Draft for Approval
Published 3/1/66

Identification

EPLBSA, Bootstrap Assembler for EPL
John William Poduska

Purpose

The EPLBSA assembler was designed and built to translate the output of the EPL compiler (7094 and 635 version) into the Multics standard binary format of a TEXT segment and a LINK segment. The assembler is provided in lieu of the original BSA mainly to allow the full set of ASCII characters and character set conventions.

The assembler also provides the system programmer with an alternate means of coding those programs which simply cannot be done with EPL. For such programs, the lack of restrictions on the input stream becomes important so that EPLBSA offers some advantages over BSA. The user of EPLBSA will however find some annoying restrictions such as the lack of macros and listing control pseudo-operations.

The primary design objectives of EPLBSA were that the assembler be operational very quickly, and that the restrictions of BSA be removed. Secondary objectives were that the speed of operation be improved and that the object code (linkage mostly) be more efficient than that offered by BSA.

To this end the assembler was written in FORTRAN-IV and operates as a standard activity on the 635. Furthermore, all tables (some 30 in total) are arranged as list structures manipulated by subroutines very similar to the old FLPL language (Fortran List Processing Language - vintage April 1960). Surprisingly, the use of FORTRAN-IV and list structured table is inefficient in neither time nor space.

Finally, the interim nature of the assembler and the close production date imply two important things:

1.) Advanced assembler features such as macros, multiple location counters, and listing control pseudo-operations, etc., are not included.

2.) The assembler must operate under compromised sets of input and output characteristics; e.g., there are three statement terminators on input 'semi-colon' (073), 'new-line' (012), and 'carrier-return' (015); and on output the segment name is imbedded in the segment text making the renaming and binding of segments impossible.

Where such compromises may have bearing, a remark will be made parenthetically.

The following portions of this section are divided into 5 parts as follows:

1.) Overall Specifications
- Input
- Identifiers
- Statement Format
- Output
- Features Allowed and Disallowed

2.) Segmentation Features
- General
- Base Register Conventions
- Intersegment Addressing Modes
- System Macros
- Linkage Information Generated

3.) Details of Machine Instruction Statement
- Internal Symbols
- External Symbols
- Internal Expressions
- Asterisk as a Symbol
- Boolean Expressions
- Literals

4.) Description of the Pseudo-Operations
- Control Pseudo-Operations
- Symbol Defining Pseudo-Operations
- Data Generating Pseudo-Operations
- Storage Allocation Pseudo-Operations
- Intersegment Communication Pseudo-Operations

5.) Operation of the Assembler
- Operation of Pass 1
- Operation of Pass 2
- Operation of Post Processor
- The Assembly Listing

1. Overall Specifications

Input

The input to the assembler is a character stream without card boundaries or line marks. Characters are 7 bit ASCII code, embedded in 9 bit sub-fields 4 per 36 bit word. Statement termination is by 'semi-colon' (073), 'new-line' (012), or 'carrier-return' (015).

(In reality, the assembler receives its input as column binary card images with 22 usable words per card. While the

normal user will never be aware of this feature, it does mean that 7-punched ASCII format files are directly useable for card input to the 635.)

Both the 'null-idle' (000) and 'null-delete' (177) characters are completely ignored on input; and the 'relative horizontal tab' (021) and 'relative vertical tab' (023) are recognized and create the proper number of blanks or new lines.

Identifiers

Named identifiers in EPLBSA consist of 31 (or fewer) characters drawn from the set:

(26 upper case letters, 26 lower case letters,
10 digits, 'period', 'underline')

Furthermore, the identifier must begin with one of the 52 alphabetic characters; and there may be no blanks nor any other break imbedded within an identifier. (Note that 'period' is chosen as an EPL substitute for 'underline'. Also, a later modification may allow 'underline' as an initial character.)

Statement Format

A statement in EPLBSA consists of number of fields separated by separation characters and terminated by a line terminator. The format has been termed 'free-field' which means in essence that the blank is a special separator: any number of blanks is one separator only and any separator followed by any number of blanks is usually only one separator. (Note that as a separator, 'tab' (011) is treated as a number of blanks; not true in character fields.)

The general statement format is as follows:

$$\left\{ \langle \text{loc. id.} \rangle : \right\} * \rightarrow \langle \text{op. id} \rangle \rightarrow \langle \text{var. field} \rangle \rightarrow \langle \text{comment} \rangle$$

The graphic ' $\rightarrow$ ' is taken to imply separation of fields meaning a separation character (perhaps blank, tab, etc.), followed by any number (perhaps zero) of blanks or tabs. The fields are given the following meanings:

1.) There may be any number of location field identifiers each of which is immediately followed by a 'colon' (072). These identifiers are uniformly defined as internal symbols having the current value of the (current) location counter. (Note the implication for EQU, BFS, and REM statements.)

2.) The operation field identifier specifies the action to be accomplished by the statement involved. The names of all instructions and pseudo-operations happen to map into the 6 bit GE-BCI code and this fact is used by the

assembler. The letters involved may be any mixture of upper and lower case with the same result. (A later macro-implementation may alter this description).

3.) The variable field has a meaning dependent on the operation field. For normal instructions, the variable field specifies the address, the modifier, bit 29, and the base register (if any). Furthermore, the variable field must be written without blanks except for some character strings.

4.) The comment field contains any arbitrary string of characters exclusive of statement terminators. In the special case of a statement which expects a variable field, but the variable field is void; the comment field must be preceded by a 'double-quote' (042).

Output

The output of the assembler consists of the following items: 1) a Text file, 2) a Link file, 3) a Listing file, 4) an Error file, and 5) an assembly listing on the 635 printer. All files except the listing on the 635 printer are returned to the 7094 in the normal fashion of the MRGEDT system.

The Text file contains the instructions coded, literals, and the invariant part of the linkage information (names and such). The Text file is a pure transcription of what one expects to see for the text of the segment in core storage during execution.

The Link file contains a map of itself and five regions: 1) links and entries, 2) symbol table, 3) relocation information, 4) linkage map (for unlinking), and 5) binding information. Only information of type 1 is currently included and all other regions are void. (A symbol table and binding information may later be included. Also note that the Link file is not a transcription of what will later be the linkage segment.)

The Listing file and the Error file follow standard 6.36 format. The listing file contains a standard assembly listing with all generatives and pseudo-ops expanded and printed in "detail" mode. The error file contains comments about the progress of the assembly and any drastic error comments. All error comments are preceded by the identification ".EPLBSA."

The 635 Listing is a more edited printed output. In particular pages are titled, dated, and numbered. In addition, tabs appearing in the input stream cause escapement of the output to a tab point spaced every 10 blanks. Furthermore, the ASCII set is mapped into GE-hollerith as closely as possible with unintelligble characters printed as '¿' and hardware escapes on the PR-20 accounted for.

Features Allowed and Disallowed

A partial list of the features specifically offered by EPLBSA is as follows:

1.) All machine instructions for the 645. Also, base register names are known to the assembler so that 'eap5' and 'eaplb' are both allowed and mean the same thing.

2.) Most generative and storage allocating pseudo-ops.

3.) Most forms of VFD and literals including DU and DL modifications.

4.) All forms (including TEMP and TEMPD) of intersegment communication.

Some of the important features currently disallowed are as follows:

1.) Most special formats for tally words, repeat instructions, character and byte manipulating instructions, and all I/O instructions. The pseudo-ops ZERO and ARG are allowed.

2.) All macro and macro-related operations.

3.) Multiple Location Counters.

4.) Most Listing Control (e.g., TTL) pseudo-operations.

(Many of the disallowed features are under cosideration for eventual inclusion.)

2. Segmentation Features

General

One of the most interesting features separating the current assembler from more conventional ones such as FAP or GMAP is its ability to deal with intersegment communication problems of the Multics system. The assembler is able to properly handle all new instructions and pseudo-operations for manipulating base registers, generating external segment references, etc. In addition, all new types of indirect modifiers (esp ITS and ITB) are available.

Base Register Conventions

Standard base register pairing and assignment is assumed and will be compiled by the assembler. Standard base register assignment is as follows:

ap=0	bp=2	lp=4	sp=6
ab=1	bb=3	lb=5	sb=7

The mnemonic and the numeral for specifying a base may be used interchangably throughout the assembler.

Intersegment Addressing Modes

The variable field of instruction and some pseudo-operatons specifies the address and modifier for the word(s) assembled. The address may refer to internal (within this segment) or external (not in this segment) locations and can take on one of six general forms as follows:

1) op <seg>|[xsym]±inexp,mod
2) op <seg>|inexp,mod
3) op base|[xsym]±inexp,mod
4) op base|inexp,mod
5) op xsym±inexp,mod
6) op inexp,mod

where <seg> is a segment name in pointed brackets, [xsym] is an external symbol (defined in some other segment) in square brackets, inexp is an interval expression composed only of symbols and operators defined locally, base is any absolute symbol, and mod is any legal (or defined) modifier.

Address types 1, 2, and 3 are the full-blown external reference types. The names <seg> and [xsym] are defined locally as segment name and external symbol respectively; these definitions are not carried beyond the current statement. The inexp must be connected to any [xsym] by a plus or minus sign but may be void, in which case the ± is dropped. The mod may be any legal modifier including the new 645 modifiers. These external references cause the local text word to be assembled as an indirect reference to the linkage segment, and the linkage segment contains an ITS pair (initially an FI until linked) pointing to the proper place.

Address type 4 is similar to the first 3 except that no further indirection is required. This is in fact the form that every external address eventually becomes.

Address type 5 is a special form in which xsym has been defined in this assembly as being external. If xsym appears in a Segref or Basref pseudo-op then the address will be compiled as an indirect link through the linkage segment. However, if xsym appears in a Temp or Tempd pseudo-op, then the reference is to

some point in the stack.

Address type 6 is a normal internal expression consisting of internal symbols and constants and the operators + - * and / with the usual meaning. Nesting of subexpressions is allowed using parentheses and the depth is unlimited (except by table overflow).

System Macros

A macro facility is not provided with EPLBSA but certain system macros are required including CALL, ENTRY, RETURN, and SAVE. These macros are implemented as pseudo-operations in the assembler and generate code as specified by section BD.7.02. (these macros are not currently implemented for Mastermode or Executeonly programs.)

Linkage Information Generated

The EPLBSA assembler will generate a linkage file containing information in regions as dexcribed in BD.7.01. However, only region 1 containing links and entries will be non-void. No relocation, symbol table, binding, or unlinking information will be put out; i.e., those regions will be void. (Symbol table and binding information may later be added to the linkage file).

4. Details of Machine Instruction Statements

Internal Symbols

Internal symbols are those identifiers given meaning only within the current procedure. These symbols are defined in one of two ways:

1.) Appearance in the location field(s) of any statement.

2.) Appearance as the first subfield of one of the pseudo-operations equ, bool, link, bss, or bfs.

Every internal symbol used in a program must be defined precisely once; the assembler will indicate an error for use of an undefined or multiply defined symbol. (The symbols are classed as to absolute, relocatable, bool, etc., in the assignment table but no use is currently made of this information.)

External Symbols

External symbols, i.e., symbols representing locations in other segments, may appear in the variable field of instructions in one of two forms: by an identifier specifically defined as external, or by a special construction for a local definition. The two local constructions allowed are:

<seg>|[xsym] or base|[xsym]

The portion of the structure to the left of the vertical line is a segment name if in pointed brackets; otherwise it is either a numeric base number, a symbolic base (e.g., sp), a symbol defined as a base (by base), or an ordinary internal symbol. The [xsym] is a symbol defined in the segment specified by <seg> or base; if void it is taken as zero.

Normal identifiers defined as external symbols may be defined by the Segref, Basref, Temp, or Tempd pseudo-ops. If such an external symbol is used in the variable field of an instruction, it must be the first identifier, preceded by no operator and followed only by + or - or one of the terminators. The same identifier may be assigned to an external symbol and an internal one; the assembler determines which definition to use by context.

Internal Expressions

Internal arithmetic expressions are used in the assembler to specify an offset or address to an instruction or pseudo-op. These internal expressions are formed from internal symbols, decimal integer constants, the operators + - * and / and parentheses used as delimiters. Evaluation is performed according to the normal rules of algebra with nested subexpressions (delimited by parentheses) allowed. The expression is calculated to 36 bits and then truncated to the accuracy required. An internal expression is terminated by a blank, comma, statement terminator, or a (preceded by a symbol or number. When an internal expression is terminated as many)'s as necessary will be appended to complete the expression.

Asterisk as a Symbol

The asterisk (*) when placed in the position of a symbol in an internal expression is evaluated with the meaning of 'the current value of the location counter', i.e., the value at which any location field symbol would be defined. There is no ambiguity between the use of asterisk as a symbol and asterisk as an operator since the operator is always binary.

Boolean Expressions

The assembler also accepts expressions to be interpreted as Boolean Expressions where the meaning of the operators is then:

*	and
+	or
-	exclusive or
/	unary not

If the / is encountered as a binary operator it is treated as the combination */ (note difference from original BSA).

Literals

A literal in the variable field is a specification of a data operand rather than the location of the data. Literals may appear only as the first item in the variable field of an instruction; the literal is specified by an 'equal sign' (=) and the data immediately follows.

The assembler accepts four types of literals: decimal, octal, ascii, and vfd. (Address pair and an instruction literals are possible improvements.) The literals are pooled at the end of the program before linkage information, and duplication is avoided whereever possible. Multiple word literals always begin on an even word.

Literals may be modified by the DU or DL modifier, in which case the literal is not pooled but is truncated to 18 bits and inserted in the address portion of the instruction. If the literal is floating point, fixed point (not integer), or ascii, the leftmost 18 bits of the first word are used; otherwise the rightmost 18 bits of the first word are used.

The general forms of literals allowed is as follows:

1.) Decimal: integer, fixed, floating, and double-precision with the usual GMAP conventions regarding 'point', B, E, and D modifiers.

2.) Octal: 12 (or fewer) unsigned octal digits, literal specified by 'letter O' following the equal sign.

3.) Ascii: two forms as follows

=a ⟨four characters⟩

=na ⟨n characters⟩

In the second case, n must be from 1,2,3, or 4 and the literal is filled with nulls.

4.) Variable Field Literal: usual form of variable field definition with decimal, octal and ascii subfields. Literal is specified by the 'letter V' following equal sign. (Note that 'comma' cannot terminate this literal.)

5. Description of the Pseudo-Operations

The following is a brief description of the pseudo-operations currently available in EPLBSA. For the most part only the novel features of these pseudo-ops are described, and the

usual conventions apply to those features left undescribed.

Control Pseudo-Ops

END - End of input stream
EVEN - Force location counter even
FILE - Gives CTSS name (peculiar to 6.36 system)
NAME - Segment name of procedure
NULL - Void statement
REM - Remark (same as NULL)

Symbol Defining Pseudo-Operations

BASREF - Definition of external symbols relative to a base register. The format is as follows:

```
basref     base,s1,s2,s3(call(arg)),s4,...
```

The symbol s1 (or s2,...) is then defined as external and any reference to it as the first symbol in the variable field is equivalent to base s1 . A trap routine to be called before linking can be specified as shown for s3; here the call is the routine to call and arg specifies the argument list. The call and arg can be used to specify internal or external references.

BOOL - Define a symbol with an equivalence given by a boolean expression. The format is

```
bool       boolsym,boolequiv
```

where boolsym is the symbol to be defined and boolequiv is a boolean expression giving its equivalence. If the symbol cannot be defined in pass-1, then an attempt is made to define it in pass-2.

EQU - Define a symbol with an equivalence given by a normal internal expression. Operation is comparable to BOOL.

LINK - Define an internal symbol as the link number of an ITS pair in the linkage section. The format is

```
link       linksym,generaladdress
```

where linksym is the symbol to be defined as the link number of the generaladdress. The generaladdress is any form of external address including any modifier. The link pseudo-op allows the sequence

```
lda    <seg>|[name],mod
```

to be represented as

```
link    1, <seg>|[name],mod
lda     lp/1,*
```

and is useful when one doesn't want the * on the latter lda.

SEGREF - Defines an external symbol relative to an external segment name. The format is as follows:

```
segref    segname,s1,s2,...
```

and use of s1 (or s2,...) is equivalent to segname s1. Trap pointer conventions are identical to those for Basref.

TEMP - Defines symbols to locations in the stack relative to sp. The format is

```
temp    s1,s2,s3(inexp),s4,...
```

The symbols are assigned to sequential locations. The length of the block may be defined by an expression in parentheses.

TEMPD - Defines symbols as locations of word pairs in the stack relative to sp. The format and operation is the same as temp with the understanding that each pair is assigned to an even location, and any block length receives twice that many words.

Data Generating Pseudo-Operations

ACC - Generate data consisting of ascii characters quoted by 'single quote' ('). A count precedes the character string in character position one, and a single quote can be imbedded by use of two consecutive single quotes in the string. A maximum of 9 data words (not 8) may be generated with acc. Any partial words are filled with nulls (000).

ACI - Same as ACC but without character count.

DEC - Normal decimal data generator. Integer, fixed, floating, and double precision data may be generated with the 'point', E, B, and D characters having the usual meanings. The first blank terminates the variable field.

OCT - Normal octal generator except that no signs are allowed, and a blank terminates the variable field.

VFD - Define data words under a variable field format. Same as normal VFD except that the H modifier is replaced by A for ascii.

Storage Allocation Pseudo-Operations

BFS - Block followed by symbol. Similar to BFS in GMAP except that format is

bfs symbol, length

BSS - Block started by symbol. Similar to BSS in GMAP except that format is

bss symbol, length

Special Word Format

ARG - Treated like an instruction with zero opcode.

ZERO - Generates one word with two specified 18 bit fields. The two fields are defined by interval expressions or literals.

Intersegment Communication Pseudo-Operations

CALL - Call subroutine with argument list. The format is

call name (arglist)

The name may be a full-blown external reference with modifier as may the arglist. The arglist may be void and name may be internal. Code generation is as described in BD.7.02.

ENTRY - Define the internal symbol in the variable field as an entry point. This pseudo-op causes considerable linkage information to be generated.

RETURN - Return to caller, code sequence as described in BD.7.02.

SAVE - Save conditions, code sequence as described in BD.7.02.

(Note that later inclusion of Mastermode and Execution pseudo-operations will cause these system macros to take on different meanings in different modes.)

6. Operation of the Assembler

Operation of Pass 1

The fundamental purpose of pass-1 is to define all Internal Symbols, i.e., to enter them into a table along with their value. In addition, all literals should be accounted for, pseudo-ops and macros expanded, and possible phase errors accounted for. To

accomplish these tasks, pass-1 reads the input stream, analyzes ops and pseudo-ops, and keeps track of an internal Program Counter by which internal symbols are assigned. Symbols appearing on SEGREF, ENTRY, etc. cards require special entries into some of the other tables, but it is not necessary in pass-1 to take account of a local external reference such as LDA ⟨X⟩|[Y]

Phase errors are detected and general synchronization is maintained by recording (in a list) the PC at the end of each statement. No collation or intermediate 'tape' is written and pass-2 will simply re-read the input stream. At the end of pass-1, the literal origin is established for pass 2 assignment, but the linkage information origin is not set since the length of the literal table is not known (because of VFD literals).

Operation of Pass 2

The fundamental purpose of pass-2 is to generate the text (binary output) of the program. This is accomplished by rereading the input stream, analyzing the ops and pseudo-ops, and keeping track of the Program Counter. Pseudo-ops are handled according to whatever action they require. Normal operations are assembled by combining the results of a variable field analysis, a modification field analysis, and a lookup of the binary operation codes.

The variable field analysis is by far the trickiest and may require table entries for external symbols and associated linkages. In addition, literals cause some problem especially those with DU or DL modifiers. Literals are assigned to locations as required, and link numbers are defined (in the link table) as used.

The output of pass-2 consists of an assembly listing and the binary text with literals. The literals are put out after the END card is encountered; then the origin of the linkage portion of the text is defined and the post processor is called.

Operation of Post Processor

The post processor is called after pass-2 to generate the linkage portion of the text file, and then to generate the linkage file. The linkage portion of the text consists primarily of symbolic names and pointers put out from table information in the following order:

1. External Symbol Definitions (Entry Points and Segdefs)
2. Segment Names
3. External Symbols

4. Trap Words

5. Type Pairs

6. Internal Expression Words

The order in which the information is put out is important, because as each piece of information is 'punched' its location is entered into some table. For this reason, Trap Words are put out before Type Pairs because the Type Pair points to the Trap Word.

After the text file is completed, the linkage file is written. This is fairly easy because all locations refer to the text segment relative to the origin of the linkage information. The assembly is completed with the writing of the linkage file.

The Assembly Listing

The listing provided by EPLBSA is the usual type of assembly listing as provided by FAP or GMAP. For each statement one or more lines is printed consisting of error flags, location, assembled word, and input statement. After the normal text is listed the literals and linkage information, and following the text segment is listed the linkage file.

The 635 printer listing is similar to the listing file returned except that all characters are mapped into GE-BCI. Any ascii characters which cannot reasonably be mapped are printed as the graphic '&'. Tabs are interpreted to cause escapement in multiples of 10 positions.

There are 8 possible error flags which are defined as follows:

U	use of an undefined symbol
M	use of a multiply defined symbol
P	phase error, something (probably BSS or BFS) has caused the program counter to be off
E	error in some field of a data generator (including literals)
F	field error, variable field is improperly constructed
T	error in address modifier (tag)
O	illegal operation code
S	error in definition of some symbol

The assembler will also complain of a fatal error if any of the error flags A, M, P, or O appear anywhere.