations must also not overlap memory locations assigned to low core or to non-relocatable segments (which is why non-relocatable segments must come first). In addition, segments with different memory access modes must be assigned locations on different pages. The linker will assign ranges of memory addresses to each segment and leave gaps of memory locations unassigned in order to satisfy these requirements. The segments are then relocated into these assigned locations, relocating all address values in the address fields of TXT and other records. Since segments are kept contiguous, all words in any given segment are relocated by the same amount. The order of the resultant

segments is left up to the linker implementation, but may be based on the order of segments in the input or linker control records, if any. Note that the addresses assigned to the segments may be changed when subsequently relinking the module if it remains relocatable.

The relocatability attribute of the resultant module may be absolute, non-relocatable, or relocatable. If any of the input modules were absolute or non-relocatable, then the output module may not be relocatable. If the output module is not relocatable, then it will undergo the step of merging the relocation dictionary into the text. Note that this occurs for both absolute and non-relocatable modules. The only difference in the linker output is that unresolved external references (ESR records) and relocation involving external symbols (RLD records referencing ESR symbols) may remain in non-relocatable modules.

If the output module is not relocatable, then the relocation dictionary is merged into the text. Since each segment has been assigned a memory location, each symbolic name has a corresponding address. Therefore, except for unresolved external references, the relocation specified in the relocation dictionary may be accomplished by performing the indicated operation to the specified memory locations using the addresses corresponding to the specified symbolic names. (See the description of the RLD record for further information.) RLD records so processed are deleted and do not appear in the output. The remaining RLD records are those that refer to the unresolved external references. If the output module is to be absolute and unresolved external references remain, it is up to the linker implementation to determine what action is to be taken. The linker may report the error and/or delete the ESR and RLD records involved.

Alignment

All addresses specified to the LINKER are 36 bit byte addresses. Segments must be aligned on word boundaries, and may have arbitrary lengths, which are rounded up to a multiple of 4 (number of bytes in an S-1 word). Addresses specified on ESD, ESR, and LST records may have any alignment. The starting address on the TXT record must be on a word boundary. Words relocated through RLD records must be on word boundaries, although the currently defined RLD operations only alter the low order 30 bits. Segments with differing access modes may not share the same memory page (one page equals 512 words) in the LINKER output. Uncorrectable sharing, *e.g.*, in non-relocatable modules, will result in a warning to the user.

Record Order

The description of each of the record types describes the order in which the records may appear. Since order does affect module processing by the linker and the loader, a standard preferred order is defined. Modules that have not been processed by the linker are not assumed to be in standard order. For modules that have been processed by the linker, the SOM record indicates whether the modules is in standard order.

Standard order has been chosen so as to optimize module processing. The first record must be the SOM record. SEG records follow in ascending segment index number order. ESD records follow, in ascending index number order. Also in ascending index order are the ESR records, if any. The TXT and RLD records follow, in ascending segment index number order. Any LST records follow in ascending order by segment index number of containing segment. SIN records follow in arbitrary order. The last record is the EOM record. Standard order format allows the module to be processed with the minimum amount kept in memory during processing.

Linker Options

Linker options inform the linker of information such as the whether output is to be absolute or relocatable. When the linker is available, the format of these records and their use will be documented in this section. Until then, no statements about the capability of the linker or the form of the commands to the linker may be implied. Options previously stated in this document are suggestions for options to be implemented, but should not be taken to mean that these options will be available in the first (or in any subsequent) version of the linker.

The linker documentation, when available, should be used to determine implementation dependent information, such as options available. The reader may also find it worthwhile to consult the documentation on the compiler or debugging package being used.

S-1 LINKER
Implementation Manual

(SLIM-0)

Arthur M. Keller

26 January 1979

0. Table of Contents

S-1 LINKER Implementation Manual

Arthur M. Keller

Abstract. The Linker is part of the S-1 programming system. It was designed to support linking modules from compilers of different languages. In order to make it transportable, it was written in PASCAL. The output can be relocatable or absolute, but it is usually absolute.

Index terms. Loader, linker, S-1, PASCAL, transportable, module, object code, relocation.

Acknowledgements. The author wishes to acknowledge crucial support for this work which has been received from the Department of the Navy via Office of Naval Research Order Numbers N00014-77-F-0023 and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which is operated for the U. S. Department of Energy under Contract No. W-7405-Eng-48) and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation). This work was also partially supported by a National Science Foundation Graduate Fellowship.

The author would like to thank a number of people who contributed suggestions helpful in the design of the LINKER or in the improvement of this manual. These include Gio Wiederhold, Erik Gilbert, Mike Farmwald, Jeff Rubin, and Curt Widdoes.

I. Introduction

The LINKER reads the output of compilers, relocates it, and writes output for the loader. The format of the input and output is LDI [KeW78]. The input can be relocatable, i.e., contain relocation information, or it can be non-relocatable. The output can be relocatable, but it is usually absolute or non-relocatable. The purpose of running the LINKER is to resolve external references and to relocate location dependent words. It is used to support separate compilation of programs and subroutines or "runtimes." LDI format is currently produced by SOPA [WaG78] and FASM [HaH79] and is read by FSIM [Rub78] and by the loader on the S-1 [WiH78].

II. Overview

The LINKER operates in two phases, input and output. In addition, there is initialization and termination. The LINKER is essentially a two pass operation, with the first pass being done during the input phase and the second pass being done during the output phase.

During the input phase, segments (and symbols) are assigned memory locations, and TXT, RLD, LST, and SIN information is saved for output. RLDs referring to symbols and segments that have already been assigned addresses are performed, while other RLDs are saved for later processing. The equivalent of a core image is built in the internal data structure. All of the input is read before any of the resultant module is output.

During the output phase, the remaining RLDs are performed, except for those referring to unresolved ESRs. In addition, the module is prepared for output, including setting up the SOM record and updating segment index numbers to reflect merged segments.

III. Data Structures

In order to understand the operation of the LINKER, it is important to understand the data structures used by the LINKER.

Information about the modules read are stored in an array indexed by the sequential module number (assigned by the LINKER). This information is primarily for debugging purposes, and so, when the table is full, the last position in the table is reused.

Segments, ESD symbols and ESR symbols are referenced within a module by specifying their index number. As a result, the information is organized into an array for each record type indexed by the index number. These tables (SEG, ESD, and ESR) contain information contained on the

input record as well as additional information determined by the LINKER. Since index numbers for a particular record type are unique only within a module and may be duplicated in other modules, a base index number is assigned to each module for each record type. This base index number is added to the index number appearing in the input to obtain a unique index number for use throughout the LINKER. The addition of the base index number does not occur for index number zero, which is used for special purposes. Segment index number zero is a special non-relocatable segment used for storing text from non-relocatable modules and non-relocatable LSTs. There may not be a SEG record specifying segment index number zero, although segment index number zero may appear on other records. ESD index number zero is used for the start symbol. ESR index number zero is not used and is illegal.

A data structure pointed to by the SEG table is used to store the text from the TXT records and relocation definitions from the RLD records. (Text is the definitions of initialization of storage for code or data.) Text for relocatable modules is stored by the segment. This is so that the segments can be relocated without having to alter the text data structure. The SEG table for a particular segment containing text definitions points to an array of pointers. Each of the pointers in this array point to a list of text blocks. Each text block describes a portion of memory (currently 64 words, as defined by BLOCK_WORD_COUNT, a compile time parameter). It contains an array for the text words, a pointer to a chain of RLD blocks, the relative word number of the lowest and highest numbered words initialized in the text block, and the offset of the first word of the block from the start of the segment. The relocation definitions (RLD's) for relocating words in a text block are stored in a linked list based in the text block.

Text initializations from different absolute modules may overlap. In addition, it is not necessary to be able to relocate segments from absolute modules. As a result, text and RLD information from absolute modules are stored together in a special segment specifically for this purpose, segment number zero. The text structure for segment zero has the same format as for segments from relocatable modules, as described above.

Local symbols defined on LST records are stored in a linked list based in the SEG table entry for the segment. Local symbols specifying a segment index number of zero to prevent relocation by the LINKER are stored together under segment index number zero.

SIN records are stored in a linked list depending on the symbol source. The SIN record is chained from the SOM, SEG, ESD, or ESR table depending on the value of symbol source. There are separate linked lists for each index number for each record type. Currently, no SIN records are generated by the LINKER since there is no access to date and time.

Information on the EOM record is compared with that on the corresponding SOM record, so there is no need to store this information separately.

Between modules, symbolic names are used to correlate references. There are two hash tables maintained for lookup of symbolic names. One contains segment names and is used for merging COMMON block definitions and verifying that other segments have unique names. The other hash table contains symbolic names from ESD and ESR records, and is used for matching up ESRs from one module with ESDs in another module. Both hash tables have the same organization: an array of pointers to lists of entries.

To store variable length information appearing on LST and SIN records, a string array is used. A string descriptor consisting of an index into the string array and a length is stored in the block describing the LST or SIN record.

The input line is stored in an array INPUT_LINE. The entire line is read before any of it is processed so that error messages can be issued containing the input line without waiting for the rest of the line to be read. This also facilitate some exception handling. For example, when an EOM record is omitted so that an unexpected SOM record is encountered, an EOM record is generated and a flag is set so that the SOM record will be reread and processed.

IV. Detailed Procedure Documentation

This section gives a brief description of the procedures in the LINKER. These procedures are divided into functional groups called classes. Each class is described separately below.

1. Built-in Function Class

The built-in function class consists of functions that should be built-in function. These are MIN and MAX, which each take two integer arguments and return an integer result.

2. Clock Class

The clock class consists of implementation dependent functions to interrogate the clock. Currently the only procedure is READ_CLOCK which returns the elapsed run-time in milliseconds (with unknown origin). Eventually, procedures will be needed to obtain date and time so that they can appear in the SIN record.

3. Initialization Class

The initialization class consists of procedures that initialize the state of the program of initialize variables. The procedure INIT_SPECIAL_ACTION initializes the special action field of the SEG or ESD record. The procedure SET_OPTIONS sets option switches based on the values of options. The procedure INITIALIZATION performs the bulk of the initialization including opening files and initializing fixed length tables and counters.

4. Error Class

The error class consists of procedures that produce error messages.

The WRITE_LINE procedure writes the input LDI record to the output message file.

The WRITE_ERR_MESSAGE procedure writes the actual error message. There are a common set of error messages independent of the severity of the error, as several messages are issued with different severities. Some of the messages have a variable part, usually a symbol, which contains additional information. The message number is printed, except for the symbol-only message.

The PRINT_COL_PTR procedure writes an uparrow under the most recently read column when the error was detected. It is only printed if the caller to SYSERROR, ERROR, or WARNING specified COL_VALID.

The INFOMSG procedure writes two symbols to the output file, usually some short message. On the PDP-10 the message is also written to the terminal.

There are three procedures for printing error messages, SYSERROR, ERROR, and WARNING. The SYSERROR procedure is called for internal errors in the LINKER, for example, inconsistencies in data structures. The ERROR procedure is called for errors that preclude complete processing of the current input LDI record. Additional error detection may occur for the current record, but processing continues as if this record were omitted. The WARNING procedure is called for errors that can be corrected.

5. Read Line Class

The read-line class consists of four procedures, OPEN_NEXT_LDI_INPUT_FILE, READ_LINE, READ_FIRST_COLUMN, and INPUT_CHAR.

The procedure OPEN_NEXT_LDI_INPUT_FILE is system dependent. On the S-1, it

simply does a reset of LDI_IN which causes the user to be prompted for the name of the next input file. If all input has been supplied, the name of an empty file should be supplied. On SAIL, the built-in procedure GETFILENAME is used to prompt the user for the next file name. If the filename is null, no action is taken, so effectively, a null file is returned since OPEN_NEXT_LDI_INPUT_FILE is only called to open the file the first time and subsequently, only after EOF. If the filename is not null, the user specified file is opened.

The procedure READ_LINE reads the next input line into the array INPUT_LINE. The record length is stored in REC_LENGTH. If end of file is encountered, OPEN_NEXT_LDI_INPUT_FILE is called. The flag GBL_EOF is set if the next input file is empty, as end of input is signified by an empty file or, on SAIL, a null filename.

The procedure READ_FIRST_COLUMN resets the input column counter (THIS_CHAR_POS) so that the next character read is from the first position.

The function INPUT_CHAR reads the next character from the INPUT_LINE. It ensures that reading is not done past the end of the line, and issues an error message and returns blanks should this occur. One blank has already been added to the end of the line by READ_LINE. This is necessary since all fields are assumed to be followed by a delimiter.

6. Read Field Class

The read-field class consists of procedures to read each data type that can appear in the input LDI file. In addition, the procedure READ_DLM reads the delimiter following a field on the input record.

The procedure READ_SYMBOL reads the next SYMBOL_LENGTH characters. The symbol is verified to be left justified and contain no imbedded blanks. It may, however, be entirely blank. The delimiter following the symbol is read by READ_DLM.

The procedures READ_CHAR, READ_2_CHAR, READ_REC_TYPE, READ_4_CHAR, READ_7_CHAR, READ_8_CHAR, and READ_12_CHAR read 1, 2, 3, 4, 7, 8, and 12 characters, respectively. A check is made for imbedded blanks, but the non-blank information is not assumed to be left or right justified. The field may be entirely blank. The delimiter following the characters is read by READ_DLM.

The procedure READ_INDEX_NUMBER reads an eight digit decimal index number. The eight characters and delimiter are read by READ_8_CHAR. The number must be right justified. If it is entirely blank, a warning message is issued and zero is returned. The number read is checked to ensure that it does not exceed the maximums allowed for each type of index number, segment, ESD and ESR.

The procedure READ_2_DIGITS reads a two digit decimal number. The two characters and delimiter are read by READ_2_CHAR. The number is must be right justified. If it is entirely blank, a warning message is issued and zero is returned.

The procedure READ_S1_WORD reads a 12 octal digit number. The 12 digits and delimiter are read by READ_12_CHAR. The number must be right justified. If it is entirely blank, a warning message is issued and zero is returned.

The procedure READ_S1_ADDRESS calls the procedure READ_S1_WORD. Separate procedures were created for readability and maintenance reasons.

The procedure READ_STRING reads the remainder of the INPUT_LINE into the STRING_ARRAY. If the space in STRING_ARRAY is exhausted, as much as will fit is stored and the remainder is discarded and an appropriate error message is issued. Trailing blanks are stripped off. The procedure is used for reading information that is to pass through the LINKER unchanged that appears in the LST and SIN records.

The procedure VERIFY_TRAILING_BLANKS verifies that the only blanks follow the fields that have been read on the input record. If trailing non-blank information is found, a warning message is issued. Trailing non-blank information is allowed on EOM records (and is appropriately ignored by the LINKER). Errors in the fixed portion of the TXT record may result in a warning message as the variable portion of the record (the text itself) may not be read in certain circumstances.

The procedure READ_DLM reads the delimiter that follows each field. This delimiter must be blank. A trailing blank is added to the end of the input record by READ_LINE so that there is a delimiter following the last field of the record. Note that delimiters follow fields on input, but precede fields on output. This peculiarity resulted from the fact that originally delimiters followed fields on output, resulting in an additional delimiter following all records. There is no reason other than programmer laziness precluding having delimiters precede all fields except the record type field in the input. No functional difference would result from such a change, however.

7. Write Field Class

The write-field class consists of procedures to write each data type that can appear in the output LDI file. In addition, the procedure WRITE_DLM writes the delimiter preceding a field on the output record. Also, the procedure WRITE_EOLN does a WRITELN to the output LDI file.

The procedure WRITE_SYMBOL writes the symbol to the output LDI file following a delimiter (produced by calling WRITE_DLM). No check is made for the format of the symbol, but

it is by convention left justified.

The procedures WRITE_CHAR, WRITE_2_CHAR, WRITE_4_CHAR, WRITE_7_CHAR, WRITE_8_CHAR, and WRITE_12_CHAR write a delimiter followed by the argument, which consists of 1, 2, 4, 7, 8, and 12 characters respectively.

The procedure WRITE_REC_TYPE writes a 3 character record type to the output LDI file. Unlike the other procedures, it does not cause a delimiter to precede the field being output as it by convention is the first field of the output record and does not need a preceding delimiter.

The procedure WRITE_INDEX_NUMBER converts the input index number to a string of eight decimal digits and then calls WRITE_8_CHAR to write the delimiter and the 8 digits. Leading zeroes print as blanks. The index number passed to this routine is checked to ensure that it is non-negative. Since the index number -1 is used to indicate unknown, invalid, or NIL (for lists of index numbers), attempts to print out such an index number could only result from an internal error in the LINKER.

The procedure WRITE_2_DIGITS converts the input number to a string of 2 decimal digits. The procedure WRITE_2_CHAR is used to write the preceding delimiter and the string of 2 digits. A leading zero prints as a blank.

The procedure WRITE_S1_WORD converts the input number to a string of 12 octal digits. The procedure WRITE_12_CHAR is used to write the preceding delimiter and the string of 12 octal digits. Leading zeroes print as blanks.

The procedure WRITE_S1_ADDRESS calls WRITE_S1_WORD to do the dirty work. Separate procedures were created for readability and maintenance reasons.

The procedure WRITE_STRING writes a string from the STRING_ARRAY. The preceding delimiter is written by WRITE_DLM. Only as many characters as actually appear in the string are written to the output LDI file. Null strings (length 0) are supported by this routine, as they can result from no information on the input record or from having exhausted string space.

The procedure WRITE_DLM writes the delimiter that precedes all fields except the record type field. The delimiter is a single blank. The description of READ_DLM in the read-field class contains other pertinent information about use of delimiters.

The procedure WRITE_EOLN does a WRITELN to the output LDI file. Subsequent output will appear on a new line.

8. Read LDI Record Class

The read-LDI-record class consists of procedures to read each type of LDI record. These procedures only convert the record to internal form (by use of procedures in the READ_FIELD class). They do not verify the consistency of the information. Except for READ_TXT_HEADER, these procedures read the entire input record, except for the record type field, which has already been read. The procedure READ_TXT_HEADER, however, reads only the fixed portion of the record. The remainder of record, which consists of the text words, is read by the HANDLE_TXT procedure. Note that an error on the TXT record may result in not calling HANDLE_TXT, and thereby cause the spurious error message about trailing non-blank information to be reported.

9. Write LDI Record Class

The write-LDI-Record class consists of procedures to write LDI records. The procedures WRITE_SOM, WRITE_ESD, WRITE_ESR, WRITE_RLD_BLOCK and WRITE_EOM write a single record, using the procedures in the WRITE_FIELD class to convert from internal to external format. The procedure WRITE_TXT_BLOCK writes TXT records for an entire text block. RLD records relocating words in a text block are written immediately after the TXT records for the text block. The procedures WRITE_LST_LIST and WRITE_SIN_LIST write one record for each record in the list, which may be null.

10. Symbol Table Class

The symbol table class consists of procedure to manipulate the two symbol tables in the LINKER. The procedure LOOKUP returns a pointer to a symbol table entry, which is NIL if the symbol is not in the table. The procedure ENTER enters a symbol table record into a symbol table. The COMPUTE_HASH_CODE procedure computes a hash code for a symbol. The NEW_SEG_SYMBOL and NEW_ESD_SYMBOL procedures return a pointer to a symbol table entry for the appropriate table. The table of segment names is.used to ensure that there are no duplicate segment names. The table of ESD/ESR symbol names is used to link ESR's to ESD's and to ensure that there are no duplicate ESD names. Segment names may duplicate ESD names.

11. Arithmetic Class

The arithmetic class consists of routines that perform useful arithmetic functions. The procedure S1_WORD_TO_ADDRESS returns the low order 30 bits of an S-1 word. The procedure SPLIT_S1_WORD splits an S-1 word into the high order 6 bits and the low order 30 bits, so that the inverse operation is addition. The procedure ADD_S1_ADDRESS adds two addresses, with the high order 6 bits being those of the first argument. The high order 6 bits of the

second argument and carry from the addition of the two 30 bit addresses are ignored. The procedure ADDRESS_COMPARE compares two address ranges and indicates whether there is any overlap, containment or equality. The procedure MERGE_ADDRESS_RANGES merges two address ranges. Addresses between the two ranges, but not within either are included in the resultant range, but this is not used currently.

12. Support Routines Class

The support routines class consists of procedures that perform various useful in verifying and manipulating the input.

The procedure MERGE_ACCESS_MODES OR's two sets of access modes, and is used when merging procedures that overlap and for handling common segments with the same name.

The procedure MERGE_SEGMENTS merges segments from non-relocatable modules that overlap.

The procedure CHECK_OVERLAP determines whether segments overlap. A parameter determines what action is taken if they overlap. If the input is relocatable, a warning message is issued for overlapping segments in the same module. If the input is non-relocatable, overlapping segments will be merged. A third option defines overlap to mean "shares a memory page" and issues a warning if two segments from non-relocatable modules share a memory page but do not have the same memory access modes.

The procedure ALLOC_ADDRESS_RANGE allocates the next available block of storage to the segment. If the memory access modes for this segment and the previous segment differ, the new segment starts on a page boundary.

The procedure MERGE_COMMON_SEGMENTS merges common segments favoring the segment described by the first parameter if the lengths match. Otherwise the larger is chosen.

The procedure GENERATE_NAME is used to generate a unique name for renaming ESD symbols with duplicate names. No check is made to ensure that the name generated does not conflict with an existing name, as it is expected that the names generated will only be used for this purpose as they contain strange characters.

The procedure LINK_ESD_ESR links all ESD's and ESR's with the same name together. This is used to convert RLD's refering to ESR's to refer to ESD's.

There are several procedures to verify index numbers. The procedure VERIFY_SEG_INDEX_NUMBER_AND_ADDRESS adds the base segment index number for the

module to the index number (because that's how its stored in the table), ensures that the segment is defined, and adds the appropriate displacement to the address. The address need not be within the segment. The procedure TXT_RLD_VERIFY_SEG_INDEX_NUMBER_AND_ADDRESS does the same in addition to verifying that the segment index number is reasonable (e.g., not a common segment, segment index number zero only for absolute modules) and that the address range is contained the segment. The procedures VERIFY_ESD_INDEX_NUMBER and VERIFY_ESR_INDEX_NUMBER add the appropriate base index number to the index number and ensure that it has been defined.

13. RLD Class

The RLD class consists of procedures that perform functions associated with the RLD record. The procedure RLD_CONVERT_ESR_TO_ESD converts an RLD that refers to a resolved ESR to refer to the corresponding ESD. The procedure CHAIN_RLD places an RLD_block on a chain of RLD's. The procedure REMOVE_RLD_FROM_CHAIN removes an RLD_block from a chain of RLD's. The storage associated with the RLD_block is lost, as no further storage will be obtained by the LINKER as it currently is written. The procedure PERFORM_RLD performs the RLD by relocating the text word by the value of the symbol.

14. Handle Input Class

The handle input class consists of procedures to handle each type of input record.

The procedure HANDLE_SOM performs various functions associated with the SOM record. These include setting up new base index number values, setting flags to indicate attributes of the module, and verifying the consistency of the relocatability attribute of the module, previous modules and the output.

The procedure HANDLE_SEG verifies that the segment index number has not been used (after adding the base segment index number for the module), verifies the consistency of the memory access modes with the segment type, checks for duplicate segment names, deleting duplicates of non-common segments and merging duplicate common segments, and verifying the address range and its overlap with other segments.

The procedure HANDLE_ESD verifies that the index number has not been assigned (after adding the base index number for the module except for ESD index number 0), verifies the consistency of the index number with the segment type, verifies the segment index number, verifies that the name has not been used to define an ESD (renaming duplicates), adds the symbol to the ESD name table, and links corresponding ESR's.

The procedure HANDLE_ESR verifies that the index number has not been assigned (after adding the base index number for the module), adds this ESR to the list of ESR's with the same name, and links it to the corresponding ESD if it has already been encountered.

The procedure HANDLE_TXT verifies the segment index number and address range and reads the remainder of the input record, storing the text into the text structure, creating parts of it as necessary. The text is stored by the segment for relocatable modules, but stored together (under index number 0) for absolute segments in order to facilitate of absolute segments that overlap.

The procedure HANDLE_RLD verifies the segment index number and address, tries to verify that the word to be relocated has been initialized by a text record (which cannot be completely done unless a compile time switch, CHECK_TXT_RLD, is set), verifies that the symbol on the RLD record matches the symbol on the SEG, ESD, or ESR record referred to by the RLD record, and either performs of chains the RLD for later action. The RLD is performed at this time if output is non-relocatable.

The procedure HANDLE_LST verifies the segment index number and address and chains the local symbol record off of the SEG_table.

The procedure HANDLE_SIN verifies index number and chains the SIN record off of the record to which it refers.

The procedure HANDLE_EOM verifies that the name matches that of the SOM record (or the SEG record, if no SOM started off the module).

15. Read Module Class

The read module class consists of the READ_MODULE procedure which reads a single module. Input records are flushed until an SOM or an SEG record is encountered. (SEG records are accepted instead of SOM records because SOPA does not currently generate SOM records. The processing proceeds as if a SOM record preceded the SEG record with the information from the SEG record. This code will no longer be needed when SOPA generates SOM records.) The SOM (or generated SOM) record is read and processed. If an error is encountered, the remainder of the module is flushed (until an EOM or SOM record is encountered). If no error is encountered, the remainder of the module is read and processed. An EOM record is generated if an SOM record or end of input is encountered.

16. Write Module Class

The write module class consists of procedures to prepare the data for output and then to write the module. The SET_UP_SOM_RECORD procedure sets up the SOM record for the resultant module. The UPDATE_SEG_INDEX_NUMBER procedure updates the segment index number in records that contain a segment index number to reflect merged segments. It is called from procedures that write individual record types. The WRITE_MODULE procedure writes the resultant module. The remaining RLD records that do not refer to unresolved ESR records are performed. The order of the output is SOM, SEG (in ascending segment index number order), ESD (in ascending index number order), ESR (in ascending index number order), TXT and RLD (by segment and text block), LST (in ascending segment index number order), SIN (in ascending index number order), and EOM.

17. Termination Class

The termination class consists of the TERMINATION procedure, which writes statistics to the output message file.

18. Main Procedure

The main procedure performs initialization, reads modules until end of input, writes the resultant module, performs termination and maintains clock times.

V. Compile Time Parameters

The LINKER includes a number of parameters specified at compile time, such as size of tables. Although the author would prefer for some of these to be specified at run time, the constraints of PASCAL require that they be specified at compile time. All of the compile time parameters are specified by PASCAL CONSTants.

There are a group of constants associated with table sizes. These constants are MAX_SOM_NUMBER, which specifies the size of the SOM table, and MAX_SEG_INDEX_NUMBER, MAX_ESD_INDEX_NUMBER, and MAX_ESR_INDEX_NUMBER, which specify the size of the SEG, ESD, and ESR tables, respectively. These numbers may all be different. Since SOPA places each segment in a separate module, it is recommended that MAX_SOM_NUMBER match MAX_SEG_INDEX_NUMBER. MAX_RLD_INDEX_NUMBER and MAX_INDEX_NUMBER must be the maximum of MAX_SEG_INDEX_NUMBER, MAX_ESD_INDEX_NUMBER, and

MAX_ESR_INDEX_NUMBER. Fixed size tables are allocated for storing SOM, SEG, ESD, and ESR information because of the speed and simplicity of referencing entries by index number. The author decided that this speed and simplicity was more important than allowing these tables to be of dynamic size (which would require some organization other than array, such as a hash table). The author would prefer if these arrays could be allocated dynamically based on execution parameters so that recompilation would not be needed for linking big programs and space would not be wasted for small programs.

There are two parameters defining maximum record size. One, MAX_REC_LENGTH, defines the maximum input record length. The other, MAX_TXT_REC_WORD_COUNT, defines that maximum number of text words that can appear on a TXT record, and should be computed based on the maximum record length. These parameters are based on the definition of LDI.

The parameter SYMBOL_LENGTH defines the length of a symbol. This value is currently 16, but is 8 for linking output from old version of SOPA and FASM. This value is based on the definition of LDI. Altering this value has the affect of shifting the information on the remainder of the records which have symbols on them.

The parameter SIZE_HASH_TABLE defines the size of the hash tables for symbols. Statistics indicating the use of these hash tables are listed in the message file. The size of the hash table is one greater than the value specified.

There are three parameters defining the text structure. One parameter, BLOCK_WORD_COUNT, specifies the number of words in a text block (less one, since it is indexed from zero). Decreasing this value wastes less space for small segments but requires more blocks for large segments. Statistics on the utilization of text blocks are listed in the message file. Another parameter, TXT_HASH_TABLE_SIZE, specifies the size of the hash table for text blocks. Increasing this value decreases the length of the chains of text blocks but requires more space. Statistics on the length of chains of text blocks is listed in the message file. Another parameter, CHECK_TXT_RLD, indicates whether the LINKER is to strictly check the order of TXT and RLD definitions for each word. This involves each text block having two arrays for two booleans for each word specifying whether a TXT or RLD definition for the word has been encountered. If this parameter is set to FALSE, these arrays are not allocated, and since the LINKER is no longer certain which words have been initialized with TXT records and which haven't, memory is initialized to S1_WORD_FILL (currently MAXINT) and words may be output which have this LINKER specified value. Some of the words that have been uninitialized will appear in output TXT records with the value S1_WORD_FILL while others will not appear. This discrepancy is due to the fact that the LINKER keeps track of the lowest and highest numbered word in a text block that has been initialized. Words within this range will be output in TXT records while words outside of this range will not be. This may cause unexpected results when overlapping non-relocatable modules which were previously linked are linked together.

The parameter OUTPUT_WORDS_PER_TXT_RECORD specifies the maximum number of words that will be output on a single TXT record. TXT records with a greater number of words may be read, but will be split on output.

The parameter DEBUG_ACTION controls certain debugging code in the program. The setting of this parameter specifies that additional information is to appear in the message file but does not affect processing. The only current use of this parameter is to indicate records deleted because of deleted segments. (Segments are deleted when they have duplicate names.)

The parameter INIT_MAX_ERRORS indicates the maximum number of errors or warnings that can occur. When this limit of errors is exhausted the LINKER terminates with an appropriate error message.

The parameter INIT_RELOC_OPTION indicates the desired value of the output relocatability attribute. The output cannot be relocatable if the input contains a non-relocatable (or absolute) module. The output cannot be absolute if it has unresolved ESRs. The output will be made non-relocatable if either of these rules is violated.

VI. Conventions

A number of coding conventions have been used in the LINKER. All error names are given names starting with a capital W. All types names start with T_. All string constants start with a capital X. Arrays containing tables of string constants for converting from internal to external representation are given names starting with XT. The abbreviation SEG is used for segment in all identifiers. Upper case is used for keywords, predefined types, abbreviations, and record type names. Lower case is used for identifiers, except for abbreviations and record type names.

VII. Running the LINKER

Since the LINKER is written in PASCAL, its behavior is affected by the PASCAL run-time package. This description is dependent on the current version of the run-times and does not take into account DO files.

The following illustrates running the LINKER on the PDP-10 at SAIL. The procedure for running the LINKER on the S-1 is similar with differences noted. User input is enclosed in pointy brackets. Comments appear in curly brackets.

```
. <RU LINK[DMP,S1]>                {at SAIL}
OUTPUT     = <file.LOU>
LDI_OUT    = <file.ALD>            {FILE1 on the S-1}

DEBUG: LINKER                      {at SAIL only}
$<END>                             {at SAIL only}


LDI_IN     = <file.LDI>            {FILE0 on the S-1}

READING MODULE1                    {these messages appear
                                   at SAIL only}
READING MODULE2

LDI_IN     = <PASSIX.LDI[LIB,S1]> {name of PASCAL run time at SAIL}

READING PASRUN
{When all LDI input files have been specified}
LDI_IN     = <>                    {on the S-1, specify the name
                                   of an empty file}
INPUT COMPLETED.                   {these messages appear
                                   at SAIL only}
WRITING MODULE1

Exit
↑C
```

The standard extensions are LDI for relocatable LDI files, ALD for absolute (or non-relocatable) LDI files, and LOU for the LINKER output message file.

VIII. Suggested Changes for the Future

There are several changes envisioned by the designer of the LINKER which can be implemented in the future. Explanation of these changes and suggestions on how to implement them are given below. It is recommended that the LINKER be tested with the test file TEST.LDI[LNK,S1] at SAIL, which tests out the error messages and most of the program.

1. Support LDF format

Currently, only LDI format is supported. This format is very wasteful of disk space and

requires conversion to internal format before it can be manipulated. Since the input format LDI uses ASCII characters, a major portion of the code and execution time is involved with reading the input and converting it to an internal format suitable for manipulation. Although LDF format [OKK78] has not been maintained, it would be useful to update LDF to be functionally compatible with LDI format and for it to be used by the LINKER as well as SOPA, FASM, FSIM, and the LOADER. This requires that files of arbitrary type be supported by PASCAL (PCPASC and SOPA).

The changes required to the LINKER are confined to the routines that are in the READ_LINE, READ_FIELD, WRITE_FIELD and ERROR classes. It may also affect routines in the READ_LDI_RECORD and WRITE_LDI_RECORD. The changes would be relatively straight-forward.

2. Support FORTRAN BLOCK DATA

Currently common segments may not be initialized. Block data may be supported by specifying a special segment type which may be initialized. It would behave like a regular common segment except that it could be initialized, it would be required to have the largest size of all common blocks with that name, and it would supersede all other common blocks with that name. Multiple initialized common segments with the same name would not be allowed.

The changes required include expanding the possible values of the segment type field, altering READ_SEG to reflect that, altering HANDLE_SEG to treat both types of common segments similarly, altering MERGE_COMMON_SEGs to give preference to the initialized common segment, and altering TXT_RLD_VERIFY_SEG_INDEX_NUMBER_AND_ADDRESS to allow TXT and RLD records for initialized common segments.

3. Support Large Programs

Currently the entire core image is built and kept in memory. In order to support very large programs, it may be necessary to avoid building a core image. This can be accomplished by using a two pass algorithm. The first pass would be interested in SOM, SEG, ESD, ESR, and EOM records and assigns addresses to all of the segments and symbols resolving ESRs. The second pass would be interested in TXT, RLD, LST, and SIN records and places the TXT in the assigned locations, performs RLD's, and outputs the resultant module.

There are two alternative methods of implementing such a two pass algorithm, saving information in a spill file and reading the input twice. If the input is read twice, the LINKER should not have to prompt for file names twice, so the second alternative should only be used if the LINKER maintains the list of file names. The tradeoff is whether using a spill file will involve fewer I/Os than re-reading the input files, as well as the storage space used by the spill file.

The algorithm for using a spill file is based on the partitioning of the input into modules and segments. The amount that has to be kept in memory at any given time consists of one segment. If the input file is in sorted order, then the TXT and RLD records for a given segment will appear together. Therefore, when the TXT and RLD records for a given segment have been completely read in (as signified by encountering records for another segment), the TXT and RLD information for the current segment can be written to the spill file. For unsorted modules, the TXT and RLD records for different segments can be interleaved, so the writing to the spill file must be postponed until the EOM record is encountered, and the TXT and RLD information would be written to the spill file by segment. During the final output phase of the LINKER, the TXT and RLD information would be read in, processed and RLDs performed, and output, all one segment at a time. LST and SIN records would be written to a spill file in a similar manner, but a separate spill file for these records would probably make matters easier.

The algorithm for reading the input twice involves two levels of sophistication. The simple method is simply to ignore TXT, RLD, LST, and SIN records during the first pass and to read them during the second (along with the other records). The second method works only if the module is sorted and there is only one module per input file. This method stops reading a file as soon as a TXT, RLD, LST, or SIN record is encountered. Note that during the second pass it is useful to read and process all of the record types to verify consistency with the internal data structure and to ensure that the files did not change between the two passes. The base index number values for a module would have to be reset in order for the second pass to assign the same global index numbers during the second pass as it did during the first pass.

4. Polish Fixups

Currently the only RLD operations provided are 30 bit add and 30 bit subtract. In order to reduce the complexity of compilers and assemblers, it would be useful for the LINKER to support polish fixups in the RLD. There are several things that are necessary in order to implement this change. The order of RLD records would have to maintained throughout the LINKER. All of the RLDs affecting a given S-1 word would have to be done at the same time, instead of having some done during the input phase and some done during the output phase when all symbols are known. Additional RLD operations would have to be defined to place values on the RLD stack. The semantics of existing operations would be changed to stack the value and then perform the operation on the top two values of the stack.

The manner in which RLDs would be performed would be extensively affected. During the output phase of the LINKER, the RLDs for a given text block would be processed. The RLDs would be separated into a list for each S-1 word. The RLDs for each S-1 word would then be performed in order utilizing a stack. The stack would initially get one value, the value of the S-1 word specified on the TXT record. Whenever the stack were popped so that only one value remained, that value would be stored into the text block, the RLDs performed for this word would be discarded, and the remaining RLDs would be performed. If an RLD were encountered that

couldn't be performed (e.g., it referred to an unresolved ESR), RLD processing for this S-1 word would be aborted, and RLDs for this S-1 word not discarded (including those performed which altered the stack, but excluding those which were discarded when the stack was popped and only one value remained) would be retained for output. When all RLDs for a given S-1 word are performed, the stack would be expected to contain only one value. Otherwise, an error message would be produced, and RLDs for this S-1 word not discarded would be retained for output.

5. Reallocation of Memory to Segments Between Pass One and Pass Two

Currently, the method of assigning memory to segments is rather random. Segments are assigned memory locations in the order that their SEG records are encountered. Segments are put on separate memory pages if their memory access modes differ. It may be desirable to put segments in a different order. To do so, the reassignment of memory to segments can be done between pass one and pass two (the input and output phases). The field SEG_DISPLACEMENT should be set to the amount that the segment is moved by during this reassignment. Since TXT and RLD blocks only store the offset from the start of the segment, no change need be done for these record types. For ESD and LST records, the actual address is maintained, so the appropriate write routine (in write LDI record class) would have to add SEG_DISPLACEMENT to the address in ESD and LST record.

IX. Bibliography

HaH79 Hailpern, Brent and Bruce Hitson, "The S-1 Assembler - FASM" in "S-1 Multiprocessor Architecture"; S-1 Internal Document SMA-3, in revision.

KeW78 Keller, Arthur and Gio Wiederhold, "S-1 Intermediate Loader Format"; S-1 Internal Document LDI-8, 27 November 1978.

OKK78 Olumi, Mohammad, Javad Khakbaz, and Arthur Keller, "S-1 Loader Format"; S-1 Internal Document LDF-5, 13 January 1978.

Rub78 Rubin, Jeff, "FSIM Simulator"; S-1 Internal Document FSIM-0, 8 September 1978.

WaG78 Wall, David W. and Erik J. Gilbert, "SOPAIPILLA Maintenance Manual"; S-1 Internal Document SOPADOPE-1, 23 March 1978.

WiH78 Wiederhold, Gio, and Brian Harvey, "Initial Layout Description of OS-zero"; S-1 Internal Document OS0-4, 25 February 1978.