MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

MCDONNELL DOUGLAS
CORPORATION

MCDONNELL DOUGLAS

META ASSEMBLER ENHANCEMENTS AND GENERALIZED LINKAGE EDITOR (CONTRACT NAS8-32570)

Final Report

JUNE 1979

MDC G8027

PREPARED FOR:

GEORGE C. MARSHALL SPACE FLIGHT CENTER
MARSHALL SPACE FLIGHT CENTER
ALABAMA 35812

PREPARED BY:

MCDONNELL DOUGLAS ASTRONAUTICS CO-WEST
AVIONICS CONTROL AND INFORMATION SYSTEMS
HUNTINGTON BEACH, CALIFORNIA 92647

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY-HUNTINGTON BEACH

5301 Bolsa Avenue, Huntington Beach, CA 92647

PREFACE

The final report for "Meta Assembler Enhancements and Generalized Linkage Editor" is submitted to the National Aeronautics and Space Administration, George C. Marshall Space Flight Center in accordance with the provisions of the contract number NAS8-32570. The report describes the results of the design and implementation of an enhanced meta assembler and generalized linkage editor to provide syntax responsive and target reconfigurable assembly, linkage edit and library creation and maintenance capability.

If any additional information is desired, please contact any of the following McDonnell Douglas or NASA representatives as appropriate:

- Mr. Z. Jelinski, Project Manager (MDAC)
  Huntington Beach, California
  Telephone: 714-896-5060

- Mr. K. V. Smith, Principal Investigator (MDAC)
  Huntington Beach, California
  Telephone: 714-896-2937

- Mr. Geoffrey C. Hintze, Project COR (NASA)
  Marshall Space Flight Center, Alabama
  Telephone: 205-453-5709

Page intentionally left blank

TABLE OF CONTENTS

		PAGE
	PREFACE	iii
1	INTRODUCTION	1
	1.1 Problem Statement	1
	1.2 Objectives	2
	1.3 Technical Approach	2
	1.3.1 Task 1 - User Oriented Syntax Definition Capability	2
	1.3.2 Task 4 - Meta Assembler Documentation	4
	1.3.3 Task 5 - Develop a Generalized Linkage Editor	4
	1.3.4 Task 7 - Meta Translator Installation and Training at MSFC	7
	1.4 Results	7
2	ADMINISTRATIVE DATA	11
	2.1 Team Organization	11
	2.2 Schedule/Milestones	11
	2.3 Facilities and Resources	11
	2.3.1 MDAC-W Huntington Beach, California	13
	2.3.2 NASA Marshall Space Flight Center, Alabama	13
3	TECHNICAL PERFORMANCE	15
	3.1 Meta Assembler Implementation	15
	3.1.1 Task 1 - User Oriented Syntax Definition Capability	15
	3.1.1.1 Assembler Level Language Definition Meta Language (ALLDEF)	17
	3.1.1.2 Assembler Level Language Lexical Meta Language (ALLLEX)	24
	3.1.1.3 ALLDEF Meta Language Processor	26
	3.1.1.4 ALLLEX Meta Language Processor	28
	3.1.1.5 Generalized Parser (ALTRAN)	28
	3.2 Meta Translator Implementation	28
	3.2.1 Meta Translator Description	28
	3.2.2 Meta Translator Application	31
	3.3 Generalized Linkage Editor	31
	3.3.1 General Overview	31
	3.3.1.1 Libratory Creation and Maintenance	33
	3.3.1.2 Binding of Modules	33
	3.3.1.3 Cataloging of Standard Meta Assembler System Outputs	33
	3.3.2 Flow Through the Generalized Linkage Editor	33
	3.3.2.1 Creation of a System Libratory for General Use	35

PAGE

3.3.2.2 Creation of a User Library and Load Module Generation 35
3.3.3 Use of the Generalized Linkage Editor 36
3.3.3.1 Directive Coding Conventions 36
3.3.3.2 List of Generalized Linkage Editor Directives 38
3.3.3.3 Use of the Library Creation and Maintenance Function 39
3.3.3.4 Use of Linkage Editor Function 46
3.3.3.5 Use of the Catalog Function 60
3.4 Installation and Training 64
3.4.1 Task 7 - NASA MSFC Delivery 64
3.4.2 Installation on the MSFC IBM 360 64
3.4.3 MSFC Installation Verification 65
3.4.4 Meta Assembler System/Meta Translator Demonstration 65
3.4.5 Demonstration for the NSSC-I 65
3.4.6 Personnel Training at MSFC 65
3.4.7 MSFC Deliverable Items 65
3.4.8 GSFC Deliverable Items 66
3.5 Meta Assembler Documentation 66
3.5.1 User Manuals 66
3.5.2 Detail Design Manuals 67

APPENDIX A SCHEDULE/MILESTONES A-1

APPENDIX B ALLDEF NSSC-I DEFINITION B-1

APPENDIX C ALLLEX NSSC-I DEFINITION C-1

Section 1
INTRODUCTION

McDonnell Douglas Astronautics Company-West (MDAC-W) has developed a Meta Assembler for NASA under previous contract efforts. Under contract NAS8-27202 the initial development of the Meta Assembler for the SUMC was performed. The capabilities included assembly for both main and micro level programs. Contract NAS8-30907 provided support to NASA MSFC during a period of checkout and utilization to verify the performance of the Meta Assembler. Under contract NAS10-8434 and NAS10-8833 additional enhancements were made to the Meta Assembler which expanded the target computer family to include architectures represented by the PDP-11, MODCOMP II, and Raytheon 706 computers.

1.1 PROBLEM STATEMENT

In spite of its usefulness, the system had some serious shortcomings namely the Meta Assembler used a language independent syntax for directives (pseudo ops), macros and labels because these features could differ greatly from one assembly language to another. For this reason, existing assembly language programs had to either have the source for these differences rewritten or a syntax preprocessor had to be written to change them. This put an additional burden on the user because in rewriting the source he had to substitute unfamiliar symbols for ones that he was used to. If a new syntax preprocessor had to be written he usually had to seek assistance from the program originator which resulted in additional costs and effort connected time delay.

Additionally, if a user desired to link together separately assembled modules, he was required to use whatever, if any, linking support tools were available for the target machine or write his own.

The above disadvantages provided serious obstacles to software standardization.

1.2 OBJECTIVES

The primary objective of this effort was to standardize a NASA low cost Meta Assembler and Linkage Editor. The enhancements to the Meta Assembler defined for this contract include: the design and development of a User Oriented Syntax Definition capability and the design and development of a recognition capability to support these definitions in order to perform the assembly process. Also, the design and development of a generalized linkage editor and library creation and maintenance function was defined.

The result of this effort resulted in the establishment of a Meta Assembler program and Linkage Editor program which operates in the environment of a large scale host computer and supports software development for flight and ground checkout computers (mini-computer class).

Additionally, user and maintenance documentation was developed and the inherent capabilities of the program demonstrated.

1.3 TECHNICAL APPROACH

The contract called for 7 major tasks to be performed.

Task 1 - User Oriented Syntax Definition Capability
Task 2 - Generalization of the Procedure Language
Task 3 - Improvement to the Meta Assembler Error Diagnostics and Dynamic Debug Features
Task 4 - Meta Assembler Documentation
Task 5 - Development of a Generalized Linkage Editor
Task 6 - NASA Goddard (GSFC) Delivery and Installation for the NSSC-1
Task 7 - Meta Translator Installation and Training at MSFC

Of these seven tasks, tasks 2 and 3 were deleted through renegotiation due to technical difficulty of task 1. Task 6 was deleted at the request of NASA and combined in part with task 7.

1.3.1 Task 1 - User Oriented Syntax Definition Capability

The existing Meta Assembler is designed to translate symbolic assembler level instructions into machine language instructions for a wide variety of target computers. The adaptability is achieved via a set of target definition directives which parameterize the Meta Assembler for the subsequent assembly function. The target definition directives supply the architecture characteristics (e.g., word size, register descriptions, character set definition) as well as the instruction set definition (mnemonic, operand description).

Additionally, the Meta Assembler has built in directives to perform assembly time functions (e.g., data definition, parameter definition, location counter control, listing control, conditional assembly control, procedure definition and expansion). The syntax processing of the Meta Assembler directives is fixed (e.g., DATA, PROC, EQU, ORG) and at the instruction processing level flexibility is provided for operand definition rather than syntax definition. Therefore, the Meta Assembler represents equivalency in its assembly function with a correlating target machine and assembler syntax compatibility is not maintained. This can have the effect of requiring programmers to learn the equivalent assembler language and directive syntax instead of using the familiar target assembler syntax. Additionally, maintenance of a program cannot be performed by both the Meta Assembler and the target machine assembler due to the syntactical differences.

This task alleviates the syntax incompatibility by providing the additional capability to allow the user to define the syntax of the assembler language and directives and the correlating semantics of the statements (e.g., generate intermediate language, perform an assembly time function). This was accomplished by designing a meta language for the purpose of defining assembler languages, their syntax and translation semantics.

The processors developed for this capability are the meta language processor, the lexical processor and the generalized parser. The meta language processor is a pre-assembly function which processes the meta linguistic definition of the assembler language and generates a dictionary data set containing the syntax and semantic tables to be utilized by the generalized parser. This function need not be performed for each assembly. The generalized parser performs the first pass of the assembly utilizing the syntax and semantic tables produced by the meta language processor. The first pass accomplishes the source statements translation into the Meta Assembler intermediate language which can then be processed by the existing second pass of the Meta Assembler to perform object module generation.

The design intent of this capability was not to replace the existing Meta Assembler target definition and first pass process but rather augment the Meta Assembler with the optionally invoked generalized parser function as illustrated in Figure 1. Host portability of the enhanced Meta Assembler was preserved.

Under this task a complete meta linguistic definition of the NSSC-I assembler language was developed. This represents part of the delivery items relative to Task 6.

1.3.2 Task 4 - Meta Assembler Documentation

A Detail Design Manual was produced which fully documents all subroutines and data areas of the Meta Assembler. This document is intended to support maintenance functions pertaining to the Meta Assembler. Included in the Detail Design Manual is an appendix devoted to host computer installation procedures.

The existing Meta Assembler User's Manual was updated to include the enhancements and modifications performed during this effort. Meta Assembler error diagnostics are listed with appropriate explanations as an appendix to the User's Manual.

1.3.3 Task 5 - Develop a Generalized Linkage Editor

A generalized Linkage Editor function was defined, designed, developed, and validated with appropriate documentation supporting each phase. It provides the capability to utilize modular programming techniques in the application of the Meta Assembler by combining a user library of separately assembled object modules, produced by the Meta Assembler, into an absolute or relocatable load module on a large scale host computer. Its primary processing capability is to perform relocation and external linkage functions on the object modules processed. To implement a system generation capability the Linkage Editor additionally may access object modules from an object module library to satisfy undefined global references (see Figure 2).

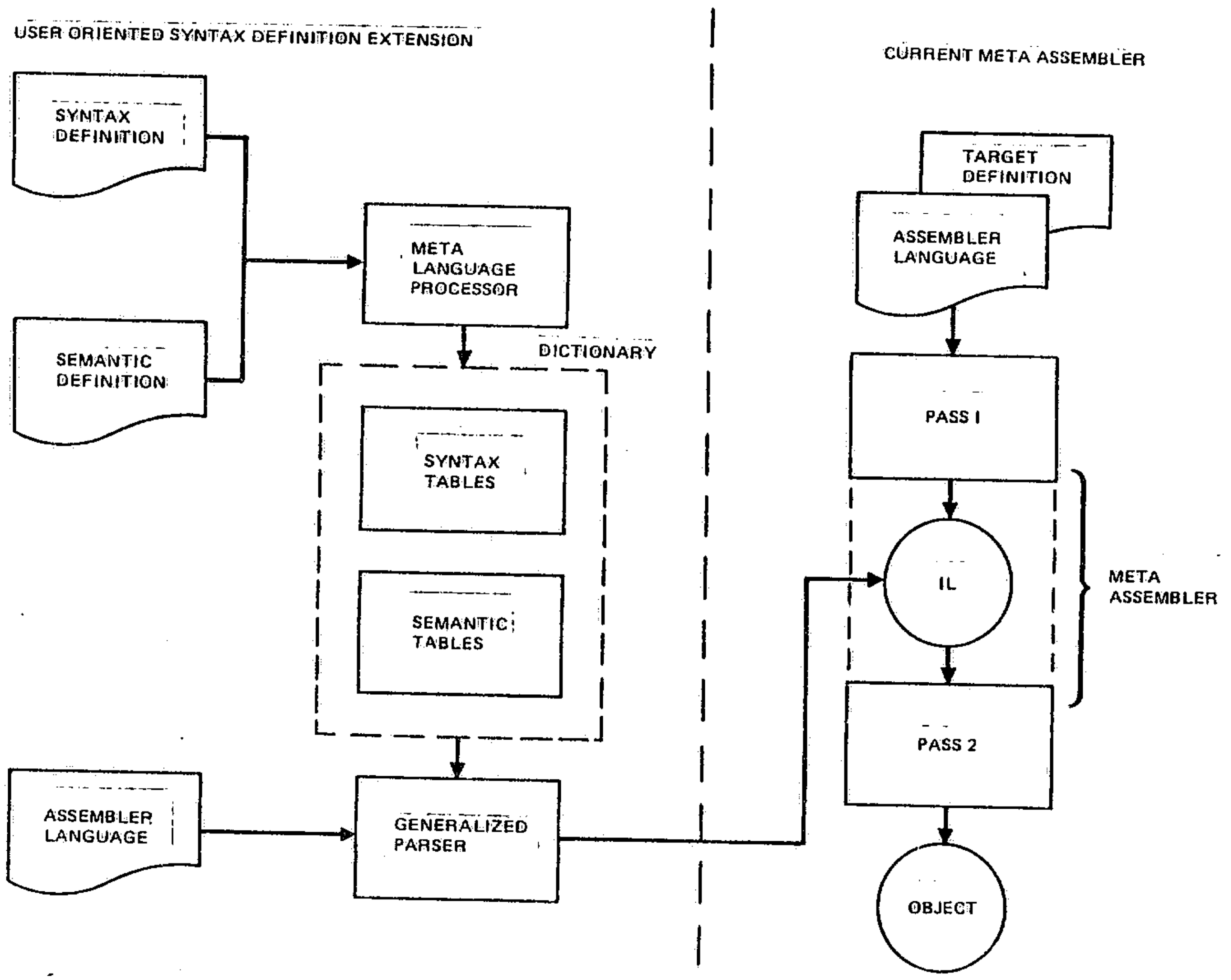
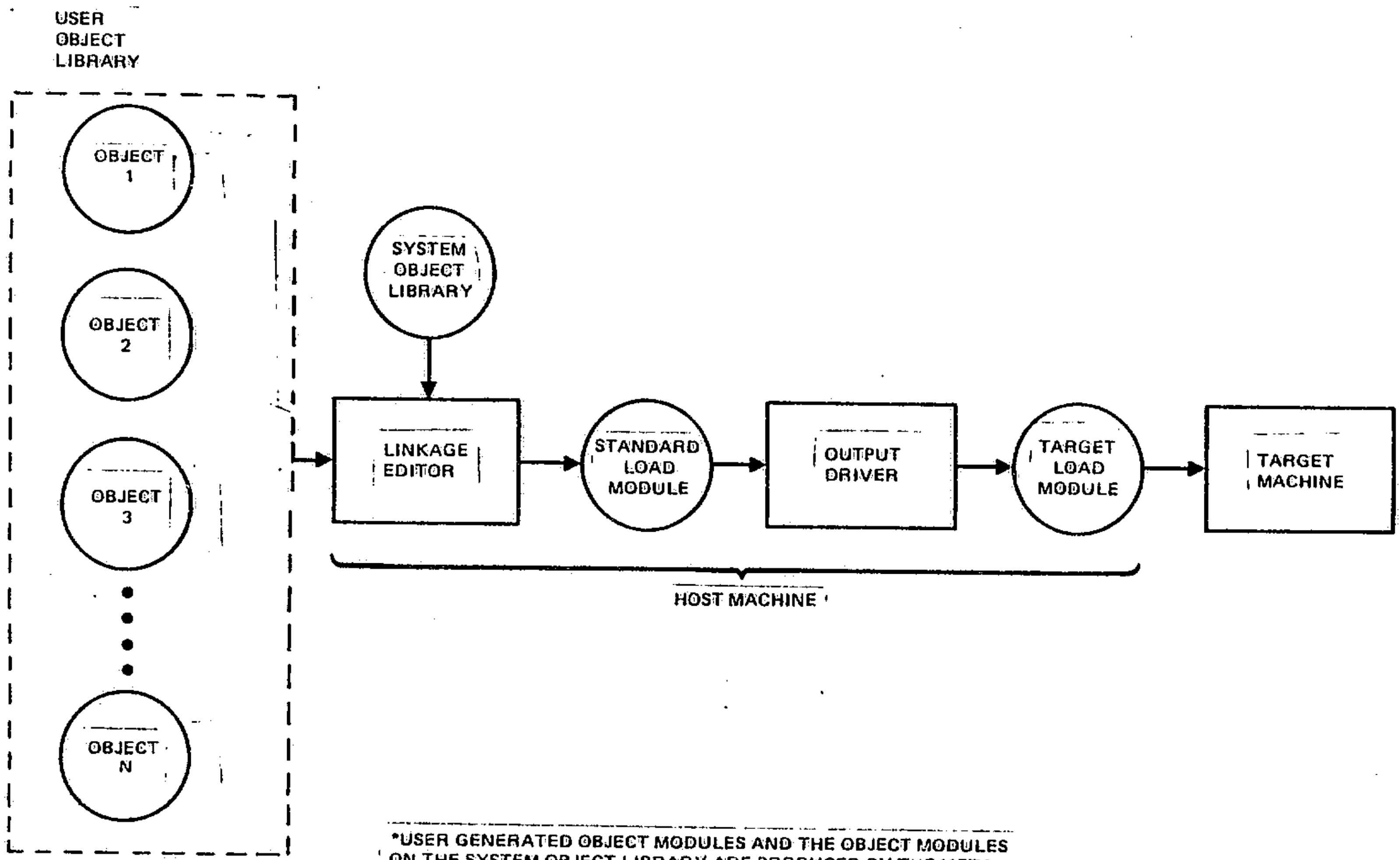


Figure 1 Meta Assembler Configuration



*USER GENERATED OBJECT MODULES AND THE OBJECT MODULES ON THE SYSTEM OBJECT LIBRARY ARE PRODUCED BY THE META ASSEMBLER

Figure 2 Generalized Linkage Editor

A critical aspect of the Linkage Editor will be its ability to respond to user defined parameters to fully utilize the resources of the target machine, specifically the NASA Standard Spacecraft Computer (NSSC-I). The resource parameters include the ability to optionally specify beginning addresses for some or all of the control sections represented in the object modules and to specify the order in which the control sections are to be loaded.

The implementation of the Linkage Editor is in ASA FORTRAN IV, as is the Meta Assembler, to facilitate ease in transporting the function from one host computer to another.

The absolute load module generation is in a standard format to maximize its applicability to a wide variety of target machines. This necessitates an output driver to be developed whenever a new target machine is interfaced. Under this task an output driver was developed to format the load module for loading and execution on the NSSC-I (see Figure 2).

1.3.4 Task 7 - Meta Translator Installation and Training at MSFC

For the exclusive purpose of maintaining the enhanced Meta Assembler,the MDAC proprietary Meta Translator was installed at MSFC on an IBM 360. This installation included the delivery of source programs (tape), program listings, technical documentation and installation procedure description for the MDAC Meta Translator, the enhanced Meta Assembler and the generalized Linkage Editor (see Figure 3).

Personnel training was conducted in the utilization of the Meta Translator.

In addition, the NSSC-1 language definition and output driver were delivered to MSFC. The GSFC furnished test cases were also delivered (see Figure 4).

1.4 RESULTS

The Meta Assembler was enhanced to allow the user to define an assembler language syntax to be processed. This capability eliminated source language reformatting or ad hoc syntax recognizer development in order to maintain compatibility with a target machine assembler language syntax.

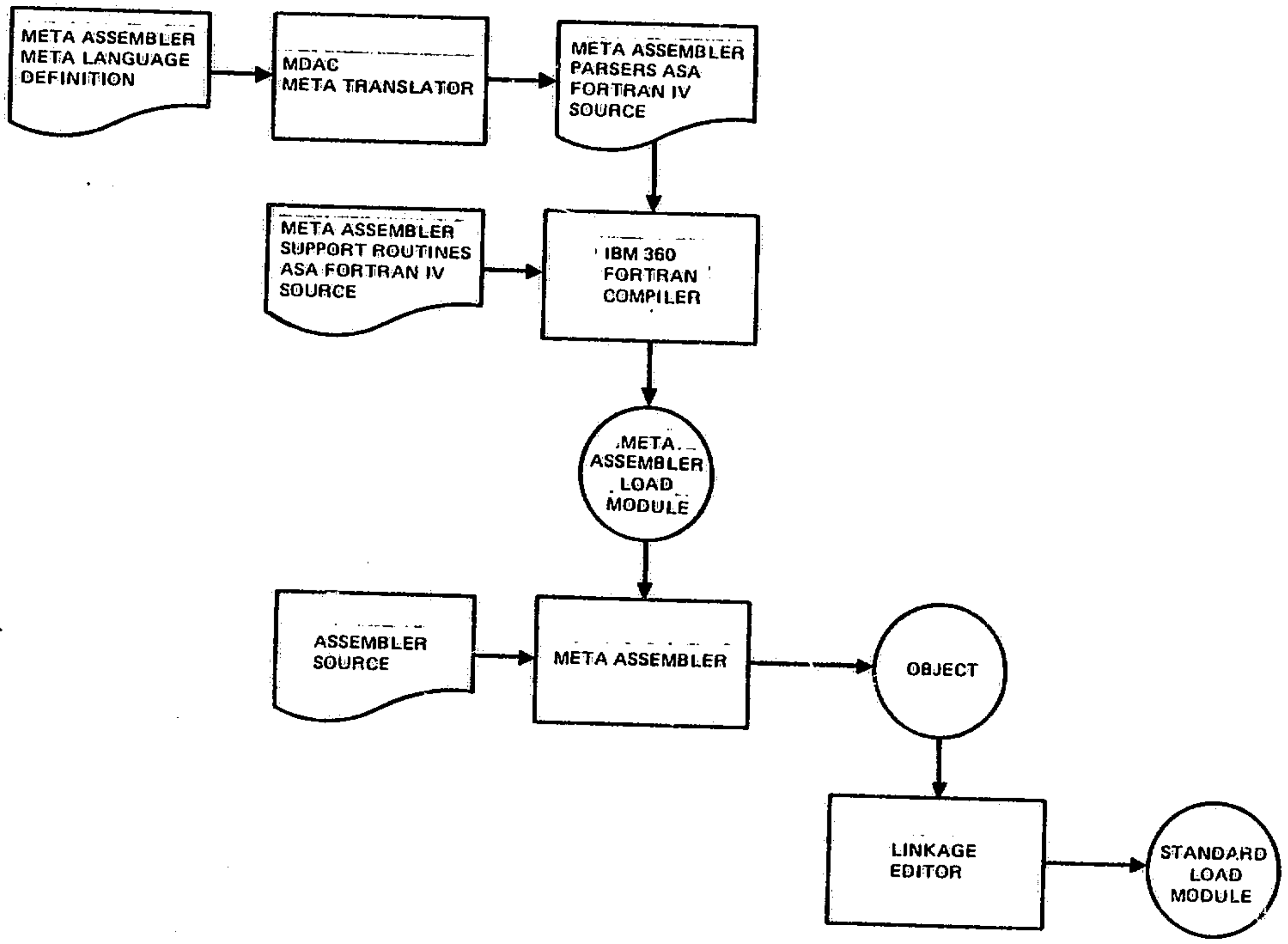

META ASSEMBLER META LANGUAGE DEFINITION
MDAC META TRANSLATOR
META ASSEMBLER PARSERS ASA FORTRAN IV SOURCE
META ASSEMBLER SUPPORT ROUTINES ASA FORTRAN IV SOURCE
IBM 360 FORTRAN COMPILER
META ASSEMBLER LOAD MODULE
ASSEMBLER SOURCE
META ASSEMBLER
OBJECT
LINKAGE EDITOR
STANDARD LOAD MODULE

Figure 3 MSFC Installation - IBM 360

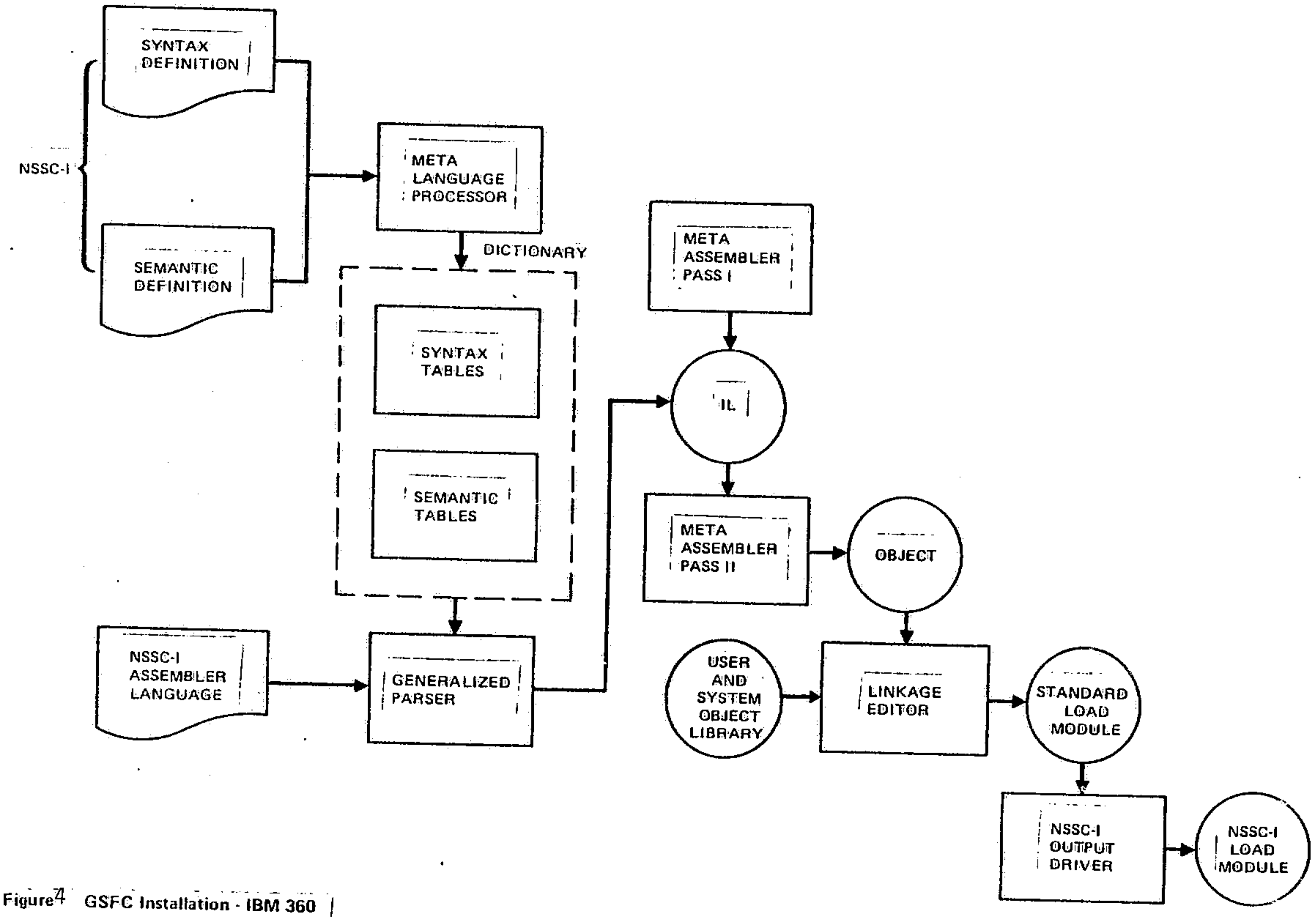


Figure 4 GSFC Installation - IBM 360

The original Meta Assembler was regenerated using the latest version of the MDAC Meta Translator. This regeneration provided an increase in efficiency, both execution time and memory requirements, and a more extensive dynamic debug capability.

These improved dynamic debug features will provide support in the maintenance of the Meta Assembler itself.

A generalized Linkage Editor was developed as a standard post processor for the Meta Assembler. The function of the Linkage Editor is to link separately assembled relocatable and/or absolute object modules into an absolute or relocatable load module. The Linkage Editor was written in FORTRAN IV to coincide with the host portability requirements of the Meta Assembler.

The NSSC-I computer was the initial target computer. The Meta Assembler and Linkage Editor were configured to accept NSSC-I assembler language syntax and produce load modules that fully utilize the NSSC-I resources.

The resultant Meta Assembler and Linkage Editor was installed at NASA Marshall Space Flight Center to facilitate centralized control of these NASA standard programs.

To provide NASA MSFC the capability to maintain the Meta Assembler the MDAC propreitary Meta Translator program was installed at NASA MSFC and training was provided in its use.

Section 2
ADMINISTRATIVE DATA

2.1 TEAM ORGANIZATION

The overall responsibility for this project was assigned to Avionics Control and Information Systems (ACIS), headed by Mr. G.A. Johnston, Director and was performed by the Computer Science Branch. ACIS is an organization of information scientists and engineers dedicated to research, design, analysis, and testing of advanced software concepts and to the development of computer applications for scientific and military use (see Figure 5)

MDAC Project Manager	-	Mr. Z. Jelinski
MDAC Principal Investigator	-	Mr. K. V. Smith
MDAC Technical Staff	-	Mr. J. B. Churchwell
		Ms. Soo Park
NASA COR	-	Mr. Geoffrey C. Hintze

The original principal investigator of the Meta Assembler Enhancements and Generalized Linkage Editor Project, Mr. A. J. Edwards, terminated employment with MDAC-W in January 1978. At that time, Mr. K. V. Smith was assigned the responsibility of principal investigator of this project.

2.2 SCHEDULE/MILESTONES

The schedule and milestones for the performance of the contract is contained in Appendix A.

2.3 FACILITIES AND RESOURCES

The development portion of this contract was performed at Huntington Beach, California, Headquarters of the McDonnell Douglas Astronautics Company-West (MDAC-W). The installation portion of the contract was performed at National Aeronautics and Space Administration, George C. Marshall Space Flight Center, Marshall Space Flight Center, Alabama.

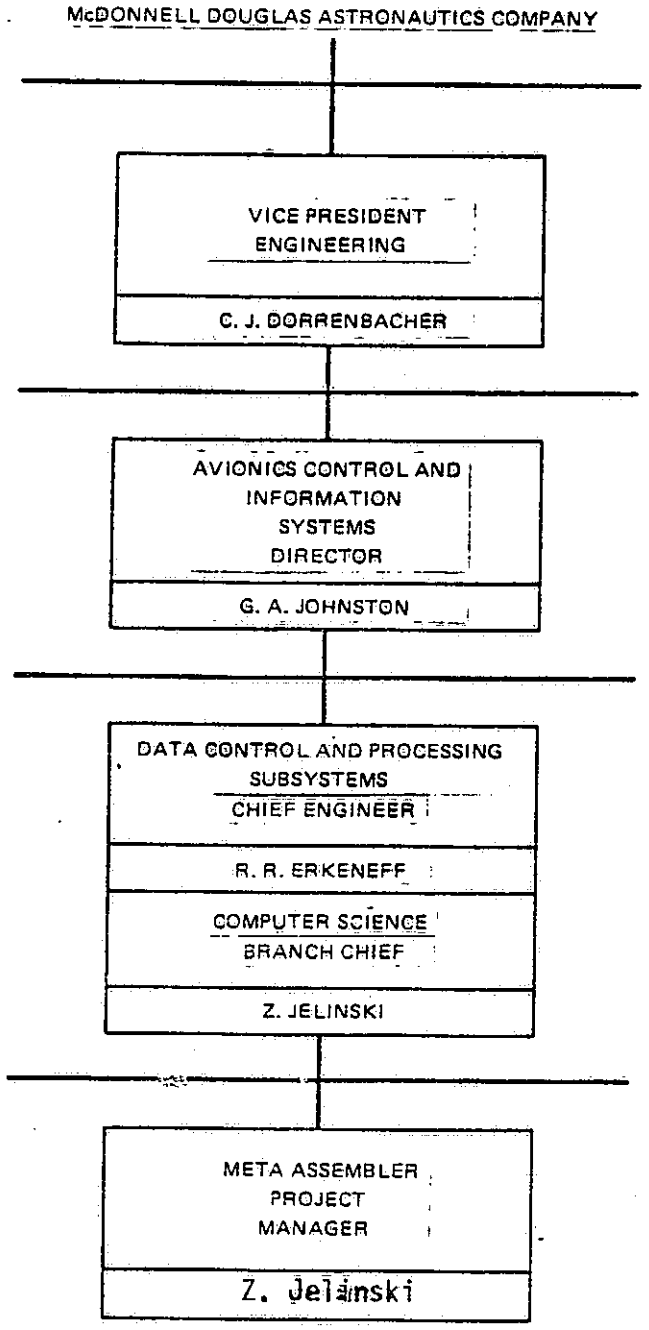

McDONNELL DOUGLAS ASTRONAUTICS COMPANY
VICE PRESIDENT ENGINEERING
C. J. DORRENBACHER
AVIONICS CONTROL AND INFORMATION SYSTEMS DIRECTOR
G. A. JOHNSTON
DATA CONTROL AND PROCESSING SUBSYSTEMS CHIEF ENGINEER
R. R. ERKENEFF
COMPUTER SCIENCE BRANCH CHIEF
Z. JELINSKI
META ASSEMBLER PROJECT MANAGER
Z. Jelinski

Figure 5 Position of Contract Within Company

2.3.1 MDAC-W Huntington Beach, California

The McDonnell Douglas Automation Company provided support to the project through the use of its facilities - the CDC Cyber 74 and DEC PDP-10 computers.

The MDAC-W proprietary Meta Translator was one of the primary support software products used in the performance of this project.

2.3.2 NASA Marshall Space Flight Center, Alabama

The host computer for the installation of the delivered software was the IBM 360 located in building 4708.

Section 3
TECHNICAL PERFORMANCE

3.1 META ASSEMBLER IMPLEMENTATION

This section contains the implementation results for the Meta Assembler extensions.

3.1.1 Task 1 - User Oriented Syntax Definition Capability

The purpose of this task was to provide a user oriented capability to syntactically define an assembler language, machine instructions and directives, enabling the Meta Assembler to maintain syntax compatibility with target computer assemblers.

The objective of this task was to integrate a meta language definition of an assembler language into the Meta Assembler technique such that the built-in semantic and support processing is available to the user at the meta language level. The built-in semantic and support processing is represented by:

- expression evaluation
- assembler directive processing
- intermediate language formatting
- object generation
- listing function

The implementation approach was to develop a meta language to define the assembler language syntax and correlating built-in semantic functions. This meta language is input to a stand-alone preprocessor for translation into syntax and semantic tables which will guide the first pass processing by the Meta Assembler. A generalized parser was developed, integral to the Meta Assembler, to perform the alternative first pass of the cross assembly. The output of the generalized parser is an intermediate language (IL) data set such that the existing second pass of the Meta Assembler can complete the cross assembly by converting the IL into the object data file and generate a program listing (see Figure 6).

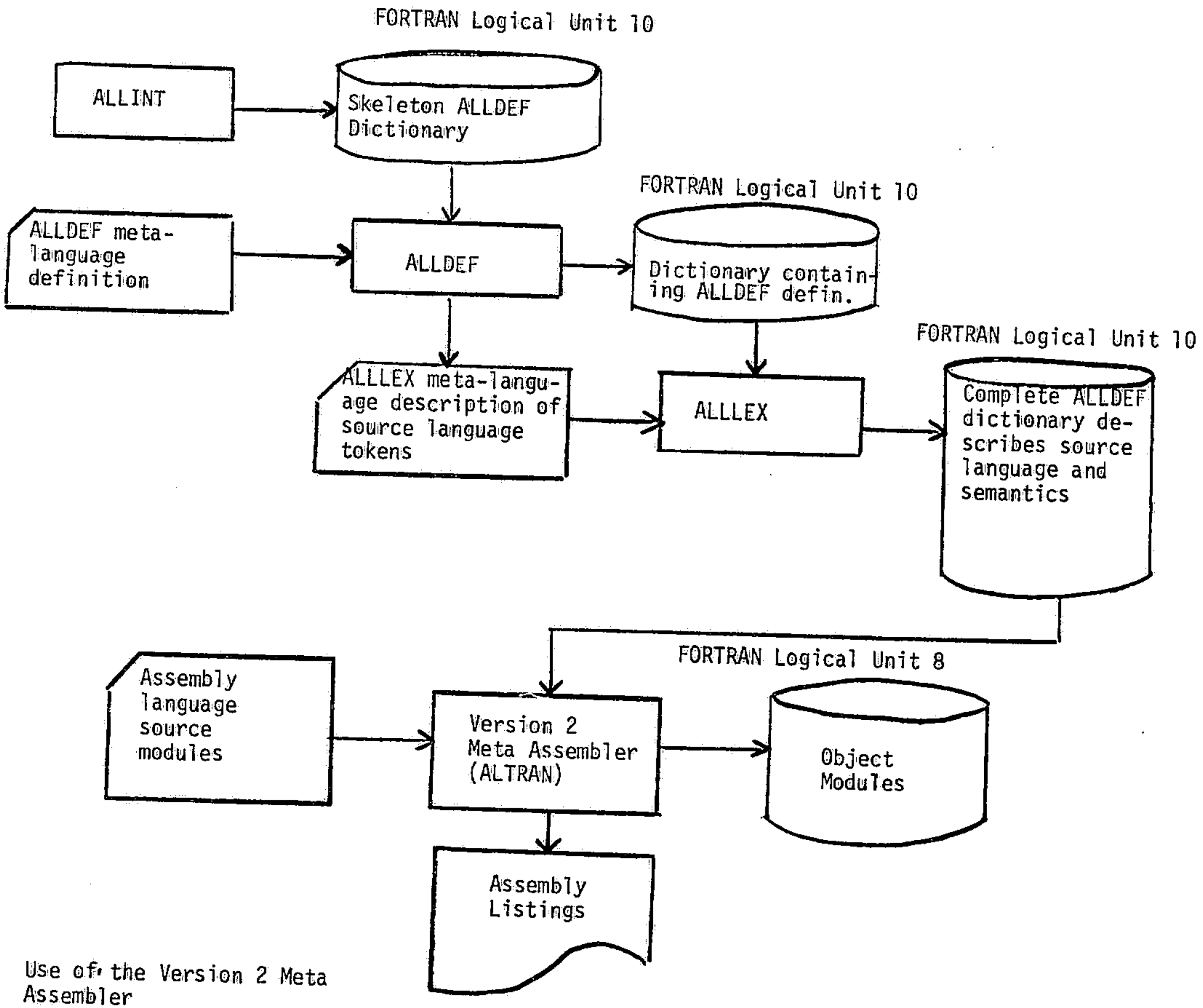

FORTRAN Logical Unit 10
ALLINT
Skeleton ALLDEF Dictionary
ALLDEF meta-language definition
ALLDEF
FORTRAN Logical Unit 10
Dictionary containing ALLDEF defin.
ALLLEX meta-language description of source language tokens
ALLLEX
FORTRAN Logical Unit 10
Complete ALLDEF dictionary describes source language and semantics
FORTRAN Logical Unit 8
Assembly language source modules
Version 2 Meta Assembler (ALTRAN)
Object Modules
Assembly Listings

Figure 6. Use of the Version 2 Meta Assembler

3.1.1.1 Assembler Level Language Definition Meta Language (ALLDEF)

The purpose of the meta language, ALLDEF, is to provide an easy to use environment in which to describe an assembler language syntax and correlating semantic process. The design of ALLDEF is based on the OPALDEF meta language developed by MDAC for the U.S. Army Armament Command, Frankford Arsenal. Key to the concept of ALLDEF is its correlation to a bottom-up operator precedence parsing function. This permits a simplistic meta language notation and results in efficient parsing. Basically, ALLDEF represents a "dictionary" definition concept where the symbols of the target assembler language are defined in terms of their spelling (lexically) and their meaning (semantics). The meanings are defined contextually, i.e., where the symbol may appear and translationally, i.e., what Meta Assembler built-in semantic function is to be performed.

A statement in ALLDEF may take forms to define user types, parameter table entries, target machine characteristics, assembler language symbols, semantic functions and comments. The ALLDEF definitions are specified in a free-form structure with the constraint that user type, parameter table and target characteristic definitions must precede their references.

User Type Definition

A type is an attribute associated with a symbol which categorizes that symbol uniquely. Thus, a symbol may be bound unambiguously to an operator based on its type. A set of built-in types will be provided to the ALLDEF language including:

NUMBER	-	a digit string
VALUE	-	a NUMBER symbol which has been converted to its binary representation.
NAME	-	a character string which satisfies a definition of an assembler level mnemonic or symbol notation.
LABEL	-	a NAME symbol which is identified in the label field of a statement.
ADDRESS	-	a NAME symbol defined in the assembler symbol table as an address value.

CHAR_STRING	-	a character string normally delimited and typed for text processing.
SPECIAL	-	a character string composed of special characters.
SYMBOL	-	a character string which cannot otherwise be typed as NUMBER, NAME, CHAR_STRING, or SPECIAL.

The available built-in types are used to provide initial token classification and the set may be extended further via the TYPE statement in ALLDEF. This provides unique binding attributes for tokens defined in ALLDEF.

Example:

```
    TYPE    'REGISTER,' 'MEMORY,'....$
```

<u>Parameter Table Entry Definition</u>

A parameter table is available for utilization. Essentially, the entries in the parameter table are the translation time variables defined, optionally initialized, and used as desired. The parameter table is divided into two sections, a global and a local section. The global section contains the variable entries that are initialized only at the start of the assembly. The local section contains the variable entries that are initialized at the start of each statement assembly. Additionally, all of the built-in translation parameters are available in fixed entries in the parameter table including:

CURSOR	-	current input statement cursor position in the local section and initialized to 1.
CURSOR_CHAR	-	character under the CURSOR position, in the global section.
OPCODE	-	operation code value for object generation, in the global section.
BIT_LENGTH	-	bit string length for object generation, in the global section.

FIELDS	-	the number of fields to parse for a statement, in the local section and initialized to 3.
LOCATION	-	assembly location counter, in the global section initialized to zero.
CTL-SECTION	-	current control section for LOCATION, in the global section initialized to 1.
MEM-SIZE	-	global section parameters correlating to the Meta Assembler SIZE directive
ADDRESS_UNIT	-	
ACCESS_UNIT	-	
ERROR_SIZE	-	
VALUE_SIZE	-	
OBJECT_SIZE	-	

The user may extend the parameter table via the GLOBAL and LOCAL statements in ALLDEF.

Example:

```
     GLOBAL    'LEVEL'=1, 'NEST'....$
                                  Global section definitions LEVEL is initialized
                                  to 1 and NEST is initialized to zero by default.
     LOCAL     'SOURCE', 'DEST', 'STYPE' = DOUBLE...$
                                  Local section definitions SOURCE and DEST
                                  are initialized to zero by default and STYPE
                                  is initialized to DOUBLE (previously defined
                                  on a TYPE statement).
```

Target Machine Characteristics

The target machine characteristics are the parameters needed to perform the cross assembly function. Some of the characteristic parameters are maintained as fixed built-in entries in the parameter table (see paragraph 2.1.1.2).

Assembler Language Symbols

The process of building an assembler language "dictionary" consists of defining the assembler language symbols, or tokens, and the correlating semantic functions, i.e., object generation and assembler directive processing. ALLDEF statements are needed to define the assembler level tokens in terms of operator precedence rules for the syntactic processing, and the semantic functions to be performed. It is at this point that the essence of unique assembler language translation into Meta Assembler intermediate language occurs.

ALLDEF Statement for Assembler Language Operator Definition - ALLDEF statements are used to define the assembler language symbols, i.e., instruction mnemonics, directive mnemonics, and the special operators of the assembler language statements, creating the environment for an operator precedence syntax processing. The remaining task is to define the syntactic meaning of the operator definitions. The syntactic meaning of an assembler level token defined in ALLDEF takes the form of:

- definition of the results
- definition of the operands allowed
- definition of the operator precedence
- parameter table action
- semantic action

The collective ALLDEF terms to define the assembler level symbols and their meaning comprise the ALLDEF statement.

Assembler Level Operator Definition - The assembler level operator definitions describe the verbs and special operators of the assembler language and provide the mechanism to perform a statement parse. The operator definition term occurs first in an ALLDEF statement. Machine instructions and directives are the action verbs of the assembler statements which result in a statement level semantic, i.e., object generation and directive function. Special operators are the sub-statement identifiers that perform on the action verb operands. Their associated semantics build toward full statement recognition at assembly time.

Examples:

```
     INSTRUCTION 'MOV':  }
                         }      action verbs
     DIRECTIVE 'EQU':    }
     PREFIX OPERATOR'#':
     POSTFIX OPERATOR '@':}     special operators
     INFIX OPERATOR',':   }
```

Definition of Results - A result is the mandatory type of information to be returned to the parsing process upon complete recognition of an operator (other than the action verbs INSTRUCTION and DIRECTIVE). A result is expressed in terms of ALLDEF types.

Example:

```
     RESULT=REGISTER
```

Definition of the Operands Allowed - Operands are defined in terms of their order, optionality, type, kind, and term or sublist structure. The order position of the operand is correlated to a left-to-right scan of the operands. The type must be an ALLDEF type. The kind refers to the built-in generic type used to further bind operands and operators, i.e., EXPRESSION. The sublist structure, SUBLIST, indicates a delimited term, i.e., a parenthesized notation. The keyword OPTIONAL defines the presence of an operand is allowed but not required.

Examples:

```
     OPERAND(1) = REGISTER SUBLIST
     OPERAND(2) = OPTIONAL ADDRESS EXPRESSION
```

Definition of the Operator Precedence - The precedence specified in a definition provides the priority for reducing an operator to its result. Default precedence is assigned to the various operators, however, the precedence may be explicitly specified.

Example:

```
     PRECEDENCE=50
```

Semantic Action - The semantics of an operator definition are described in a semantic clause which explicitly specifies semantic functions or refers to a separate semantic definition statement.

Semantic actions occur at two different levels of processing. First, there are the assembler function semantics which perform statement level semantics, i.e., symbol table definition and object code generation. Second, there are syntactic processing semantics which manipulate parameter table variables and operands, i.e., building operand lists, in order to effect precise assembler language statement recognition. Additionally, decision making phrases and arithmetic operations are available to the semantic clause providing flexibility over the semantic definition. This consists of an IF-THEN-ELSE-END type of phrase structure and arithmetic function keywords.

Action may be taken upon parameter table variables in the form of assignment statements. This is an immediate translation semantic available for use at the language definer's discretion.

Example:

```
    NLEVEL=NLEVEL+1
```

The assembler function semantics are represented by directive processing, i.e., symbol table definitions, macro processing, literal pool processing and object generation.

Examples:

```
    CREATE_SYMBOL(OPERAND(2))         |
    CREATE_DATA('DATA',OPERAND(1))    |           symbol table processing
    LITERAL(OPERAND(3))                   |       literal pool processing
    SECTION('DATA')                   |
    SECTION(OPERAND(1))               |           control section processing
    CREATE_MNEMONIC(OPERAND(1))           |       mnemonic definition,ie.,macro
    OBJECT(ADDRESS_TYPE(LOCATION_COUNTER),FIELD(0-3)=  |
        OPCODE,FIELD(4-7)=OPERAND(1,1) FIELD(8-15)=    } object code generation
        OPERAND(1,2))
```

The syntactic processing semantics perform actions upon the operands during the assembler level statement recognition process.

Example:

`LISTF(OPERAND(1),OPERAND*2),O'65')`	build an operand list composed of 3 elements

The decision making phrase provides the capability to have alternate paths as well as establish the truth condition for the operator definition recognition. Available to the IF phrase is the ability to test:

- operator kind, spelling or precedence
- operand value
- operand presence (optional testing)
- parameter table value
- value of expressions

Example:

```
     IF(PRESENT(OPERAND(1)))
       IF(SYMBOL_TYPE(OPERAND(1)).EQ.REGISTER),
           CHK-REG1,
       ELSE,
           CHK-REG2,
       END,
           LISTF(OPERAND(1),OPERAND(2)),
       END $
```

<u>ALLDEF Operator Semantic Definition Example</u>

```
    INSTRUCTION 'MOV':  OPERAND(1)=REG_REG,RESULT=DOUBLE,
                        SEMANTIC=OPCODE=O'01',BDL16$
    INFIX OPERATOR ',':  RESULT=REG_REG
              OPERAND(1) = REGISTER,
              OPERAND(2) = REGISTER,
              SEMANTIC=    CHK-REGS,LISTF,
                           END $
    SEMANTIC 'DBL16':      BIT_LENGTH=16
```

```
OBJECT(ADDRESS_TYPE(LOCATION_COUNTER),FIELD(0-3)=OPCODE,
       FIELD(4-9)=OPERAND(1,1),
       FIELD(10-15)=OPERAND(1,2))$
```

An example of NSSC-1 assembly language is contained in Appendix B.

3.1.1.2 Assembler Level Language Lexical Meta Language (ALLLEX)

Lexical Analysis

The lexical processing is performed by interpeting a meta definition of the lexicon to perform token identification in a top-down fashion. The meta language for defining the lexical processing is very similar to the meta language of the MDAC Meta Translator and is processed by a preprocessor step subsequent to the ALLDEF processing of the syntax meta definition.

The primary purpose of the lexical meta definition is to define the assembly time token fetch and identification process.

It became clear that a parameterized standard lexical function is prohibitive due to the context sensitive uniqueness found in assembler languages. This has led to the necessity of providing a specialized meta language to adequately address the token fetch and identification process.

It is the responsibility of the lexical process to fetch a token and identify it as one of the basic types:

NUMBER	-	a digit string token which can be converted to a binary value.
VALUE	-	a NUMBER token which has already been converted to its binary representation.
NAME	-	a character string token which has the properties of an assembler level mnemonic or symbolic notation.
LABEL	-	a NAME token which has been identified in the label field.
CHAR_STRING	-	delimited character string token

SPECIAL	-	a token composed of special characters only as defined by meta language.
SYMBOL	-	a token which cannot be otherwize identified as a NUMBER,NAME,CHAR_STRING or SPECIAL.

The lexical meta definition provides a top-down, recursive descent, goal oriented technique for token fetch and identification.

The meta definition consists of productions developed to guide the lexical process by defining the following lexical situations:

- what is a NUMBER token
- what is a NAME token
- what is a LABEL token
- what is a CHAR_STRING token
- what is a special character
- what is a subfield separator
- what is parse order for token identification
- what is the end of a statement field condition
- what is the end of statement condition

The lexical meta language is a modified Backus Naur Format (BNF) notation which will provide the basic parse functions:

- exclusive cursor control
- truth/false path prediction
- reoccurance processing
- recursive processing
- lit'ral string prediction

The extended lexical parse functions include:

- built-in primitive definitions

  e.g. LETTER,DIGIT,CHARACTER,etc.
- parse state conditional testing

  The ALLDEF PARAMETER table and built-in global variables will be available for assignment and conditional testing (e.g., CURSOR, CURSOR_CHAR,FIELDS,etc.)

- token construction and assigning the initial identity (e.g., NUMBER, NAME, etc.)

While there is a distinct separation of the lexical and syntactic parse functions, there is a common source of the overall statement recognition state. Through the ALLDEF parameter table and other built-in global variables, specialized parse functions can be controlled, i.e., label field identification, end of statement detection, assembler processing modes for special lexical and syntactic definitions (macro and text functions).

Example:

```
     <TOKEN>:= <NUMBER>//<NAME>//<SPECIAL>//<SYMBOL>$
     <NUMBER>:= (IF'0',BASE=8//BASE=10),
                <DIGITSTRING>,
                (IF BASE EQ 8, TOKEN_VALUE=VALUE OF
                 OCTAL<DIGITSTRING>//TOKEN_VALUE=
                 VALUE OF <DIGITSTRING>),TOKEN_TYPE=VALUE,
                 IF NOT LETTER $
    <DIGITSTRING>:= 1 to MANY DIGITS $
    <NAME>:= LETTER, 0 to MANY (LETTER//DIGIT),
             TOKEN_TYPE=NAME$
    <SPECIAL>:=(','//'.'//'+'//'-'//'*'//'/'),TOKEN_TYPE=SPECIAL$
    <SYMBOL>:= 1 TO MANY (IF NOT SPACE,NOT <SPECIAL>, CHARACTER),TOKEN_TYPE
               =SYMBOL$
```

3.1.1.3 ALLDEF Meta Language Processor

A meta language processor was developed to process syntactic meta definitions into the ALLDEF dictionary composed of the syntax and semantic tables. The ALLDEF processor functions as a stand-alone preprocessor to the Meta Assembler. The ALLDEF dictionary file is preserved as an input file to the generalized parser function of the Meta Assembler which eliminates the need to execute the ALLDEF processor for each cross assembly.

The design of the ALLDEF processor is based on the OPALDEF processor developed by MDAC for the U.S. Army, as is the ALLDEF meta language design based upon the OPALDEF meta language (see Figure 7).

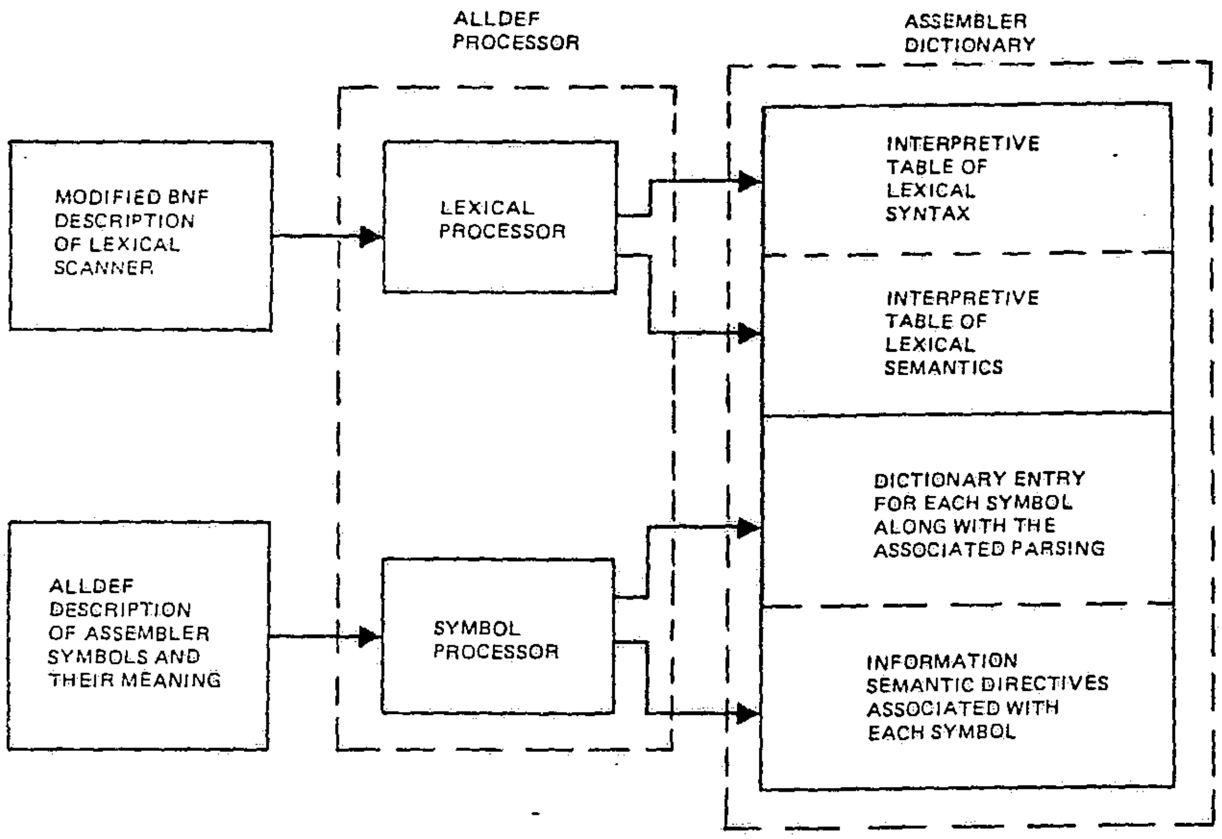

ALLDEF PROCESSOR
ASSEMBLER DICTIONARY
MODIFIED BNF DESCRIPTION OF LEXICAL SCANNER
LEXICAL PROCESSOR
INTERPRETIVE TABLE OF LEXICAL SYNTAX
INTERPRETIVE TABLE OF LEXICAL SEMANTICS
DICTIONARY ENTRY FOR EACH SYMBOL ALONG WITH THE ASSOCIATED PARSING
ALLDEF DESCRIPTION OF ASSEMBLER SYMBOLS AND THEIR MEANING
SYMBOL PROCESSOR
INFORMATION SEMANTIC DIRECTIVES ASSOCIATED WITH EACH SYMBOL

Figure 7. ALLDEF Processor

3.1.1.4 ALLLEX Meta Language Processor

A meta language processor was also developed to process lexical meta definitions into an existing ALLDEF dictionary. The ALLLEX processor functions as a post processor to the ALLDEF processor and a preprocessor to the Meta Assembler.

The design of the ALLLEX processor is based on the MDAC Meta Translator.

3.1.1.5 Generalized Parser (ALTRAN)

The ALTRAN processor will be developed as an integral module of the Meta Assembler. It provides the alternative first pass processing of the Meta Assembler by translating assembler language source statements into the Meta Assembler intermediate language structures and performing assembler directive semantics via the ALLDEF dictionary (see Figure 8).

ALLTRAN Parsing

The parsing technique employed in ALTRAN is a precedence analysis scheme utilizing a left-to-right scan. A reduction of an operator and its operands to the defined result is made when another operator is recognized of a lower or equivalent precedence value. Any semantic associated with the reduced operator is also effected at that time. The assembler directive semantics, i.e., symbol table manipulation and control section activation, are performed immediately by built-in support routines. The object generation semantics build a list of intermediate language elements on the intermediate language file. During the parsing process of ALTRAN, the operators and operands are placed on stacks for evaluation. The binding of operands to operators is performed on the basis of the ALLDEF operator definitions. The proper operator definition is detected by matching the available operands with the ALLDEF operator definition which permits operator reduction to occur. The implication is that multiple definitions of the same operator are permitted.

3.2 META TRANSLATOR IMPLEMENTATION

3.2.1 Meta Translator Description

The Meta Translator is a propreitary translator writing system (TWS) developed at MDAC-W that is a very effective tool for the generation of language translators (see Figure 9). It is machine independent in the class of medium and large scale computers that have an ASA FORTRAN IV compiler.

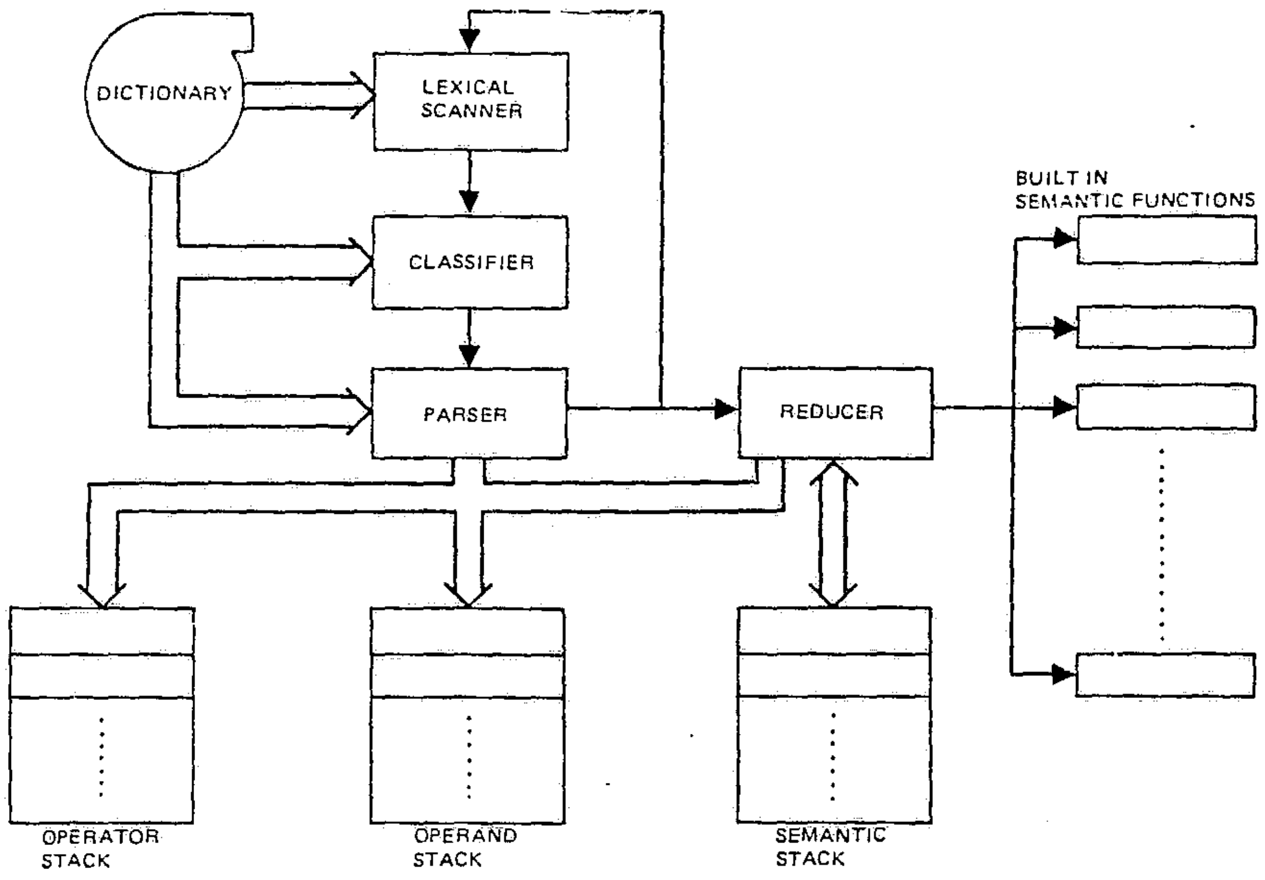


Figure 8. ALLTRAN Processor

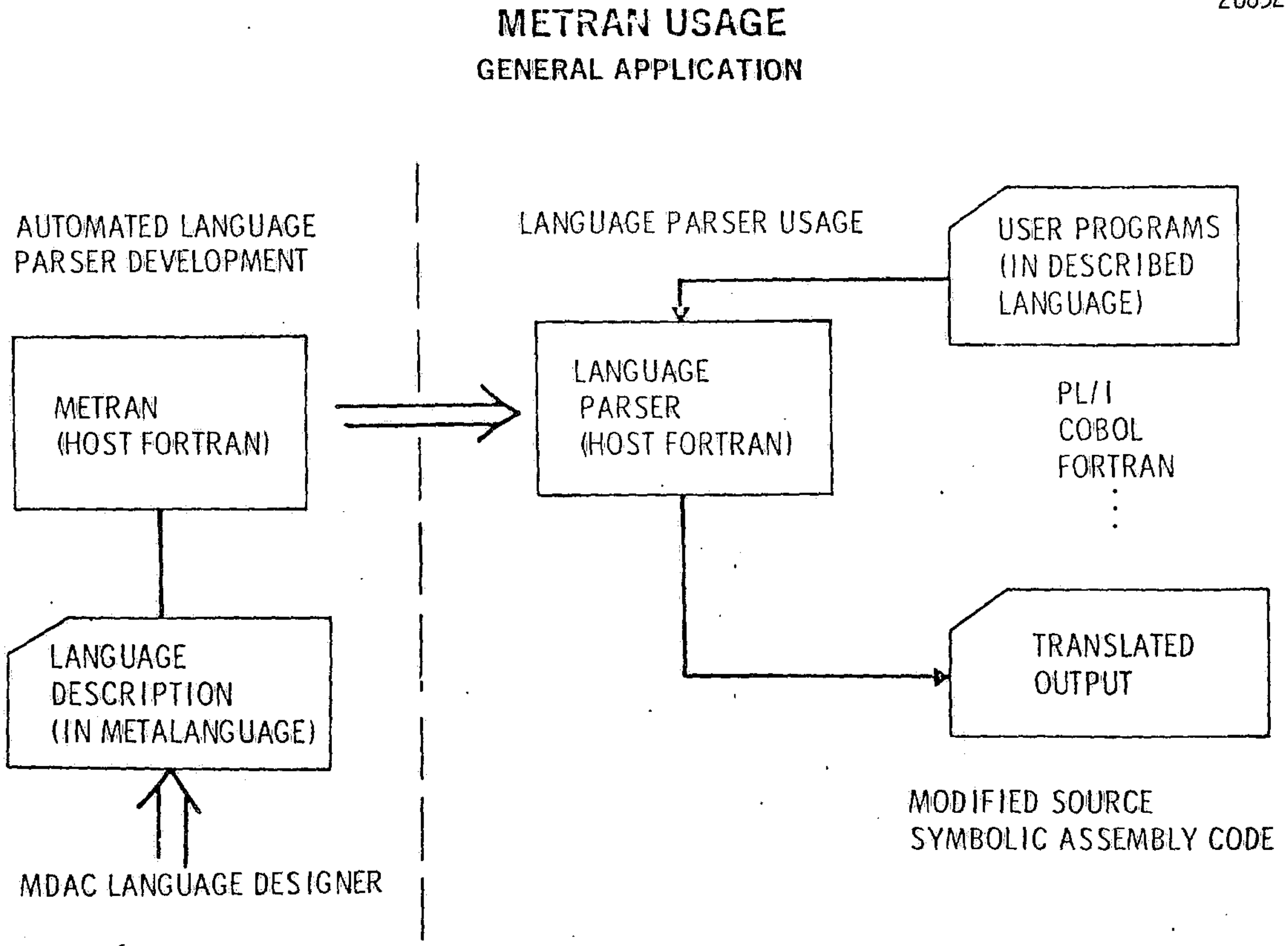


Figure 9. Meta Translator General Application

Every translator consists of a parser to recognize syntax, a procedure executor and a set of subroutines to perform semantic functions, a number of support routines that perform common functions, and a control driver to act as an executive, controlling the flow of operations.

The parser and procedure executor are generated by the Meta Translator since they are language-dependent. The semantic procedures may invoke built-in or user supplied subroutines. The support routines are not generated but are provided as an adjunct to the generated code. The control driver is a short main program which initiates translation, and is written by the language definer in FORTRAN IV.

The language definition is written in the meta language by the language definer (see Figure 10). It is this definition that is translated into the parser and procedure executor by the Meta Translator. A supporting BLOCK DATA subroutine is also generated for initialization of syntax and semantic parameters.

3.2.2 Meta Translator Application

The Meta Translator was used to originally produce the Meta Assembler syntax processing subroutines and is integral to the implementation of the ALLDEF and ALLTRAN processors. This technique utilizes a meta language for defining the syntax processing algorithms and greatly eases implementation and maintenance functions.

To provide maintenance capability to NASA, the Meta Translator was installed at MSFC and the Meta Assembler meta language source was a deliverable item.

3.3 GENERALIZED LINKAGE EDITOR

3.3.1 General Overview

The Generalized Linkage Editor (GLE) is a multi-functioned utility designed to aid the Meta Assembler user in the creation and maintenance of software systems built from Meta Assembler formatted object modules.

FORTRAN TRANSLATION EXAMPLE

```
$DECLARATIVENAME  .=SET CASE I =                         /*DETERMINE DECLARATIVE TYPE          */
                    ('DIMENSION', 'INTEGER',
                     'REAL', 'COMMON', 'DATA',
                     'EQUIVALENCE', 'DOUBLE',
                     'LOGICAL', 'COMPLEX',
                     'IMPLICIT').

$DODECLARATIVE    .=CASE I OF                            /* PARSE THE DECLARATIVE              */
                    ($DIMENSION,$INTEGER,
                     $REAL,$COMMON,$DATA,
                     $EQUIVALENCE,$DOUBLE,
                     $LOGICAL,$COMPLEX,
                     $IMPLICIT).

$DIMENSION        .=TEXT($DECLARATIVENAME,'         ), /* OUTPUT 'DIMENSION'            */
                    LIST OF 1000                         /* PARSE THE ARRAY LIST               */
                       BEGIN
                           $NAME,                        /* PARSE AND OUTPUT ARRAY NAME        */
                           TEXT($NAME),
                           $FINDDIMENSION,               /* PARSE AND OUTPUT PARENS AND        */
                           $COMMA                        /* SUBSCRIPTS. OUTPUT A COMMA.        */
                       END.
                    TEXT(EJECT).                         /* PRINT AND PUNCH OUTPUT IMAGE       */
```

Figure 10. Meta Translator Example

Functionally, the GLE provides three basic services: creation and maintenance of libraries of object modules, binding of separately assembled modules to form a generalized load module, and cataloging of object modules, libraries and load modules to gather descriptive information.

3.3.1.1 Library Creation and Maintenance

This service provided by the GLE gives the Meta Assembler user the capability to create a new user/system library directly from the output of the Meta Assembler. Once a library has been created, it may then be updated using Meta Assembler output and the old library to create a new library.

3.3.1.2 Binding of Modules

This is the primary service of the GLE. Its function is to bind separately assembled modules, developed for a common target machine and residing on user and/or system libraries, into a generalized load module. The generalized load module is then available for transformation into the structure required by the specific target computer loader. A wide range of control is given to the user, through the use of directives, for determining which modules and in what order will appear in the resultant load module.

3.3.1.3 Cataloging of Standard Meta Assembler System Outputs

This capability gives the user a tool to display descriptive information about each of the three Meta Assembler system outputs: object modules, library of object modules, and load modules.

The available information includes: type of output, module name, module creation date and time, module version, target computer.

3.3.2 <u>Flow Through The Generalized Linkage Editor</u>

The flow of data through the GLE is controlled entirely by user supplied directives which represent: invocation of a basic service, tasks for a basic service to perform, and termination of a basic service.

It is expected that a couple of basic flow paths will be performed again and again. With this in mind, the following descriptions will outline these two basic flows (a macro flowchart appears in Figure 11).

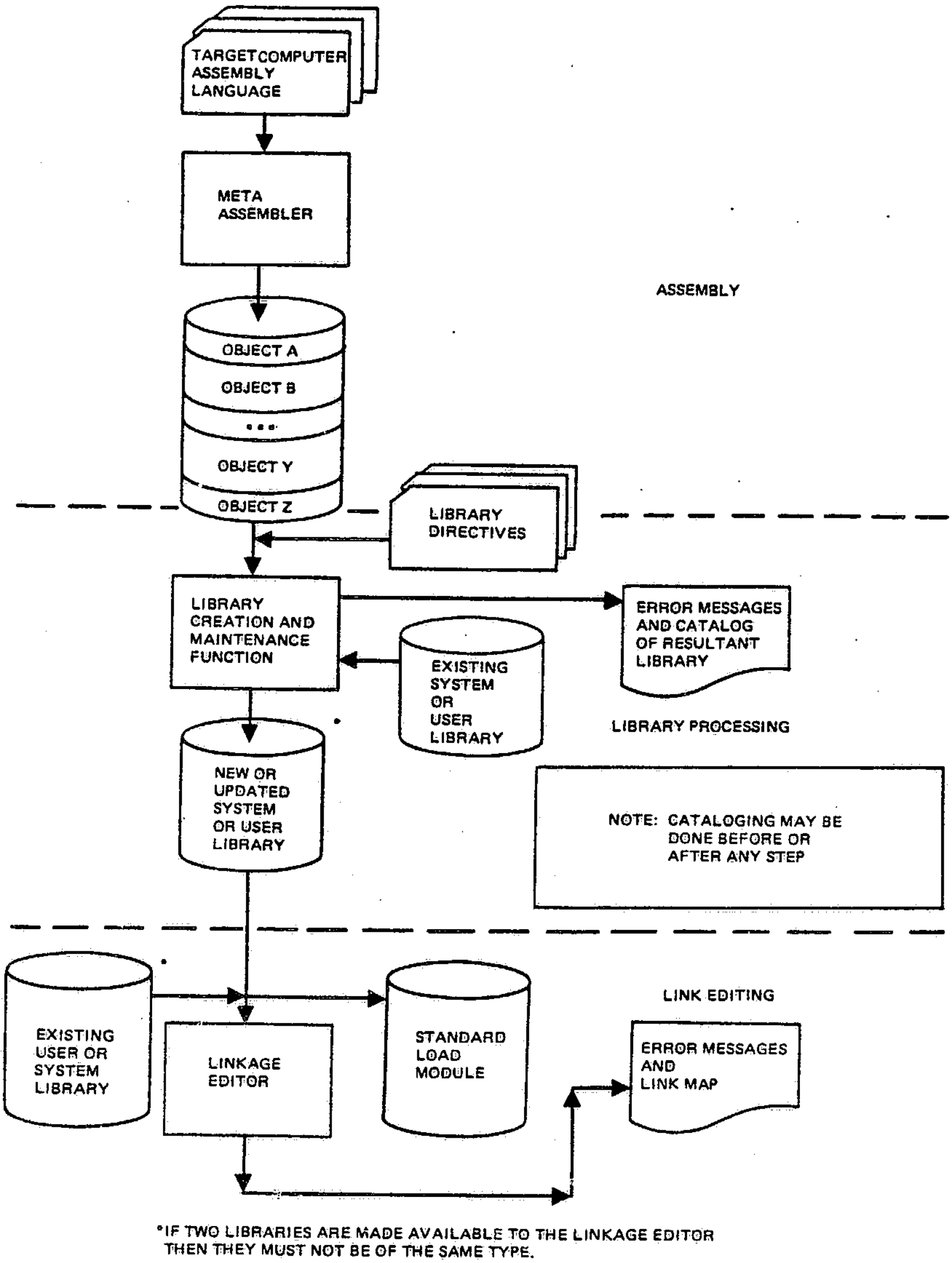

TARGET COMPUTER ASSEMBLY LANGUAGE
META ASSEMBLER
ASSEMBLY
OBJECT A
OBJECT B
• • •
OBJECT Y
OBJECT Z
LIBRARY DIRECTIVES
LIBRARY CREATION AND MAINTENANCE FUNCTION
ERROR MESSAGES AND CATALOG OF RESULTANT LIBRARY
EXISTING SYSTEM OR USER LIBRARY
LIBRARY PROCESSING
NEW OR UPDATED SYSTEM OR USER LIBRARY
NOTE: CATALOGING MAY BE DONE BEFORE OR AFTER ANY STEP
EXISTING USER OR SYSTEM LIBRARY
LINKAGE EDITOR
STANDARD LOAD MODULE
LINK EDITING
ERROR MESSAGES AND LINK MAP
*IF TWO LIBRARIES ARE MADE AVAILABLE TO THE LINKAGE EDITOR THEN THEY MUST NOT BE OF THE SAME TYPE.

Figure 11. Flow Through the Generalized Linkage Editor

3.3.2.1 Creation of a System Library for General Use

In a production atmosphere, there usually exists a set of object modules that perform widely needed utility functions: input/output, mathematical functions, date and time, etc. Once these functions are coded and tested they should be put into a library that is available to all users. The GLE's library creation and maintenance function will create a system library of object modules using the assembled utility functions as input. This library of utility functions is now in a form that may be used by the linkage editor service to satisfy references to them.

When changes to the system library are necessary, the library service has capabilities to update the old system library against newly assembled modules, using directives, to create a new system library.

3.3.2.2 Creation of a User Library and Load Module Generation

The Meta Assembler user will create a set of object modules to perform a particular task. As new tasks are required or old tasks become unnecessary, the set of object modules will change to reflect the current requirements. The GLE's library creation and maintenance function (LCMF) can model such a sequence. Given an initial set of object modules, the LCMF can create a user library of object modules. As changes are made, the LCMF can make the required changes to the user library.

Once a library is built, the linkage editor service may then be invoked. The linkage editor service, using a user library and/or system library, will create a load module.

3.3.3 Use of the Generalized Linkage Editor

Each service of the GLE is accessed by user directives. These directives control service invocation, service termination and service tasks to be performed.

All directives are of the general format described below.

3.3.3.1 Directive Coding Conventions

Notation Used to Describe Directives

The descriptive notation used to define the syntax of the input directives makes use of upper and lower case letters and the characters left bracket ([) right bracket (]), periods(...), and vertical bar (|).

All keywords and other explicitly required symbols appear as upper-case or special characters. An implicit operand appears as a lower-case name which is described in a narrative subsequent to its usage.

An optional operand is shown enclosed within brackets([]). Occasionally, more than one level of optionality is required and is described in terms of brackets within brackets:

```
MAP     [ON ]     ; describes     MAP; or
        [OFF]                     MAP ON; or
                                  MAP OFF:
```

Choosing one of a list of operands is denoted by listing the operands vertically and enclosing them with vertical bars (||):

```
ENTRY  |    module   |  ;   describes ENTRY module; or
       |  (|symbol|) |                ENTRY (symbol); or
       |   |addr  |  |                ENTRY (addr);
```

Specifying a repetitive collection of identical operands is described by following the operand with a triple dot (...):

```
name [,name...] describes name or
                          name, name or
                          name,...,name
```

Format of Directives

All GLE directives are formed according to the following rules and restrictions:

- All directives are free-form using columns 1-72
- Blanks are ignored and are used for readability only
- Each directive is terminated by a semicolon
- All text between the strings /* and */ is ignored, this string may not contain intervening blanks.
- More than one directive may appear on a card
- Directives may be contained on more than one card

The following example illustrates the preceding points:

```
 1                                             72 73      80
---------------------------------------------------------------
  LIBRARY                                          DIR     001
  LIBRARY SERVICE FUNCTION */;                     DIR     002
  BEFORE A,B; /*PUT B BEFORE A, AND */A            DIR     003
  FTER C/*PUT/,D; END;                             DIR     004
  LINKEDIT; INCLUDE A:SLIB(4000)                   DIR     005
,S1(1),S2(2);MAPON;END;CATALOG                     ;IR     006
;FILES=8,ULIB /*,*/,SLIB;                         /DIR     007
*CATALOG END*/END;                                 DIR     008
---------------------------------------------------------------
```

3.3.3.2 List of Generalized Linkage Editor Directives

The directives listed below give a quick summary of capabil'ties for each basic service provided by the GLE.

Library Creation and Maintenance Function Directives

° LIBRARY	Invoke Library Function
° CREATE	Build New Library from Meta Assembler Output On
° NAME	Specify name of library
° KIND	Specify kind of library
° BEFORE	Position for a new module
° AFTER	Position for a new module
° DELETE	Delete modules from old library
° IGNORE	Ignore new module
° RENAME	Give module new name
° NO AUTOREP	Only processes new modules named on before, after and replace directives
° REPLACE	Allows select replacement of modules
° END	Terminate library function

Linkage Editor Directives

° LINKEDIT	Invoke LINKAGE EDITOR
° FILES	Indicates file to be linked
° RELOCATION	Specify address fields
° MODE	Force type of load module
° INCLUDE	Force inclusion of a module from a library
° EXCLUDE	Force exclusion of a module from load module
° NOULIB	Force exclusion of entire user library
° NOSLIB	Force exclusion of entire system library
° RENAME	Cause external reference name change
° ENTRY	Specify execution start address
° NAME	Name load module
° MAP	Turn link map listing on or off
° GSECT	Cause assembly time control sections to be loaded consecutively
° BOUND	Determine module bounding
° END	Terminate linkage editor function

Catalog Directives

° CATALOG	Invoke catalog service
° FILES	Specify which files are to be cataloged
° END	Terminate catalog function

3.3.3 3 Use of the Library Creation and Maintenance Function

An important function of the GLE is to be able to create and maintain two types of libraries; system and user. The purpose of a library in the GLE system is to provide the user with a utility with which to manipulate assembled object modules and to provide the linkage editor with a set of object modules from which external references may be satisfied.

Even though there are two distinct types of libraries, the only real difference between them is in the way they are used by the linkage editor. Structurally, a system and user library are equivalent.

Modes of Use

The library creation and maintenance function (LCMF) operates in two modes; creation and maintenance.

Creation Mode - The creation mode of the LCMF causes the object modules output from the meta assembler to be formatted into a standard library (see Figure 12). During library creation, the following restrictions must be kept in mind:

- A library may not contain modules with duplicate names
- The CREATE directive is mandatory and must be the second directive
- NAME,KIND and PASSWORD are the only other directives allowed
- The library will contain the modules in the order in which they are encountered.

Maintenance Mode - If the CREATE directive is not the second directive encountered then the mode is assumed to be the maintenance mode. Processing of new modules is handled by two basic procedures: implied automatic replacement, and directed replacement by use of directives.

If no processing directives are given, then the LCMF creates a new library by replacing the modules of the old library with modules that have the same

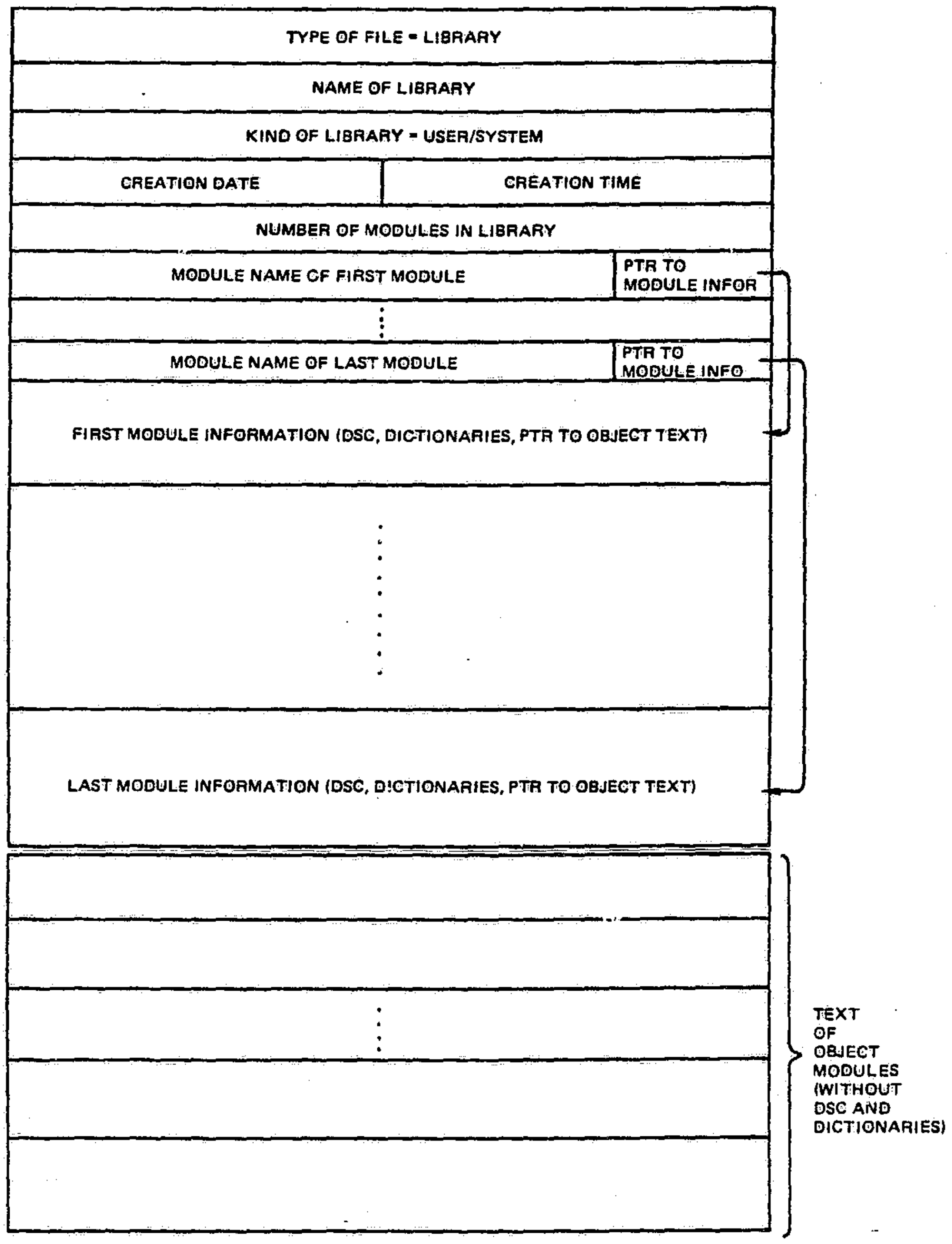

TYPE OF FILE = LIBRARY
NAME OF LIBRARY
KIND OF LIBRARY = USER/SYSTEM
CREATION DATE
CREATION TIME
NUMBER OF MODULES IN LIBRARY
MODULE NAME OF FIRST MODULE
PTR TO MODULE INFOR
MODULE NAME OF LAST MODULE
PTR TO MODULE INFO
FIRST MODULE INFORMATION (DSC, DICTIONARIES, PTR TO OBJECT TEXT)
LAST MODULE INFORMATION (DSC, DICTIONARIES, PTR TO OBJECT TEXT)
TEXT OF OBJECT MODULES (WITHOUT DSC AND DICTIONARIES)

Figure 12. Standard Library Format

name as output from the Meta Assembler. Any new modules will be written at the end of the new library.

If processing directives are given then transcription of modules to the new library will take place according to the directives.

A functional flowchart of the LCMF appears in Figure 13.

Detailed Description of LCMF Directives

FORMAT

LIBRARY;

DESCRIPTION

This directive must be present as the first directive to invoke the LCMF.

FORMAT

CREATE;

DESCRIPTION

This directive must be the second directive encountered in order to cause a new library to be created, using Meta Assembler output only. If CREATE is not the second directive encountered then it is assumed that an old user or system library is available to update against.

FORMAT

NAME=libname;

DESCRIPTION

This directive uses the symbol string "libname" to give the library a name. If this directive is absent for a creation mode then a default name of "LIBRARY1" is given to the library.
An updated library retains its original name unless changed by the NAME directive.

FORMAT

$$\text{KIND} = \left|\begin{array}{l}\text{USER}\\ \text{SYSTEM}\end{array}\right| \;;$$

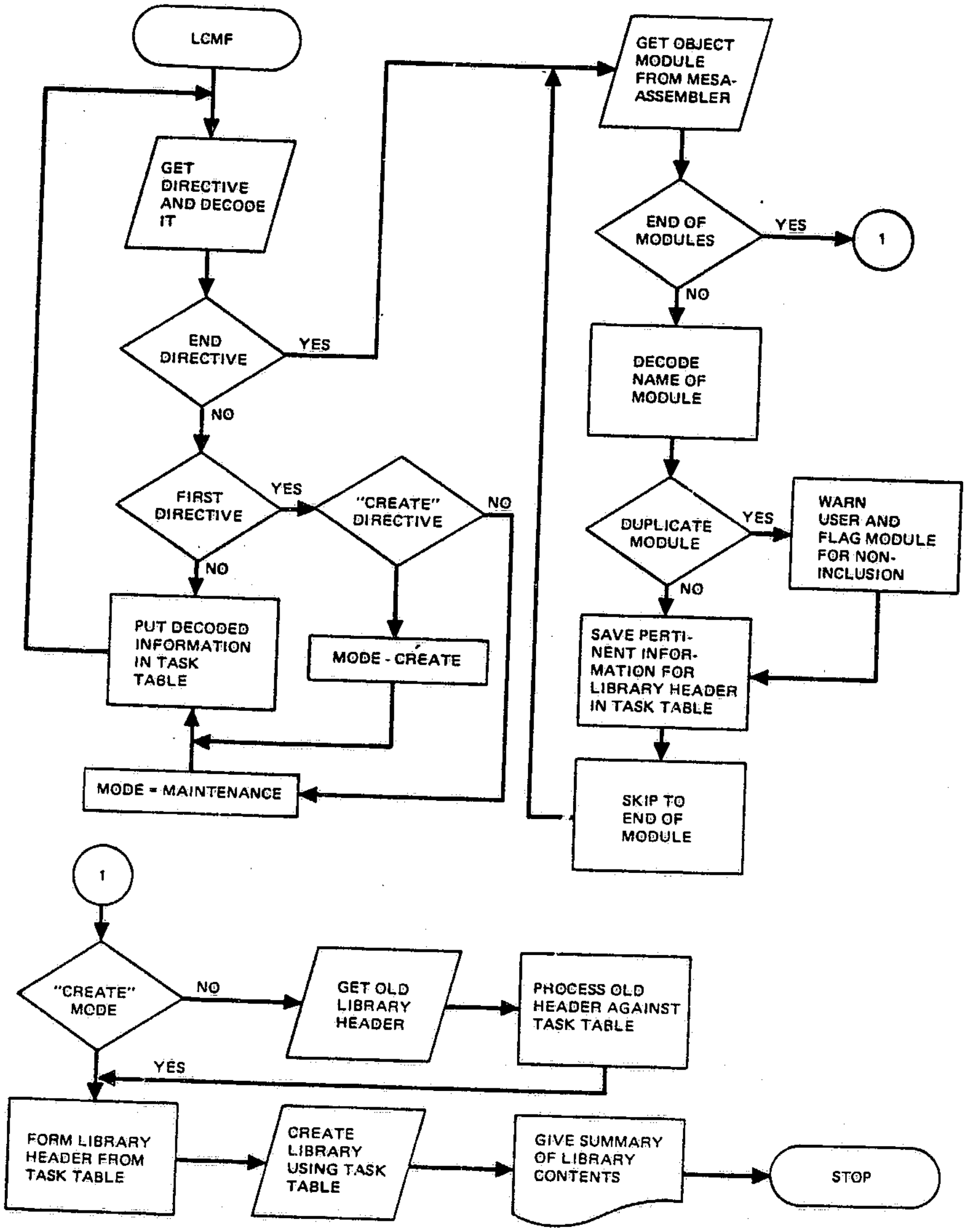

LCMF
GET DIRECTIVE AND DECODE IT
END DIRECTIVE
YES
NO
FIRST DIRECTIVE
YES
NO
"CREATE" DIRECTIVE
NO
PUT DECODED INFORMATION IN TASK TABLE
MODE - CREATE
MODE = MAINTENANCE
GET OBJECT MODULE FROM MESA-ASSEMBLER
END OF MODULES
YES
1
NO
DECODE NAME OF MODULE
DUPLICATE MODULE
YES
WARN USER AND FLAG MODULE FOR NON-INCLUSION
NO
SAVE PERTINENT INFORMATION FOR LIBRARY HEADER IN TASK TABLE
SKIP TO END OF MODULE
1
"CREATE" MODE
NO
GET OLD LIBRARY HEADER
PROCESS OLD HEADER AGAINST TASK TABLE
YES
FORM LIBRARY HEADER FROM TASK TABLE
CREATE LIBRARY USING TASK TABLE
GIVE SUMMARY OF LIBRARY CONTENTS
STOP

Figure 13. Functional Flowchart For the Library Creation and Maintenance Function

DESCRIPTION

This directive provides the library with a kind attribute. If this directive is absent for a creation mode then a default kind of "USER" is given to the library. An updated library will retain its original kind unless changed by the KIND directive.

FORMAT

NOAUTOREP;

DESCRIPTION

This directive declares that the LCMF function will not replace all modules from the old library with modules from the Meta Assembler having identical names, but selectively replaced modules according to REPLACE, BEFORE and AFTER directives.

FORMAT

BEFORE oldmod, new mod_1 [, new mod_i...];

DESCRIPTION

This directive causes the LCMF to insert the "newmod" modules from the Meta Assembler before the specified "old mod" for transcription to the new library. This causes automatic deletion of old modules having the same names from the old library.

FORMAT

AFTER oldmod, newmod_1 [,newmod_i...];

DESCRIPTION

This directive causes the LCMF to insert the "newmod" modules from the Meta Assembler after the specified "oldmod" on the old library for transcription to the new library. Insertion of this type causes automatic deletion of old modules having the same names from the old library.

FORMAT

PASSWORD=password;

DESCRIPTION

This directive specifies a password for the library. If this directive is absent then there is no default password given to the library. An updated library will retain its original password unless changed by the PASSWORD directive.

FORMAT

DELETE $oldmod_2$ [,$oldmod_i$...];

DESCRIPTION

This directive causes the LCMF to not copy the "oldmod" modules from the old library to the new library.

FORMAT

IGNORE new mod_1 [,$newmod_i$...];

DESCRIPTION

This directive causes the LCMF to ignore the "newmod" modules from the meta assembler during processing.

FORMAT

RENAME $oldname_1$ = $newname_1$ [,$oldname_i$= $newname_i$...];

DESCRIPTION

This directive assigns a new name to a module that will appear in the new library. If any other directives refer to this module, the old name should still be used.

FORMAT

REPLACE $newmod_1$[,$newmod_i$...];

DESCRIPTION

This directive is meaningful only during the effect of a NOAUTOREP directive. It causes the "newmod" modules to replace modules on the old library with the same names on the new library.

FORMAT

END;

DESCRIPTION

This directive causes termination of directive reading for the LCMF and initiates processing of the directives.

Examples of LCMF Use

For the following examples assume the existance of two Meta Assembler generated files, A and B, of object modules containing modules MA, MB, MC, MD and modules MD, MA, OX, OY, OZ respectively.

Example 1. Creation of a system library LIB1 from file of modules B.

```
Directives:  LIBRARY;
             CREATE;
             NAME=LIB1; KIND=SYSTEM;
             END;
```

System Library LIB1 contains MD, MA, OX, OY,OZ.

Example 2. Automatic update of LIB1 using file A to create user library LIB2.

```
Directives:  LIBRARY; KIND=USER, NAME=LIB2; END;
```

User Library LIB2 contains:

```
MD from A
MA from A
OX from LIB1
OY from LIB1
OZ from LIB1
MB from A
MC from A
```

Example 3. Restore MA from B on LIB2.

```
Directives:  LIBRARY;                        LIBRARY;
             NOAUTOREP;                      IGNORE MD;OX,OY,OZ;
             REPLACE MA;        or           END;
             END;
```

User Library LIB2 contains:

```
MD from LIB2
MA from B
OX from LIB2
OY from LIB2
OZ from LIB2
MB from LIB2
MC from LIB2
```

3.3.3.4 Use of LINKAGE EDITOR Function

The most important service provided by the GLE is the LINKAGE EDITOR (LE). The LE service provides the Meta Assembler user with the means to generate a standard format load module (see Figure 14) by binding separately assembled modules that reside in user and/or system libraries.

Since the LE must handle a variety of linkage editing requirements, a set of directives has been provided to give the user direct control over much of the load module generation process. The basic control features are:

- specification of execution start address
- order of module appearance in load module
- link map generation

Data Flow through the LINKAGE EDITOR

The LE expects as its primary inputs a user library of object modules from which to form a basis for a load module, and an optional system library from which to satisfy external references. The LE then reads and decodes the user directives, if any.

A "task" table is initialized with the decoded directives. Pertinent information includes: module order and start addresses supplied by "INCLUDE" directives, library to find module, and modules to exclude from the load module. If no service directives have been input then the "task" table is initialized by using the entire user library.

The "task" table is then processed to determine all the modules that will appear in the load module. This processing includes searching for definitions to any undefined references.

Once all the modules to be linked have been determined, addresses for all modules and control sections can be assigned. This completes filling in the "task" table. If a link map has been requested then the "task" table is used to create the map.

All that remains to be done is to generate the standard load module. First, the header block is written. The user and/or the system libraries are then

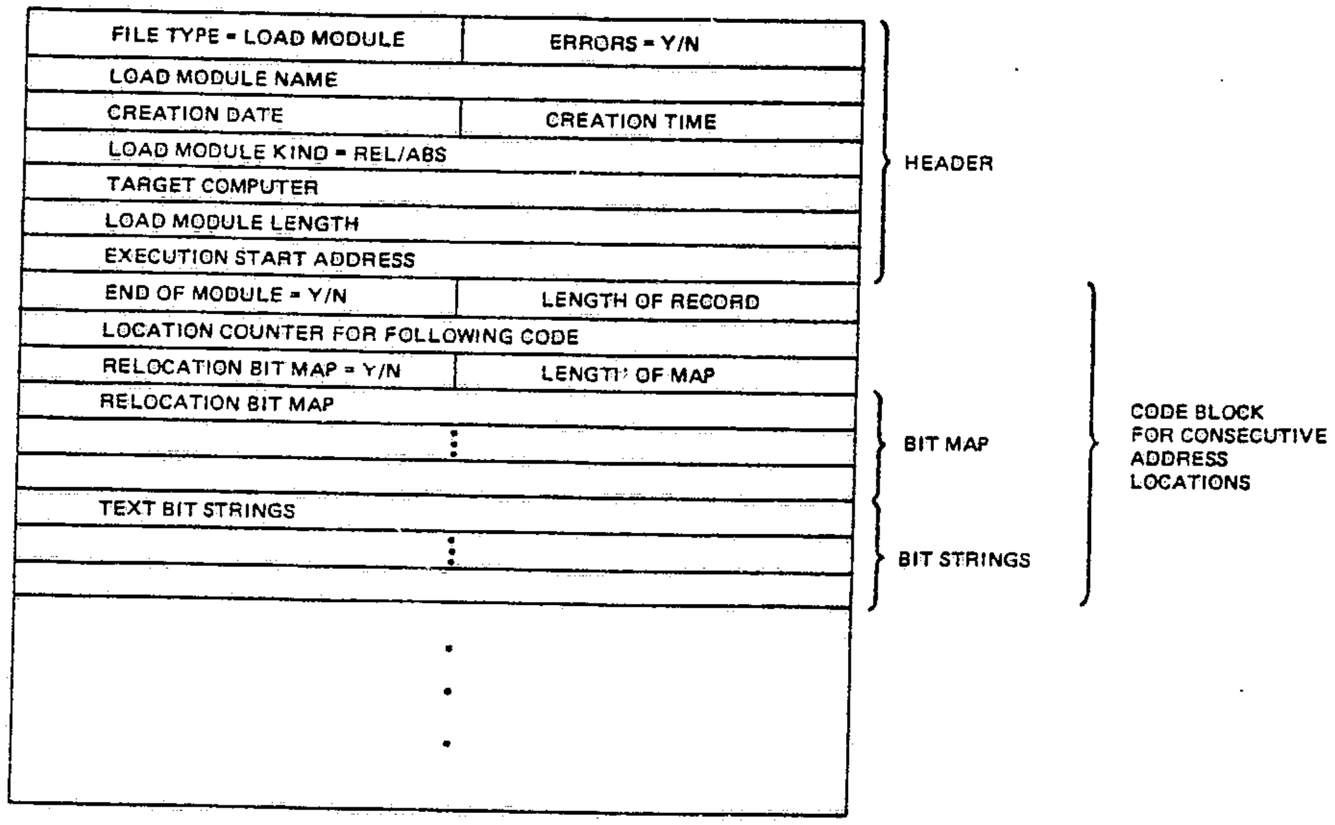


Figure 14. Standard Load Module Format

read sequentially. As a new module is read, it is either skipped or processed. All the information necessary to do address location is available from the "task" table. When the libraries have both been processed the linkage edition is complete. Figure 15 contains a functional flowchart of the LINKAGE EDITOR.

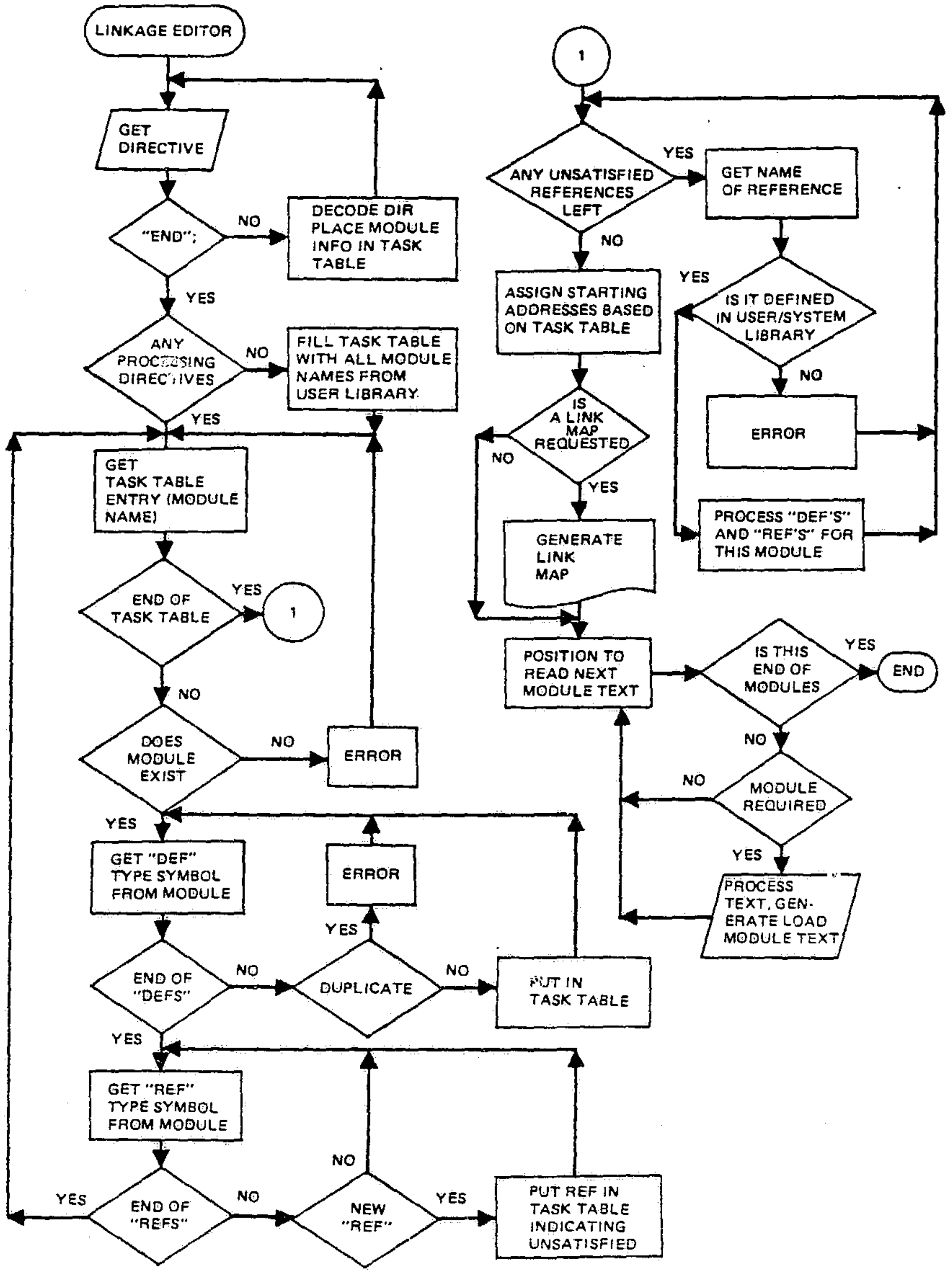

LINKAGE EDITOR
GET DIRECTIVE
"END";
NO
DECODE DIR PLACE MODULE INFO IN TASK TABLE
YES
ANY PROCESSING DIRECTIVES
NO
FILL TASK TABLE WITH ALL MODULE NAMES FROM USER LIBRARY
YES
GET TASK TABLE ENTRY (MODULE NAME)
END OF TASK TABLE
YES
1
NO
DOES MODULE EXIST
NO
ERROR
YES
GET "DEF" TYPE SYMBOL FROM MODULE
ERROR
YES
END OF "DEFS"
NO
DUPLICATE
NO
PUT IN TASK TABLE
YES
GET "REF" TYPE SYMBOL FROM MODULE
NO
YES
END OF "REFS"
NO
NEW "REF"
YES
PUT REF IN TASK TABLE INDICATING UNSATISFIED
1
ANY UNSATISFIED REFERENCES LEFT
YES
GET NAME OF REFERENCE
NO
ASSIGN STARTING ADDRESSES BASED ON TASK TABLE
YES
IS IT DEFINED IN USER/SYSTEM LIBRARY
NO
ERROR
IS A LINK MAP REQUESTED
NO
YES
GENERATE LINK MAP
PROCESS "DEF'S" AND "REF'S" FOR THIS MODULE
POSITION TO READ NEXT MODULE TEXT
IS THIS END OF MODULES
YES
END
NO
NO
MODULE REQUIRED
YES
PROCESS TEXT, GENERATE LOAD MODULE TEXT

Figure 15. Functional Flowchart for the LINKAGE EDITOR

Control of Load Module Generation

The GLE gives the user a wide range of control over the load module creation process. This control is divided into four main sections; load module type, execution start address, modules that will appear in load module, and generation of a link map.

General Directives

These directives control obvious features in the LINKAGE EDITOR.

- $\boxed{\text{LINKEDIT}}$

  Format

  LINKEDIT;

  Description

  This directive is required to invoke the LINKAGE EDITOR service.

- $\boxed{\text{NAME}}$

  Format

  NAME=lmod;

  Description

  The user may supply a name to be given to the generated load module. If the optional NAME directive is included, then the name of the load module will be 'lmod'. In the case where the directive is not included, then the default name of 'LOAD MODULE 1' will be supplied.

- $\boxed{\text{END}}$

  Format

  END;

  Description

  This directive terminates service directive reading and causes the LINKAGE EDITOR to perform the requested services.

Load Module Generation Node

The GLE will have the ability to produce load modules for a wide variety of target computers. The intent of the load module generation mode is to interface with various target machine loaders by producing absolute or relocatable load modules as required.

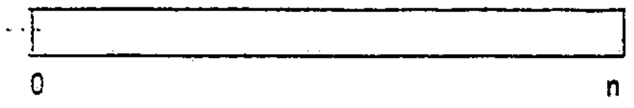
° RELOCATION

Format

```
  RELOCATION=(startbit:  endbit)[,(startbit:  endbit),...];
```

Description

This directive provides the GLE with a specification of all the fields that may contain addresses during an assembly. This allows the load module to create a relocation bit map, based on the specified fields, so that a relocating loader will know which addresses will need a load bias added. The 'startbit' indicates the starting bit position and the 'endbit' indicates the ending bit position for a field. All fields are described left to right with bit 0 (zero) assumed to be on the extreme left.



The relocation bit map will be created only if the load module mode is 'REL' (see the MODE directive). This directive is mandatory and must be the third linkage editor directive.

° MODE

Format

```
  MODE   |ABS|
         |   |;
         |REL|
```

Description

In the absence of the MODE directive, the mode of the load module will be relocatable unless:

- ° the ENTRY directive is given
- ° no relocatable text is found

Execution Start Address Specification

The starting address for execution of the load module produced by the GLE can be specified by the optional ENTRY directive.

- ENTRY

Format

```
ENTRY  |    module      |
       | ( |symbol| )   |
       |   |addr  |     |
```

Description

A start address may be specified by giving the name of an object module. If the module has an end transfer address specified, then this address will be used, otherwise the default end transfer address as supplied by the Meta Assembler will be used.

If a 'symbol' is used to specify the start address, then the definition of this symbol, as supplied by the LINKAGE EDITOR, will be used.

The use of 'addr' gives the user the ability to specify an absolute address for the start of execution. It must be described in the same base as the meta assembler output listing.

Module Appearance in a Load Module

The essence of module binding is the determination of the modules that will appear in the load module, the order in which they will appear in the load module, and the types of addresses that may be bound.

At this time, the LINKAGE EDITOR will be able to handle three addressing schemes provided by the Meta Assembler; direct memory addressing, base displaced addressing, and location counter relative addressing.

There are several user directives available to determine which object modules will appear in a load module; ULIB, SLIB, EXCLUDE, RENAME and INCLUDE. Even with the user directives, there are important assumptions that will be made when processing object modules using these directives.

The first assumption concerns the default processing of external references. If a module is needed for satisfaction of an external reference, then it will be searched for. The first place to look will be the 'task' table to see if it is already linked. If it is not linked, then the user library will be searched. If the user library does not contain the module, then the system library will be searched. If after searching the system library the module is still not found, then the reference will remain unsatisfied. So we see the search hierarchy is:

1) already linked
2) user library
3) system library

The search hierarchy may be changed by use of the FILES,NOULIB,NOSLIB and EXCLUDE directives.

° $\boxed{\text{FILES}}$

<u>Format</u>

```
FILES |[USER][,][SYSTEM]|;
```

<u>Description</u>

This directive indicates the files to be used in order to create the load module. This directive is mandatory and must be the second directive encountered.

° $\boxed{\text{INCLUDE}}$

Format

INCLUDE module [(msa)][,csect(csa)...][: $\left|\begin{array}{c}\text{ULIB}\\\text{SLIB}\end{array}\right|$];

Description

This directive causes the forced inclusion of 'module' from an optional library. The directive also allows a starting address, 'msa ', to be specified for the module. Additionally, assembly time control sections, 'csect', may have starting addresses specified. This directive has the power to determine not only order of appearance but starting addresses as well.

There are some restrictions depending upon the memory allocation scheme of the Meta Assembler. If the mode of the load module is defined as the"section"mode and 'msa' is specified, there will be a warning. However, the control section address will be allocated back-to-back for the specified module. If the mode is "normal", 'msa' can be specified but any control section address, 'csa', will be ignored if specified.

If this directive is not included in the creation of the load module, then ALL the modules from the user library will be included as a default.

All addresses must be specified in the base of the Meta Assembler output listing.

° $\boxed{\text{EXCLUDE}}$

Format

EXCLUDE modnam[,modnam...][: $\left|\begin{array}{c}\text{ULIB}\\\text{SLIB}\end{array}\right|$];

Description

This directive forces the exclusion of particular modules from appearance in the final load module. If no library is specified, then the module is ignored no matter which library it is found on. This affects the search hierarchy by implying which library may contain the module.

- NOULIB

  Format

  ```
  NOULIB;
  ```

  Description

  This directive forces exclusion of all modules in the user library from appearing in the final load module. This implies that the search hierarchy effectively becomes:

  1) already linked
  2) system library

- NOSLIB

  Format

  ```
  NOSLIB;
  ```

  Description

  This directive forces exclusion of all modules in the system library from appearing in the final load module. Therefore, the search hierarchy effectively becomes:

  1) already linked
  2) user library

- RENAME

  Format

  ```
  RENAME oldname=newname [,oldname=newname,...];
  ```

  Description

  This directive causes external references to 'oldname' to be satisfied by the definition supplied by 'newname'. If 'newname' is one of the external references to a module that has been mentioned on an EXCLUDE directive, then 'oldname' will not be renamed and will be left as undefined.

o GSECT

Format

```
GSECT csect,csa [,bound];
```

Description

The GSECT directive causes text in control section 'csect' from all linked modules to be linked consecutively into one global control section, starting at address 'csa'. Optionally included is the bounding information, 'bound', to be used to determine where to start addresses in this section when the next module is encountered.

If the memory allocation scheme is defined as the normal mode, then this GSECT directive will cause the error of memory overlapping.

The address must be described in the base of the Meta Assembler output listing.

o BOUND

Format

```
BOUND start [,next];
```

Description

The optional bound directive controls location counter processing for modules that are not supplied with starting addresses. The default values would cause modules to start at location 0 and be butted up against one another.

The address must be described in the base of the Meta Assembler output listing.

Generation of a Link Map

The user has control over the inclusion or exclusion of a link map as part of the LINKAGE EDITOR outputs. This control is available through the optional MAP directive.

- MAP

  Format

  ```
  MAP    [ON     ]
         [OFF    ];
         [GLOBAL ]
         [MODULE ]
  ```

  Description

  If the MAP directive is included without an operand or is not included, then the default information will be generated with the unsatisfied external map. When a link map is generated, the following fixed contents will be available. All addresses will be printed in the same base as the Meta Assembler output listing.

  - Default map
    - Echo of input directives
    - Error/warning messages
    - load module header information
      - Creation date and time
      - Load module kind
      - Load module length
      - Execution start address
    - Block assignments
      - Name of module and control section
      - Start address
      - Length
      - Library linked from
    - Relocation fields
  - Module map
    - External references
    - External definitions

- Global cross-reference map
  - Name of definition
  - Defined value
  - Module name defined
  - References to definition by module
- Unsatisfied external
  - External references
  - Module name referenced
  - References to externals

Example of Link Edit Use

The directives described previously imply a hierarchy of ordering on object modules and control sections. The simplest explanation of this hierarchy is through the use of an example.

Example 1. Show ordering hierarchy.
Assume the memory allocation scheme is the section mode and the base is octal.
Let object modules PG1, PG2, PG3 and PG4 exist.
PG1 contains A1, A3, B1 and B2 as control sections.
PG2 contains A1, A5, B0 and B2 as control sections.
PG3 contains A0, A2, A7 and B1 as control sections.
PG4 contains A8 as a control section.
PG2 and PG4 are needed to satisfy external references.
Given the following directives, show the starting addresses.

```
LINKEDIT;
     FILES USER;
     RELOCATION=(0:11),(12:23);
     NAME=LMOD;
     MODE ABS;
     BOUND 5000;
     GLOBAL A1,100,2;
     GLOBAL B1,1000;
     INCLUDE PG1;
     INCLUDE PG3(200), A7(7000);
END;
```

100	200	1000	5000	8000
A1(PG1)	A0(PG3)	B1(PG1)	A2(PG1)	A7(PG3)
A1(PG2)	A2(PG3)	B1(PG3)	B2(PG1)	
			A5(PG2)	
			B0(PG2)	
			B2(PG2)	
			A8(PG4)	

3.3.3.5 Use of the Catalog Function

During use of the Meta Assembler system, many files will be created along the path to load module generation. Some of these files, such as libraries, will be saved and used many times. To aid the user with configuration control, a catalog function is provided by the GLE. This function extracts descriptive information about the three basic Meta Assembler system outputs object modules, libraries of object modules, and load modules.

Summary of Catalog Directives

° CATALOG	Invoke catalog service
° FILES	Specify which files are to be cataloged
° END	Terminate catalog function

Detailed Description of Catalog Directives

° $\boxed{\text{CATALOG}}$

Format

CATALOG;

Description

Mandatory directive required to invoke the CATALOG function.

° $\boxed{\text{FILES}}$

Format

```
FILES = |filename    |    [/F] [, |filename    |   [/F],... ] ;
        |logical unit|            |logical unit|
```

Description

During a GLE run, several files are created. Before the LIBRARY function, a file of object modules generated by the Meta Assembler, known as OBJ, and optionally an old library of object modules to update, known as OLIB, exist. After the LIBRARY function, a new library, known as NLIB, exists. Before the LINKAGE EDITOR function, a user and/or system library, known as ULIB and SLIB, respectively, exist. After the LINKAGE EDITOR function, a load module, known as LMOD, exists.

So, at any of the described points, several files with generic names are available for cataloging. In addition to their generic names, the files will also have a FORTRAN logical unit associated with them. The table below describes the 'filename' and its corresponding 'logical unit'.

The '/F' indicates the full catalog for the file mentioned.

FILENAME	LOGICAL UNIT
OBJ	8
OLIB	7
NLIB	9
ULIB	9
SLIB	11
LMOD	12

° $\boxed{\text{END}}$

Format

END;

Description

This directive causes the CATALOG function to perform the catalog of files.

Available Information

The information that is available for each of the three basic files is shown below.

Object Module

° Object Module Description (DSC)	
° Control Section Dictionary (CSD)	: '/F' only
° External Reference Directionary (ERD)	: '/F' only
° External Definition Dictionary (EDD)	: '/F' only
° Vector Symbol Dictionary (VSD)	: '/F' only
° Object Text (TXT)	: '/F' only
° Object Module End (END)	: '/F' only
° Object Module EOF (EOF)	

Library of Object Modules

° Library Header		
° Module Name List	:	'/F' only
° Object Module Description (DSC)		
° Control Section Dictionary (CSD)	:	'/F' only
° External Reference Dictionary (ERD)	:	'/F' only
° Vector Symbol Dictionary (VSD)	:	'/F' only
° End Marker	:	'/F' only
° Object Text (TXT)	:	'/F' only
° Object Module End (END)	:	'/F' only
° Object Module EOF (EOF)		

Load Module

- ° Load Module Header
- ° Relocation Address Fields
- ° Text Bit Strings with Relocation Bit Map : '/F' only
- ° End of Load Module

Examples of Catalog Use

Example 1. Catalog all files after load module generation

```
Directives: LIBRARY;
              CREATE;
              NAME=EXLIB;KIND=USER;
            END;
              LINKEDIT;
                FILES USER;
                RELOCATION=(0:11),(12:23);
                ENTRY MAIN;
                INCLUDE MAIN (0):  ULIB;
              END;
              CATALOG;
                FILES=OBJ/F,NLIB,9/F, LMOD;
              END;
```

Note: File OLIB is not cataloged because the library function operated in a creation not a maintenance mode.

Example 2. Catalog of an unknown file on FORTRAN logical unit 8 to determine its type.

Directives:
```
CATALOG;
  FILES=8;
END;
```

3.4 INSTALLATION AND TRAINING

This section describes the delivery, installation and training procedures for the products developed under this contract. The facility utilized for installation and training was MSFC at NASA request.

3.4.1 Task 7 - NASA MSFC Delivery

The enhanced Meta Assembler, developed under Task 1, and the Linkage Editor, developed under Task 5 was installed at NASA MSFC on an IBM 360 (see Figure 3). To provide system maintenance capability at MSFC the MDAC proprietary Meta Translator was alos be installed on the IBM 360. The delivery consisted of the following:

- Installation on the MSFC IBM 360
- MSFC Installation Verification
- Meta Assembler System/Meta Translator Demonstration
- Personnel Training at MSFC
- MSFC Deliverable Items

3.4.2 Installation on the MSFC IBM 360

THe MSFC IBM 360 was selected as the host machine for the installation of the enhanced Meta Assembler, Linkage Editor, NSSC-1 target output driver, and MDAC Meta Translator. The procedures to perform the installation of the enhanced Meta Assembler, Linkage Editor, NSSC-1 target output driver, and MDAC Meta Translator were:

- to develop IBM 360 JCL for file creation, Meta Translation, FORTRAN compilation, link edit and execution of the components of the Meta Assembler system.
- to determine the overlay structure for the Meta Assembler
- to meta translate the Meta Assembler component meta language descriptions
- to compile the Meta Assembler FORTRAN source
- to link edit the Meta Assembler system object modules

3.4.3 MSFC Installation Verification

The installation verification was performed utilizing standard test cases for the Meta Assembler system and the meta language definition of the Meta Assembler for the Meta Translator. The verification procedure exercised each Meta Assembler system program involving the NSSC-1 assembler creation. The Meta Translator was verified by regenerating the Meta Assembler parsing subroutines via meta language processing.

3.4.4 Meta Assembler System/Meta Translator Demonstration

The Meta Assembler system and the Meta Translator demonstration consisted of reproducing the verification process utilizing the standard test cases and the Meta Assembler meta language definition.

3.4.5 Demonstration for the NSSC-I

The system was demonstrated as fully supporting assembly level software development for the NSSC-I. This was performed via cross assembly of GSFC supplied NSSC-I programs, object module link edit, and load module formatting.

The NSSC-I assembler language definition in ALLDEF and ALLLEX and processing by both the ALLDEF and ALLLEX processors was also demonstrated. Since a NSSC-I computer was not available at MSFC, actual execution could not be performed.

3.4.6 Personnel Training at MSFC

A period of one week was allocated for personnel training at MSFC. The primary thrust of this training period was toward Meta Assembler system maintenance. Items addressed were:

- ALLDEF processor design and use
- ALLLEX processor design and use
- Meta Assembler design and use
- Linkage Editor design and use
- Meta Translator Utilization

3.4.7 MSFC Deliverable Items

All installation support materials were included in the delivery as follows:

- Meta Assembler system FORTRAN source on magnetic tape
- Meta Assembler system program listings
- Meta Assembler meta language source on magnetic tape
- Meta Translator FORTRAN source on magnetic tape
- Meta Translator User's Manual
- Installation procedure documentation

Available Meta Assembler system user oriented documentation was delivered at this time. This delivery task, however, preceded the formal documentation development. All formal documentation, Task 4, and final product versions will be made available to MSFC for subsequent installation.

3.4.8 GSFC Deliverable Items

Due to the cancellation of the GSFC installation at NASA request, items which were scheduled for this delivery were delivered to MSFC. These items, all on magnetic tape were:

- NSSC-1 ALLDEF source
- NSSC-1 ALLLEX source
- NSSC-1 target output driver source
- GSFC furnished test cases

3.5 META ASSEMBLER DOCUMENTATION

This section pertains to the Meta Assembler system documentation developed under Task 4. The two types of documentation produced are:

- User Manuals
- Detail Design Manuals

3.5.1 User Manuals

Comprehensive user manuals were developed for each of the Meta Assembler system programs including:

- ALLDEF User Manual
- ALLLEX User Manual
- Meta Assembler User Manual
- Linkage Editor User Manual

The content of the user manuals is presented in a topical narrative fashion and thoroughly discusses the user interface considerations including:

- product overview/capabilities
- detailed presentation of user interface (control cards, language statements, etc.)
- extensive examples of user interface
- assumptions and restrictions
- diagnostics

3.5.2 Detail Design Manuals

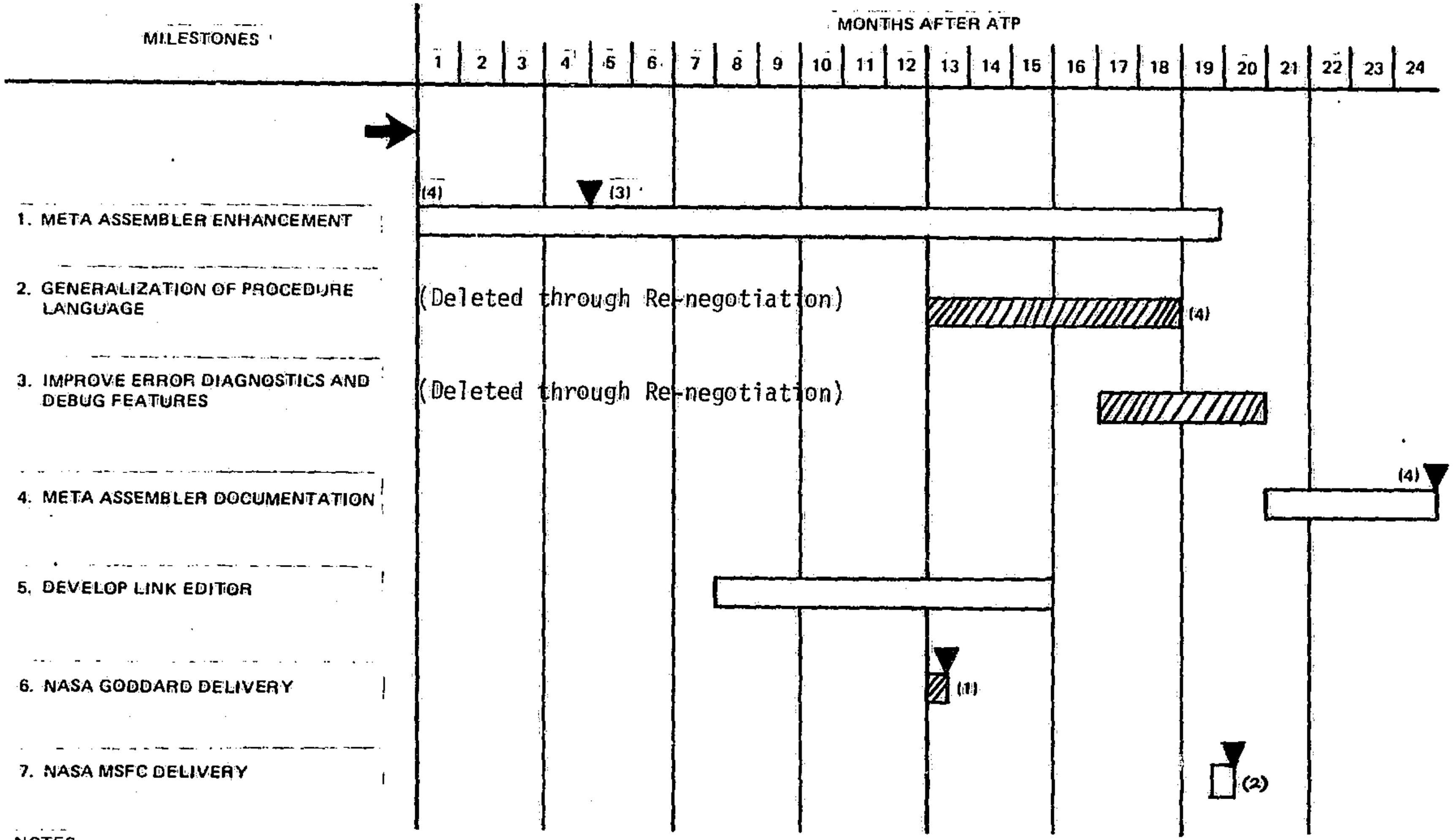
To support the maintenance aspect of the Meta Assembler system, detailed design documentation was developed for the following programs:

- ALLDEF processor
- ALLLEX processor
- Meta Assembler
- Linkage Editor
- NSSC-I target output driver

The content of the detail design manuals is presented with a blend of topical narrative discussions and supporting schematic representations including:

- program capabilities
- functional flow chart
- block structure diagram
- input/output description
- global data area description
- subroutine summary
  - function description
  - local data description
  - system interface requirements
- host installation procedures
  - machine dependent considerations

APPENDIX A
SCHEDULE/MILESTONES



NOTES

▼ TASK 1 - DESIGN REVIEW FOUR MONTHS AFTER ATP
TASK 4 - FINAL META ASSEMBLER SYSTEM AND DOCUMENTATION DELIVERY
TASK 6 - DELIVERY OF NSSC-I META ASSEMBLER AND GENERALIZED LINKAGE EDITOR TO GSFC CANCELLED AT NASA REQUEST
TASK 7 - DELIVERY OF META TRANSLATOR TO MSFC AND TASK 6 ITEM TO MSFC

(1) ONE (1) TRIP TO GSFC OF TWO (2) WEEKS DURATION CANCELLED
(2) ONE (1) TRIP TO MSFC OF TWO (2) WEEKS DURATION
(3) ONE (1) TRIP TO MSFC OF ONE (1) WEEK DURATION
(4) THREE (3) TRIPS TO MSFC OF THREE (3) DAYS DURATION EACH

Project Schedule

Appendix B
ALLDEF NSSC-I DEFINITION

```
/* DEFINITION OF NSSC-1 ASSEMBLER */

/* MACHINE DESCRIPTION AND ENVIORNMENT */

  OPTION : PAGE-LENGTH = 56,
           DATE = 8/11/78 ,
           COMPUTER = NSSC-1 ,
           CONTINUATION = NO ,
           BOUNDING = YES ,
           ERROR-SIZE = 18 ,
           UNDEFINED-EXTERNALS = YES,
           LIST-BASE = 8 $
  SIZE : ADDRESS-UNIT = 18,
         ACCESS-UNIT = 18,
         MEM-SIZE = 4096 $

/* USER DEFINED TYPES */

TYPE 'NOLIST','NOPROC' $
TYPE 'MAJOR', 'MINOR' $
TYPE 'DIRECT', 'INDIRCT' $
TYPE 'LIT' $
TYPE 'CONTROL-COUNTER', 'CCV' $

/* LOCAL AND GLOBAL VARIABLE DEFINITIONS */

DEFAULT MNEMONIC 'DATA','NONE'$
LOCAL 'ANSWER' = Ø $
LOCAL 'DEF' = 1 $
LOCAL 'EXTERNL' = Ø $
LOCAL 'INDIRECT' = Ø $
LOCAL 'LIT-FLAG' $
LOCAL 'SECTION-SAVE','LOCATION-SAVE' $
LOCAL 'LITERAL-SCAN' = Ø $
GLOBAL 'DOLLAR-SEC','DOLLAR-LOC' $
GLOBAL 'MACRO-LINE' = Ø $
GLOBAL 'BUILD' $
GLOBAL 'FIRST-CARD' = 1 $
GLOBAL 'LABEL-FIELD' $
GLOBAL 'STRING' $
GLOBAL 'MS' $
GLOBAL 'ML' $
GLOBAL 'PS' $
GLOBAL 'PL' $
GLOBAL 'STANDARD' = 1, 'SOURCE-LIBRARY' = 2,
       'REPEAT-ARRAY' = 3, 'MACRO-EXPAND' = 4,
       'LITERAL-POOL' = 5, 'DEF-LIT-POOL' = 6 $
GLOBAL 'NORMAL' = 1, 'SKIP' = 2, 'REPEAT-FILL' = 3,
       'MACRO-BUILD' = 4 $

/* USER DEFINED SEMANTIC FUNCTIONS */

SEMANTIC 'DEFLABEL' :
        IF (PRESENT(OPERAND(1))),
           IF (SYMBOL←TYPE(OPERAND(1)).EQ.UNDEFINED),
           ELSE,
              ERROR(35),
           END,
           CREATE←SYMBOL(OPERAND(1)),
           IF (EXTERNL.EQ.1),
              CREATE←REFDEF(OPERAND(1),DEF),
           END,
        END $
     SEMANTIC 'MAJOP' :
        BIT←LENGTH=18,
        DEFLABEL(OPERAND(1)),
        OBJECT(ADDRESS←TYPE(LOCATION←COUNTER),
               FIELD(0:5)=OPCODE,
               FIELD(5:5)=INDIRECT,
               FIELD(6:17)=OPERAND(2)) $
     SEMANTIC 'MINOP' :
        BIT←LENGTH=18,
        DEFLABEL(OPERAND(1)),
        OBJECT(ADDRESS←TYPE(LOCATION←COUNTER),
               FIELD(0:11)=0,FIELD(12:17)=OPCODE) $
     SEMANTIC 'ORGSEM' :
        ANSWER = VALUE(OPERAND(1,1)),
        IF (ANSWER.EQ.0),
           SECTION('DATA'),
        ELSE,
           IF (ANSWER.EQ.1),
              SECTION('CODE'),
           ELSE,
              FAIL,
           END,
        END,
        SET←ORIGIN(OPERAND(1,2),*) $

     /* SPECIAL KINDS OF DEFINITIONS */

     GROUP BEGIN SYMBOL '(' $
     GROUP END SYMBOL ')' $
     END STATEMENT SYMBOL '.EOL.' $
     END MODULE SYMBOL 'END' : SEMANTIC = END←MODULE,
                                          SOURCE←MODE=DEF←LIT←POOL $
    5
     INSTRUCTION 'MACRO..CALL' :
        OPERAND(1) = OPTIONAL LABEL TERM,
        OPERAND(2) = SYMBOL TERM LIST,
        SEMANTIC = DEFLABEL(OPERAND(1)),
                        STACK←PARM(OPERAND(2)),
                       MACRO←LINE=1 $

     /* DIRECTIVE AND PSEUDO OP DEFINITIONS */

     DIRECTIVE 'DATA' :
        OPERAND(1) = OPTIONAL LABEL TERM,
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        OPERAND(2) = ANY EXPRESSION LIST,
        SEMANTIC   = DEFLABEL(OPERAND(1)),
                     CREATE-DATA(*,OPERAND(2)) $
    UNLABELED DIRECTIVE 'ASSEMBLE' :
          OPERAND(1) = ADDRESS TERM,
            SEMANTIC = START(OPERAND(1)),
                       SECTION('DATA'),
                       SET-LITERAL-POOL('DATA'),
                       SOURCE-MODE=STANDARD $
     DIRECTIVE 'RES' :
        OPERAND(1) = OPTIONAL LABEL TERM,
        OPERAND(2) = ANY EXPRESSION,
        SEMANTIC   = DEFLABEL(OPERAND(1)),
                     RESERVE(OPERAND(2),*,*) $
    DIRECTIVE 'EQU' :
            OPERAND(1)=LABEL TERM,
            OPERAND(2)=ANY EXPRESSION,
        SEMANTICS =
            DEFLABEL(OPERAND(1)),EQUATE(OPERAND(1),OPERAND(2),1) $
     UNLABELED DIRECTIVE 'LIT' :
          OPERAND(1) = VALUE EXPRESSION,
          SEMANTIC = ANSWER=OPERAND(1).MOD.2,
                     IF (ANSWER.EQ.0),
                         SET-LITERAL-POOL('DATA'),
                     ELSE,
                         IF (ANSWER.EQ.1),
                             SET-LITERAL-POOL('CODE'),
                         ELSE,
                             FAIL,
                         END,
                     END $
     UNLABELED DIRECTIVE 'PAGE' :
          SEMANTIC = EJECT-PAGE $
     UNLABELED DIRECTIVE 'LIST' :
        SEMANTIC   = LISTING=1; PRINT(1) $
     UNLABELED DIRECTIVE 'UNLS' :
        SEMANTIC   = LISTING=0; PRINT(0) $
     DIRECTIVE 'PROC' :
        OPERAND(1)=LABEL TERM,
        SEMANTIC = PROCESS-MODE=SKIP $
     UNLABELED DIRECTIVE 'END' :
       SEMANTIC = IF (PROCESS-MODE .EQ. SKIP),
                      PROCESS-MODE = NORMAL,
                  ELSE,
                      FAIL,
                  END $

                                                     *********** WARNING ***********
                                                     END REDEFINED AS ANOTHER WORD KIND
     UNLABELED DIRECTIVE 'PEND' :
        SEMANTIC = IF(SOURCE-MODE.EQ.MACRO-EXPAND),
                       END-MACRO,
                       SOURCE-MODE=STANDARD,
                       MACRO-LINE=0,
                ELSE,
                   FAIL,
                ENO $
UNLABELED DIRECTIVE 'AORG' :
   OPERAND(1) = ANY EXPRESSION LIST,
  SEMANTIC = IF (SYMBOL-TYPE(OPERAND(1,1)).EQ.ABSOLUTE.AND.
                 SYMBOL-TYPE(OPERAND(1,2)).EQ.ABSOLUTE),
                  ORGSEM(OPERAND(1)),
              ELSE,
                  FAIL,
           END $
UNLABELED DIRECTIVE 'RORG' :
 OPERAND(1) = ANY EXPRESSION LIST,
 SEMANTIC = IF (SYMBOL-TYPE(OPERAND(1,1)).EQ.ABSOLUTE),
                     ORGSEM(OPERAND(1)),
                  ELSE,
                     FAIL;
                  END $
ERROR MESSAGE :
   NUMBER      = 35,
   LEVEL       = 1,
   'DUPLICATE LABEL' $

ERROR MESSAGE :
   L           = 1,
   N           = 25,
   'ILLEGAL CONTROL SECTION' $

NOUN '$' :
     RESULT=ADDRESS TERM,
     SEMANTIC = IF (SUBFIELD.EQ.OPERAND-FIELD),
                  IF (SOURCE-MODE.EQ.LITERAL-POOL.OR.
                       SOURCE-MODE.EQ.DEF-LIT-POOL),
                       SECTION-SAVE=CTL-SECTION,
                       LOCATION-SAVE=LOCATION,
                       CTL-SECTION=DOLLAR-SEC,
                       LOCATION=DOLLAR-LOC,
                       RETURN(LOCATION),
                       LOCATION=LOCATION-SAVE,
                       CTL-SECTION=SECTION-SAVE,
                  ELSE,
                       RETURN(LOCATION),
                  END,
                ELSE,
                  FAIL,
                END $
INSTRUCTION 'NONE' :
      OPERAND(1) = CONTROL-COUNTER $                        ********** WARNING **********
                                                            NO SEMANTICS SPECIFIED

INSTRUCTION 'NONE':
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC = DEFLABEL(OPERAND(1)) $
```

```
/* EXECUTABLE INSTRUCTION DEFINITIONS */

INSTRUCTION 'FLP' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'22',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'LOD' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'13',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'LDP' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'12',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'NEG' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'04',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'ADC' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'06',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'CMP' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'10',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'NORM' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'14',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'ACX' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'25',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'XAX' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'25',
                 MINOP(OPERAND(1)) $
INSTRUCTION 'AEA' :
   RESULT      = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC    = OPCODE=O'26',
                 MINOP(OPERAND(1)) $
```

```
INSTRUCTION 'XAE' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'26',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'EAX' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'27',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'REVERSE-EAX' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'27',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'HLT' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'00',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'NOP' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'02',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'EXIT' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'16',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'TOV' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'01',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'TAP' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'03',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'TOP' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'05',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'ROV' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
   SEMANTIC     = OPCODE=O'07',
                  MINOP(OPERAND(1)) $
INSTRUCTION 'CPO' :
   RESULT       = MINOR,
   OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'17',
                     MINOP(OPERAND(1)) $
   INSTRUCTION 'SIO' :
      RESULT       = MINOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'20',
                     MINOP(OPERAND(1)) $
   INSTRUCTION 'TAZ' :
      RESULT       = MINOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'21',
                     MINOP(OPERAND(1)) $
   INSTRUCTION 'RED' :
      RESULT       = MINOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'23',
                     MINOP(OPERAND(1)) $
   INSTRUCTION 'RIO' :
      RESULT       = MINOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'24',
                     MINOP(OPERAND(1)) $
   INSTRUCTION 'TIX' :
      RESULT       = MINOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'11',
                     MINOP(OPERAND(1)) $
   INSTRUCTION 'TIE' :
      RESULT       = MINOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      SEMANTIC     = OPCODE=O'15',
                     MINOP(OPERAND(1)) $
   NOUN ',' :
      SEMANTIC     = IF (SUBFIELD.EQ.OPCODE-FIELD),
                         INDIRECT=1,
                     ELSE,
                         FAIL,
                     END $
   INSTRUCTION 'LOA' :
      RESULT       = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC     = OPCODE=O'20',
                     MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'LOL' :
      RESULT       = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC     = OPCODE=O'40',
                     MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'LOI' :
      RESULT       = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'12',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'LDE' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'52',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'LDX' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'54',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'STA' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'6Ø',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'STI' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'32',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'STE' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'1Ø',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'STX' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'74',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'ADX' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'Ø2',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'ADD' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'Ø4',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'SUB' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'24',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'MUL' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'44',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'DIV' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'64',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'ETR' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'20',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'AND' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'20',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'MRG' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'50',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'OR' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'50',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'EOR' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'70',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'OPT' :
      RESULT     = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC   = OPCODE=O'16',
                   MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'IPF' :
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'76',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'SHF' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'14',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'DSH' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'36',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'DCY' :
          RESULT     = MAJOR,
          OPERAND(1) = OPTIONAL LABEL TERM,
          OPERAND(2) = ANY EXPRESSION,
          SEMANTIC    = OPCODE = O'56',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'CYC' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'34',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'BRM' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'06',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'BRU' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'62',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'BRC' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'42',
                    MAJOP(OPERAND(1),OPERAND(2)) $
   INSTRUCTION 'TIN' :
      RESULT      = MAJOR,
      OPERAND(1) = OPTIONAL LABEL TERM,
      OPERAND(2) = ANY EXPRESSION,
      SEMANTIC    = OPCODE=O'72',
                    MAJOP(OPERAND(1),OPERAND(2)) $
```

```
        INSTRUCTION 'TXLE' :
           RESULT      = MAJOR,
           OPERAND(1)  = OPTIONAL LABEL TERM,
           OPERAND(2)  = ANY EXPRESSION,
           SEMANTIC    = OPCODE=O'22',
                         MAJOP(OPERAND(1),OPERAND(2)) $
        INSTRUCTION 'TAL' :
           RESULT      = MAJOR,
           OPERAND(1)  = OPTIONAL LABEL TERM,
           OPERAND(2)  = ANY EXPRESSION,
           SEMANTIC    = OPCODE=O'26',
                         MAJOP(OPERAND(1),OPERAND(2)) $
        INSTRUCTION 'TAE' :
           RESULT      = MAJOR,
           OPERAND(1)  = OPTIONAL LABEL TERM,
           OPERAND(2)  = ANY EXPRESSION,
           SEMANTIC    = OPCODE=O'46',
                         MAJOP(OPERAND(1),OPERAND(2)) $
        INSTRUCTION 'TAG' :
           RESULT      = MAJOR,
           OPERAND(1)  = OPTIONAL LABEL TERM,
           OPERAND(2)  = ANY EXPRESSION,
           SEMANTIC    = OPCODE=O'66',
                         MAJOP(OPERAND(1),OPERAND(2)) $

        /* SYMBOL DEFINITIONS TO SUPPORT ASSEMBLY */

        PREFIX OPERATOR '(' :
           RESULT      = ANY EXPRESSION,
           OPERAND(1)  = LIT,
           SEMANTIC    = ANSWER=CHECKOP(KIND,1),
                         IF (ANSWER.EQ.INSTRUCTION.OR.ANSWER.EQ.DIRECTIVE),
                            LITERAL('DATA',OPERAND(1),*,LIT*FLAG),
                            RETURN(OPERAND(1)),
                         ELSE,
                            FAIL,
                         END $
                                                                ********** WARNING **********
                                                                PRECEDENCE NOT SPECIFIED 25 USED
        PREFIX OPERATOR '$(' :
           RESULT      = CONTROL_COUNTER,
           PRECEDENCE  = 10,
           OPERAND(1)  = CCV,
           SEMANTIC    = ANSWER=VALUE(OPERAND(1)),
                         IF (ANSWER.EQ.0),
                            SECTION('DATA'),
                         ELSE,
                            IF (ANSWER.EQ.1),
                               SECTION('CODE'),
                            ELSE,
                               ERROR(25),
                               FAIL,
                            END,
                 END $
POSTFIX OPERATOR ')' :
    RESULT      = CCV,
    OPERAND(1)  = ANY EXPRESSION,
    SEMANTIC    = IF (CHECKOP(SPELLING,1).EQ.'$('),
                      RETURN(OPERAND(1)),
                  ELSE,
                      FAIL;
                  END $

                                                              ********** WARNING **********
INFIX OPERATOR ',' :                                          PRECEDENCE NOT SPECIFIED 1ØØØ USED
    RESULT      = LABEL TERM,
    PRECEDENCE  = 1Ø,
    OPERAND(1)  = CONTROL.COUNTER,
    OPERAND(2)  = LABEL TERM,
    SEMANTIC    = RETURN(OPERAND(2)) $
POSTFIX OPERATOR ')' :
    RESULT      = LIT,
    OPERAND(1)  = SYMBOL TERM,
    SEMANTIC    = RETURN(OPERAND(1)) $

                                                              ********** WARNING **********
POSTFIX OPERATOR '*' :                                        PRECEDENCE NOT SPECIFIED 1ØØØ USED
    RESULT      = LABEL TERM,
    OPERAND(1)  = LABEL TERM,
    SEMANTIC    = EXTERNL=1,
                  RETURN(OPERAND(1)) $
                                                              ********** WARNING **********
                                                              PRECEDENCE NOT SPECIFIED 1ØØØ USED
INFIX OPERATOR '+' :
    RESULT      = ANY EXPRESSION,
    OPERAND(1)  = ANY EXPRESSION,
    OPERAND(2)  = ANY EXPRESSION,
    PRECEDENCE  = 4Ø,
    SEMANTIC    = RETURN(OPERAND(1)+OPERAND(2)) $
INFIX OPERATOR '-' :
    RESULT      = ANY EXPRESSION,
    OPERAND(1)  = ANY EXPRESSION,
    OPERAND(2)  = ANY EXPRESSION,
    PRECEDENCE  = 4Ø,
    SEMANTIC    = RETURN(OPERAND(1)-OPERAND(2)) $
PREFIX OPERATOR '+' :
    RESULT      = ANY EXPRESSION,
    OPERAND(1)  = ANY EXPRESSION,
    PRECEDENCE  = 4Ø,
    SEMANTIC    = RETURN(OPERAND(1))$
PREFIX OPERATOR '-' :
    RESULT      = ANY EXPRESSION,
    OPERAND(1)  = ANY EXPRESSION,
    PRECEDENCE  = 4Ø,
    SEMANTIC    = RETURN(Ø-OPERAND(1))$
INFIX OPERATOR '*' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 50,
        SEMANTIC     = RETURN(OPERAND(1)*OPERAND(2))$
    INFIX OPERATOR '/' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 50,
        SEMANTIC     = RETURN(OPERAND(1)/OPERAND(2))$
    INFIX OPERATOR '_-' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 20,
        SEMANTIC     = RETURN(OPERAND(1).XOR.OPERAND(2))$
    INFIX OPERATOR '>' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 15,
        SEMANTIC     = RETURN(OPERAND(1).GT.OPERAND(2))$
    INFIX OPERATOR '<' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 15,
        SEMANTIC     = RETURN(OPERAND(1).LT.OPERAND(2))$
    INFIX OPERATOR '=' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 15,
        SEMANTIC     = RETURN(OPERAND(1).EQ.OPERAND(2))$
    INFIX OPERATOR '*/*' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 60,
        SEMANTIC     = RETURN(.SHL.(OPERAND(1),OPERAND(2)))$
    INFIX OPERATOR '*+' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 60,
        SEMANTIC     = RETURN(OPERAND(1)*(10**OPERAND(2)))$
    INFIX OPERATOR '*-' :
        RESULT       = ANY EXPRESSION,
        OPERAND(1)   = ANY EXPRESSION,
        OPERAND(2)   = ANY EXPRESSION,
        PRECEDENCE   = 60,
        SEMANTIC     = RETURN(OPERAND(1)/10**OPERAND(2))$
```

```
INFIX OPERATOR '.*' :
   RESULT       = ANY EXPRESSION,
   OPERAND(1) = ANY EXPRESSION,
   OPERAND(2) = ANY EXPRESSION,
   PRECEDENCE = 3Ø,
   SEMANTIC     = RETURN(OPERAND(1).AND.OPERAND(2))$
INFIX OPERATOR '+.' :
   RESULT       = ANY EXPRESSION,
   OPERAND(1) = ANY EXPRESSION,
   OPERAND(2) = ANY EXPRESSION,
   PRECEDENCE = 2Ø,
   SEMANTIC     = RETURN(OPERAND(1).OR.OPERAND(2))$
INFIX OPERATOR './_' :
   RESULT       = ANY EXPRESSION,
   OPERAND(1) = ANY EXPRESSION,
   OPERAND(2)= ANY EXPRESSION,
   PRECEDENCE = 6Ø,
   SEMANTIC     = RETURN(.SHR.(OPERAND(1),OPERAND(2)))$
INFIX OPERATOR ',':
   RESULT       = SYMBOL TERM LIST,
   OPERAND(1) = SYMBOL TERM LIST,
   OPERAND(2) = SYMBOL TERM,
   PRECEDENCE = 12,
   SEMANTIC     = LISTF(OPERAND(1),OPERAND(2)) $
INFIX OPERATOR ',' :
   RESULT       = ANY EXPRESSION LIST,
   OPERAND(1) = ANY EXPRESSION LIST,
   OPERAND(2) = ANY EXPRESSION,
   PRECEDENCE = 12,
   SEMANTIC     = LISTF(OPERAND(1),OPERAND(2)) $

END OF ALLDEF DEFINITION $
```

```
TOTAL NUMBER OF RECOGNITION ERRORS =    Ø
```

Appendix C

ALLLEX NSSC-I DEFINITION

```
BEGIN LEXICAL DEFINITION $
<LEXICON> :=
         IF FIRST-CARD NE 0,                       /* CHECK FOR ASSEMBLE CARD  */
         <FIRSTCARD>
     //(
         IF MACROS EQ 0                            /* IS A MACRO PASS NEEDED   */
      // <MACROPREPASS>,                           /* GET ALL MACROS           */
         MACROS=0,
         CURSOR=1
      ),
      IF SUBFIELD LE FIELDS,                       /* UNTIL ALL FIELDS         */
      (
         IF NOT ' ',                               /* NEXT FIELD IF BLANK OR   */
         IF NOT '.',                              /* COMMENT                  */
         <TOKEN>,                                  /* GET A TOKEN              */
         (
               IF MNEMONIC EQ 0,                   /* ONLY FOR ACTUAL          */
               TOKEN-START=POSITION OF <TOKEN>,
               TOKEN-SIZE=SIZE OF <TOKEN>
          // NULL
          )
        // IF SUBFIELD EQ 1,                      /* CHECK FOR NULL LINE      */
             SCAN,
             (
              '.',
              CURSOR=LENGTH
           // IF CURSOR-CHAR EQ =999
           ),
           MNEMONIC=2,
           TOKEN-TYPE=NAME;
           SUBFIELD=2
     // SCAN,                                      /* GET NEXT FIELD           */
        (
          '.',                                    /* IGNORE REST IF COMMENT   */
          CURSOR=LENGTH
       // IF CURSOR-CHAR EQ =999                   /* OR REACHED END OF LINE   */
       // SUBFIELD=SUBFIELD+1                      /* REACHED NEXT FIELD       */
        ),
        <LEXICON>                                  /* CONTINUE WITH NEW TOKEN  */
      )
       //        CURSOR=LENGTH,                    /* JUST GET OUT IF TOO      */
                                                   /* MANY FIELDS              */
      42
                 TOKEN-TYPE=END-OF-LINE $
<TOKEN> ::=
      IF SOURCE-MODE EQ MACRO-EXPAND,             /* PARAMETER SUBSTITUTION   */
      IF MACRO-LINE EQ 0;                           /* MUST BE FIRST LINE     */
      IF SUBFIELD EQ 3,                           /* IN ARGUMENT FIELD        */
      <MACROARG>,                                   /* RETURN AS A SYMBOL     */
      TOKEN-TYPE=SYMBOL
   // IF PROCESS-MODE EQ SKIP,                    /* IGNORE TEXT OF MACRO     */
        <SKIPMACROTEXT>
   // IF LITERAL-SCAN EQ 1;                       /* PARSE LITERAL            */
      <ZEROLEVELPAREN>,
          LITERAL-SCANER,
          TOKEN-TYPE=SYMBOL
            // IF LETTER,
          <NAME>,                                       /* CHECK FOR A NAME          */
          (
             IF SUBFIELD EQ 1,                          /* IS THIS A LABEL           */
             TOKEN-TYPE=LABEL
          // TOKEN-TYPE=NAME                            /* NO, JUST A NAME           */
          )
       // IF DIGIT,                                     /* CHECK FOR A NUMBER        */
         <NUMBER>
       // IF SPECIAL,                                   /* CHECK FOR SPECIALS        */
          <SPECIAL>,
          TOKEN-TYPE=SPECIAL
       // IF ALMERIC,                                   /* UNKNOWN SYMBOL            */
          <SYMBOL>,
          TOKEN-TYPE=SYMBOL
       // IF CURSOR-CHAR EQ -999,                       /* END OF LINE TOKEN         */
          CURSOR=CURSOR+1,
          TOKEN-TYPE=END-OF-LINE $
<NUMBER> :=
          IF SUBFIELD EQ 2,
          MNEMONIC=1,
          TOKEN-TYPE=NAME,
            CURSOR=CURSOR-1
      // (
            IF '0',                                       /* A LEADING 0 MEANS OCTAL  */
          <OCT>,
          TOKEN-VALUE=VALUE OF OCTAL <OCT>
      // <INT>,                                         /* IF NOT THEN DECIMAL       */
          TOKEN-VALUE=VALUE OF <INT>
          ),
          TOKEN-TYPE=VALUE $
<MACROARG> :=
            1 TO 20
            (
                      IF NOT ' ',
                      IF NOT ';',
                      CHARACTER
            ) $
<SKIPMACROTEXT> :=
            (
                      IF SUBFIELD EQ 1,
                      <NAME>,
                      SCAN,
                      SUBFIELD=2
            //        NULL
            ),
            (
                      IF SUBFIELD EQ 2,
                      IF CURSOR-CHAR NE -999,
                      IF NOT 'END ',
                      CURSOR=LENGTH,
                  MNEMONIC=2;
                  TOKEN+TYPE=NAME
          //      IF SUBFIELD NE 2
           ) $
  <SYMBOL> :=
        1 TO MANY ALMERICS $                /* LETTERS AND/OR DIGITS    */
  <INT> :=
        1 TO 12 DIGITS $                    /* DECIMAL DIGITS           */
  <OCT> :=
        1 TO 12 OCTALS $                    /* OCTAL DIGITS             */
  <NAME> :=
        1 TO 8 LETMERICS $                  /* LETTER AND LETTERS       */
                                            /* AND/OR DIGITS            */
  <SPECIAL> :=
        '*',
        (
           '/',
           (
              '+'                           /* A SHIFTED LEFT N PLACES  */
          //  '-'                           /* A SHIFTED RIGHT N PLACES */
           )
       // '+'                               /* A MULTIPLIED BY 10**N    */
       // '-'                               /* A MULTIPLIED BY 1/10**N  */
       // '&'                               /* LOGICAL AND              */
       // NULL                              /* MULTIPLICATION           */
        )
    // '/',
       (
          '/'
       // NULL                              /* REMAINDER OF A/N         */
       )                                    /* DIVISION                 */
    // '+',
       (
          '+'
       // NULL                              /* LOGICAL OR               */
       )                                    /* ADDITION                 */
    // '-',
       (
          '-'
       // NULL                              /* EXCLUSIVE OR             */
       )                                    /* SUBTRACTION              */
    // IF SUBFIELD EQ 1,
       '$('                                 /* CONTROL SECTION INDICATOR*/
    // IF SUBFIELD EQ 3,                    /* CHECK IF ARGUMENT FIELD  */
                                            /* THEN POSSIBLE LITERAL    */
       '(',                                 /* LEFT PAREN FOLLOWED BY   */
       (
         IF IMAGE(CURSOR-2) EQ 0            /* A BLANK, OR              */
      // IF IMAGE(CURSOR-2) EQ -11          /* A COMMA                  */
      ),
      LITERAL+SCAN=1
   // SPECIAL $                             /* RELATIONAL OPERATORS ETC.*/
<MACROPREPASS> :=
          CYCLE
          (
             CURSOR=1,
             IF CURSOR-CHAR NE -999,                       /* UNTIL END OF INPUT          */
             SCAN,
             (
                IF CURSOR EQ 1,                            /* CHECK LABEL FIELD           */
                <LABEL>,
                LABEL-FIELD=1
             // LABEL-FIELD=0
             ),
             SCAN,
             (
                'PROC',                                    /* MACRO DEFINITION            */
                <LEGAL>,
                IF LABEL-FIELD EQ 1,                       /* MUST HAVE LABEL FIELD       */
                IF BUILD NE 1,                             /* NESTED NOT ALLOWED          */
                MS=POSITION OF <LABEL>,                    /* SAVE MACRO NAME             */
                ML=SIZE OF <LABEL>,
                PS=MS,                                     /* AND PARAMETER NAME          */
                PL=ML,
                ANSWER=STARTMACRO(MS,ML,PS,PL),
                BUILD=1
             // 'END',                                     /* END OF MACRO                */
                <LEGAL>,
                IF BUILD EQ 1,                             /* MUST BE IN A MACRO          */
                IF LABEL-FIELD EQ 0,                       /* MUST NOT HAVE A LABEL       */
                END-MACRO,
                BUILD=0
             // IF BUILD EQ 1,                             /* MACRO BODY                  */
                CURSOR=1,
                STRING=CURSOR,
                START-BODY-LINE,
                <BUILDELEMENTS>                            /* BUILD PIECES OF LINE        */
             // NULL
             ),
             NEXT-IMAGE
          ),
          RESET-INPUT,                                     /* RE-POSITION INPUT           */
          3(NEXT-IMAGE) $
   <ZEROLEVELPAREN> :=
          (  SCAN FOR '$',
             ANSWER=LTSCAN
          // ANSWER=0
          ),
          (
             SCAN FOR ')' '
          // SCAN FOR '),'
          ),
          CURSOR=LTSCAN,
          (  IF ANSWER GT 0,
             IF ANSWER LT LTSCAN,
             LIT-FLAG=1
          // LIT-FLAG=Ø
           ) $
   <LEGAL> :=
           ' '
        // '.' $                                    /* BLANK AND COMMENT           */
   <LABEL> :=                                      /* ARE ONLY LEGAL TOKEN        */
          1 TO 4 ALMERICS,
          IF ' ' $                                  /* MACRO NAME IS FOUR          */
   <BUILDELEMENTS> :=                               /* LETTERS AND/OR DIGITS       */
          SCAN-FOR-PARM,
                                                    /* LOOK AHEAD FOR A SUB-       */
          IF LTSCAN GT Ø,                           /* STITUTION                   */
          CURSOR=LTSCAN,                            /* IF IT IS FOUND              */
          (                                         /* MOVE TO THE PARAMETER       */
             CURSOR=CURSOR+PL,
             ANSWER=Ø,                              /* MOVE PAST PARAMETER         */
             '(',
             <NUMBER>,                              /* DETERMINE PARAMETER         */
             ')',                                   /* NUMBER                      */
             TOKEN-VALUE=TOKEN-VALUE+1,
             ANSWER=1,
             NOT NULL
          // IF ANSWER EQ 1,
             <NONSUB>,                              /* PUT IN NON-SUBSTITUTABLE    */
             ANSWER=SUBPARMNUM(TOKEN-VALUE),        /* PART OF THE MACRO LINE      */
             CURSOR=CURSOR+PL,                      /* PUT IN PARAMETER NUMBER     */
             '(',
             <NUMBER>,
             ')',
             STRING=CURSOR
          // CURSOR=CURSOR+PL                       /* NOT A PARAMETER, IGNORE     */
          ),
          <BUILDELEMENTS>
       // CURSOR=LENGTH,                            /* SCAN OFF REST OF LINE       */
          <NONSUB> $                                /* AND OUTPUT                  */
   <NONSUB> :=
          IF STRING GE CURSOR                       /* NON-SUBSTITUTABLE CODE      */
                                                    /* ONLY PUT OUT IF THERE       */
       // ANSWER=NONSUB(STRING,CURSOR-STRING),
          STRING=CURSOR $
   <FIRSTCARD> :=
            IF FIRST-CARD EQ 1,
            ';',
            SCAN,
            SUBFIELD=2,
            TOKEN-START=CURSOR,
            'ASSEMBLE',
            TOKEN-SIZE=8,
            TOKEN-TYPE=NAME,
            FIRST-CARD=2
   // IF FIRST-CARD EQ 2,
         SCAN,
           <NAME>,
```

C-6

ORIGINAL PAGE IS
OF POOR QUALITY

```
            SUBFIELD=3,
            TOKEN-TYPE=NAME,
            TOKEN-SIZE=SIZE OF <NAME>,
            TOKEN-START=POSITION OF <NAME>,
            LISTING=1,
            MACROS=1,
            Ø TO 2
            (
                    SCAN,
                    (
                            '(NOLIST)', LISTING=Ø
                    //      '(NOPROC)', MACROS=Ø
                    )
            ),
            FIRST-CARD=3
        // SCAN,
            IF CURSOR-CHAR EQ =999,
            CURSOR=CURSOR+1,
            TOKEN-TYPE=END-OF-LINE,
            FIRST-CARD=Ø $
     END OF LEXICAL DEFINITION $
```

```
TOTAL NUMBER OF RECOGNITION ERRORS =    0
```

END
DATE
SEP 14 1979