November 26, 1969

TO: R. Beatty
C. T. Clingen
F. J. Corbató
R. C. Daley
P. Green
R. Gumpertz
J. D. Mills
J. H. Saltzer ✓

FROM: N. Adleman

SUBJECT: An EPLBSA - compatible assembler

REFERENCE: M0065 (Poduska, May 26, 1966)

1.0 INTRODUCTION

As one of the laboratory projects associated with the undergraduate seminar on Multics, the current EPLBSA assembler will be modified to be maintainable and extendable wholly within the Multics environment. The present assembler is written in (impure) GE-635 FORTRAN IV and GMAP. Of the 37 routines that comprise the Version 3 assembler (which is awaiting a new daemon program), 27 modules are written in FORTRAN and the remaining 10 programs are written in GMAP. The GMAP code will be transcribed into EPLBSA code or, if deemed practical, the GMAP code may be transcribed directly into PL/I code. The FORTRAN code will be transcribed into PL/I.

The major objectives of this effort are to provide an assembler which has not lost any of the capabilities of the current EPLBSA program.

The goals of this project, in their order of importance, are as follows:

1. availability
2. maintainability
3. pure code
4. faster assembler (not necessarily, the first capability)

It is expected that the product of this effort will be an assembler that is completely compatible with the functional capabilities of EPLBSA.

This memo considers the additional design required to make the new assembler pure (and more efficient) within the Multics environment. The following issues will be discussed:

1. A general overview of the assembler's operation.
2. The management of the current assembler's large (42000+ words) data area within the assembler segment.
3. The disposition of the current assembler's labeled COMMON and other data (e.g., temporary data, table of op-codes, etc.).

2.0 OVERVIEW of the Assembly Technique

There are three major programs within the assembler. These are affectionately named PASS1, PASS2, and POSTP2. All other programs within the assembler fall into the category of utility (e.g., GLPL) or special purpose (e.g., DECEVL) in their contribution to the translation of a given source program. The first two major programs (PASS1 and PASS2) read the same source segment. For each object (binary) word or block of binary words, PASS1 stores the error flags for PASS2. PASS1 does preliminary processing on the pseudo-ops (e.g., CALL, USE, etc.), establishes an internal (to the assembler) symbol table, and, in general, maps out the core layout of the program that is being assembled.

PASS2 uses (and often duplicates) the information established by PASS1 and produces the text portion of the object segment and, optionally, the corresponding listing segments.

POSTP2 produces the LITERALS (of the object program) followed by all the system dependent data (e.g., definitions, linkage data, and symbol data) that permits an object program to operate within the Multics environment. As POSTP2 produces this data, the listing segment is also being completed with the corresponding printable information.

3.0 FREE STORAGE MANAGEMENT

Within the assembler segment (which is 64K words), a data area of 42000 words is used (for each assembly) to contain information about the object program. In this "42K" is placed, for example, the internal symbol table, ASCII names, error flags, linkage information, entry point information, etc. This data area is managed as a large list structured table by a group of GMAP routines under the general heading of GLPL for GE-635 List Processing Language (See Poduska M0065, page 4).

For the initial transfer, this "42K" will become a separate data segment that will be managed by a slightly modified GLPL (possibly in PL/I from the GMAP code) that need be created only once per process. All accesses to this data by other programs of the assembler (e.g., the LOC function), must be made aware that this data is no longer within the same segment (originally a GECOS/GE-635 core load) but in a separate and distinct segment.

The present GLPL initializes the entire data area of the assembler for each assembly. Consideration in the new GLPL should be given to the possibility of initializing blocks of the list table (perhaps in 1024 word increments), on an as needed basis while the assembly is proceeding. This would eliminate some unncessary page "touching" during an assembly. As an alternative to this, the assembler (GLPL) could reinitialize only that portion of the data area which was used by the previous assembly. This problem of modifying GLPL is quite delicate within the assembler and should be done when a transferred assembler warrants this type of tuning.

4.0 LABELED COMMON and other data

Having examined the problem of converting BLOCK (LABLED) COMMON to Multics, I believe the best strategy is the simplest. By using EPLBSA code, a data segment can be constructed that contains pure data in the text portion and impure data in the linkage portion. This data, both pure and impure, can be accessed by other assembler programs, by assigning the data via SEGDEFs and JOINing them into the text or linkage.

Let's look at this a little closer, because this area is very important and quite open for discussion. First of all, all labeled COMMON in the assembler must be identified as to whether it is pure or impure and allocated, via EPLBSA code, in the data segment which has been arbitrarily named <eb_data_>. For example, let's look at two typical labeled COMMONs of the assembler, VARCOM (impure) and CONCOM (pure). The code in <eb_data_> would be something like this:

```
                name                           eb_data
                              ≷

                use                            pure
                segdef                         concom
concom:         null
                bss                            concom_data, 121
                use                            pure
                segdef                         varcom_size
varcom_size:    dec                            21
                              ≷
                use                            impure
                segdef                         varcom
varcom:         null
                bss                            varcom_data, 21
                              ≷
                join                           /text/pure
                join                           /link/impure
                              ≷
                end
```

Each COMMON declaration would then be transcribed into PL/I code that represented an external structure in <eb_data_>. Each structure could be a single PL/I include file that be included in a program of the assembler at compilation time by a "% include" statement. The code produced by the PL/I compiler would then access this external structure properly.

Also included in <eb_data_> could be other major data of the assembler such as the 1024_{10} word op-code/pseudo-op character table (aligned on a page boundary, of course), conversion tables, etc. Smaller data items, such as variables local to a particular subroutine, could remain in the stack frame for that program.

5.0 SUMMARY

This memo does not attempt to resolve all the difficulties associated with transferring the assembler. Rather, the major areas were examined to relate the general flavor of the expected problem areas in the project.

It has been noted that it may be less expensive to design and produce a completely new assembler for Multics. I must disagree, especially in light of satisfying the objective of "availability" mentioned above. Furthermore, I believe that structurally and logically the assembly technique used in the assembler is quite sound. Once the initial transfer is completed, a sound, working assembler can then be extended, at will, in its proper environment under proper conditions.