Table of Contents

Preface III

Glossary of terms V

PART 1 ASSEMBLY LANGUAGE 1-1
Introduction 1-3
Syntax description 1-7

Chapter 1 Format of source statements 1-9
Label field 1-11
Operation field 1-11
Operand field 1-13
Comment field 1-16
Input of source statements and corrections 1-16
Addressing modes 1-17

Chapter 2 Functional operation of instructions 1-19
Load and Store instructions 1-19
Arithmetic instructions 1-19
Logical instructions 1-19
Character handling instructions 1-19
Branch instructions 1-19
Shift instructions 1-21
Control instructions 1-22
I/O instructions 1-22
External transfer instructions 1-23
Move table instructions 1-23

Chapter 3 Assembly directives 1-25
Program framework 1-26
IDENT 1-27
END 1-28

Linkage control 1-29
ENTRY 1-30
EXTRN 1-31
COMN 1-32

Assembly control 1-34
IFT 1-35
IFF 1-35
XIF 1-35
STAB 1-36

AORG 1-37
RORG 1-37

Value definition 1-38
DATA 1-38
EQU 1-40

Area reservation 1-41
RES 1-41

Listing control 1-42
EJECT 1-42
NLIST 1-42
LIST 1-42

Symbol generation. 1-43
FORM 1-43
XFORM. 1-47
GEN 1-48

List of predefined symbols. 1-49

Chapter 4 Programming considerations 1-51

Stand Alone or Monitor controlled programming. 1-51
Interrupt system 1-51
System stack 1-51
User stack 1-52
Trap action 1-54
Simulation routine 1-54
Adaptation of P855M software to P800M software 1-54
Use of RTN instruction 1-55
Stand Alone Input and Output Programming 1-55

Absolute addressing	addressing specific locations in memory (see also relocatable addressing)
Assembler	a system program which translates programs written in Assembly Language into binary object code
Bootstrap	a program provided for initial loading of the system
Breakpoint	address at which execution of program stops to allow further debugging
Character	eight bits, representing an integer, letter or other data item
Cluster	a set of data in object code
Common	
blank common	an area to which external references can be made from one or more modules
labeled common	a predefined external reference which can be used in several modules
Debugging Package	a processor which allows the programmer to insert breakpoints in a load module and call debugging functions before execution of a program
Directive	an instruction used for providing a framework for a program or for guiding the assembly process
Effective memory address	address in memory where the actual information can be found
Entry point	a label to which an external reference is made
External reference	a reference to an entry point in another program or module
File code	one or two hexadecimal digits associated with an I/O device

Identifier	a character or a combination of characters used to label an instruction or a value which is to be referred to by other instructions
Internal symbol	identifier in a module
IPL	Initial Program Loader. A program to load the monitor
Label	identifier of max. six characters long, the first always being a letter
Linkage Editor	a processor used to link independent object modules before execution
Load Module	program output by the Linkage Editor containing no external references
Location counter	counter used to assign a relative or absolute address to program elements
Mnemonic	abbreviation for an instruction, as used in the operation code field of a source statement, to indicate a machine instruction or directive
Module	a part of a program, enclosed by an IDENT and END directive, which can be treated independently of the rest of the program
Monitor	a system program which supervises the loading, processing and execution of user programs, starts and supervises the operation of processors and initialises I/O operations
Object code	program as translated by a language translator and suitable to be input to the Linkage Editor
Operand	an expression indicating the address, value or register to be operated upon by the machine instruction
Pass	one program run
Real Time Clock	a mechanism by means of which the amount of computer time allocated to a program is measured and a signal is given when that period of time has ended

Relocatable addressing	addressing in relation to the beginning of a program, not to specific locations in memory. The relocation of the addresses is then done by the machine
Source statement	one line in a source program
Stand Alone processor	processor not running under Monitor control. It contains its own I/O routines
Symbol	an identifier, used as an address value in the operand field of other instructions
Update Package	a processor which handles the additions and deletions in source or object programs

PART 1 ASSEMBLY LANGUAGE

This part contains a description of the Assembly Language. In this description it is made clear how the programmer can write his programs using the instructions of the P800M Instruction Set as well as the directives which guide the assembly process when the program is input to the Assembler. The instruction sets of the P800M series computers are upward compatible.

Programs for the P800M computers are written in a symbolic language closely related to the machine code. Each statement (or line) of the program relates to a single machine instruction or to a data item to be taken into account by an instruction.

To write programs in the Assembly Language, the user should be familiar with the syntax of the instructions, which are devided in the following main groups:

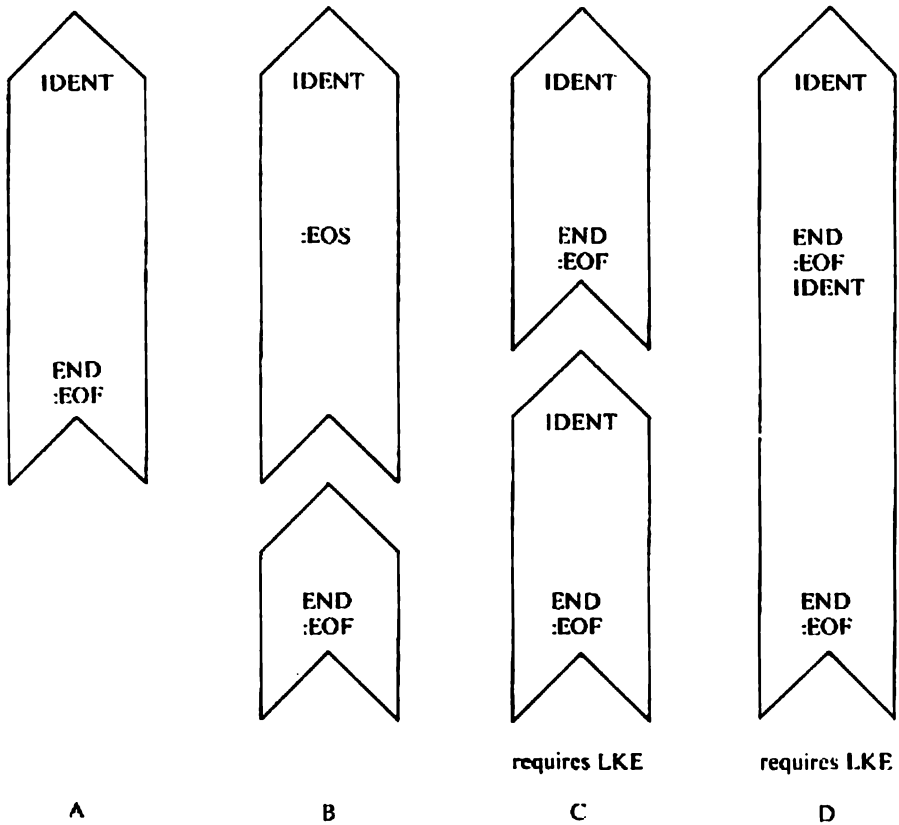
- Load and Store instructions
- Arithmetic instructions
- Logical instructions
- Character handling instructions
- Branch instructions
- Shift instructions
- Control instructions
- Input/Output instructions
- External Transfer instructions
- Move Table instructions.

Programming in Assembly Language requires certain rules to be acceptable to the Assembler.
A source program may consist of one or more modules each of which starts with an identification IDENT and terminates with an END (see directives). The whole source program must be terminated by an "End Of File" mark (:EOF).

NOTE: If a source program consists of several modules the modules need not be separated by :EOF marks but by :EOS marks (End Of Segment). An :EOS mark at the end of a punched tape indicates the physical end of that tape when the program is punched on two tapes. The mark is not part of the Assembly Language.

The following figure shows various possibilities of how programs can be punched on tape.

In example A the program is contained on one punched tape. The program starts with an identification IDENT and is terminated by END which will cause an :EOS to be punched when the program is assembled, and is followed by an :EOF.



Example B is an example of a program punched on two tapes. The first tape starts with an IDENT and is terminated with :EOS which causes the Assembler to wait for operator action. The second tape does not contain an IDENT and is read immediately after the first one. The second tape is terminated by an END and :EOF.

Example C consists of two modules on two tapes both beginning with IDENT and ending with END and :EOF.

Example D consists of several modules punched on one tape. Each module begins with an IDENT and is terminated by END and either an :EOS mark if another module follows this one or by :EOF when it is the last module to be processed. This example requires the Linkage Editor to make of those modules one larger program which can be executed.
Each module of a program consists of a number of characters grouped into lines and each statement in a module is made up of the following characters:

Letters: A to Z inclusive

Digits: 0 to 9 inclusive

Delimiters:

+	plus
—	minus
*	asterisk
=	equal
'	apostrophe
,	comma
	blank
/	slash
(	left parenthesis
)	right parenthesis
.	period
:	colon

Location counter
The Assembler maintains a location counter which is a software counter used to assign a relative or absolute memory address to program elements. The location counter starts with a relative value equal to zero, or it starts at an absolute address defined by the AORG directive, at the beginning of an assembly. The value of the counter is incremented by 2 or a multiple of 2 depending on the kind of instruction given.
The current value of the location counter is referred to by an * in the operand field (see below). In absolute program sections * has an absolute value. In that case the value is incremented in the normal way and the value may be changed by a RES or RORG directive.
The location counter may take neither a negative relative value nor an odd value.

Symbols
A symbol is a character or a string of characters used to represent addresses or values. Symbols may appear in the label field as well as in the operand field of a statement.
Their syntax is the same as for the label (see under label field). Some symbols are predefined and have a special meaning for the Assembler e.g. * indicates the current value of the location counter, P is the instruction counter etc.

The following symbols are used to define the syntax of the P800M Assembly Language.

< > to enclose syntactic items
| the vertical stroke has the meaning of or
::= is composed of
[ ] the syntactic items between these brackets may be omitted
[ ] select one of the items between these brackets
␣ space

The following list contains the definition of all items used.

<statement>	::= [<label>]␣<operation code>[<operand>][<comments>] [*<comments>]
<label>	::= <identifier>
<operation code>	::= <mnemonic>[S|(<cnd>)|L] [*]<directive>
<operand>	::= [+|−]<term>[+|−]<term> [+|−]<term>
<comments>	::= <characters>|*<characters>
<identifier>	::= <letter>|<identifier><letter>| <identifier><digit>|<identifier>
<mnemonic>	::= <letters representing operation code>
<S>	::= <store indicator>
(<cnd>)	::= <numerical condition value>| <condition mnemonic>
<numerical condition value>	::= 0|1|2|3|4|5|6|7
<condition mnemonic>	::= Z|P|N|O|E|G|L|A|U|NA| NR|NZ|NP|NE|NG|NL|NN
<L>	::= < load indicator>
*	::= indirection
<directive>	::= <IDENT, END etc.> see chapter on directives
<DATA defined hexa constant>	::= <see DATA directive>
<module name>	::= <symbol>
<symbol>	::= <characters representing address or value>

```
<predefined expression>      :: = <max. of two defined symbols>
<entry point name>           :: = <identifier within reference module>
<external>                   :: = <identifier defined in other module>
<common-field
  definition list>           :: = <commen field definition>,<common field
                                  definition> .....
<common field definition>    :: = <common field name>[<common field
                                  length>]
<common field name>          :: = <identifier>
<common field length>        :: = predefined (absolute) expression
<internal symbol list>       :: = <internal symbol>,<internal symbol>
                                  .....
<internal symbol>            :: = <identifier>
<field definition>           :: = <field length definition>[[=|:]<field value
                                  definition>
<field length definition>    :: = <number of bits>
< = field value definition>  :: = <value to be placed in field>
<:field value definition>    :: = <address of word>
<field number>               :: = <decimal integer>
<term>                       :: = <constant>|<symbol>
<constant>                   :: = <decimal constant>|<hexadecimal
                                  constant>|<character constant>
<decimal constant>           :: = <digit>|<integer>
<hexadecimal constant>       :: = <hexadecimal integer>
<character constant>         :: = <letter>|<digit>|<delimeter>
<letter>                     :: = A|B|C|D|E|F|G|H|I|J|K|L|M|
                                  N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>                      :: = 0|1|2|3|4|5|6|7|8|9
<delimiter>                  :: = +|-|*|=|'|.|␣|/|(|)|,|:
<integer>                    :: = <number>
```

A source module consists of a sequence of statements. The Assembler interprets each line as it is presented.
Statements can be divided in the following fields:

- label field
- operation field
- operand field
- comments field

< statement > :: =[< label > |␣ < operation code > ␣[< operand >]␣
[< comments > |
*[< comments >]

Each field has to be separated from the other by one (or more) blank character(s). Blanks may not appear in the fields themselves except when specified in a character constant or in a comments field. Instead of blanks a backslash may be used for separation (see page 1-16). One or more blanks at the beginning of a statement indicate that there is no label field.

If there are more than ten blanks after the operation field all following characters are considered to be belonging to the comments field.
An * (asterisk) at the beginning of a statement identifies that line as a comments line.

Statements punched on tape which are to be read by the ASR punched tape reader have to be terminated by LF XOFF CR, which switches the reader off, followed by a Null character, e.g. Rub-out, to allow for a proper reading and processing of the next usable character.

PHILIPS

data systems

DATA _____ PAGE ___ OF ___

PROGRAM EXAMPLE OF programming paper PROGRAMMER _____

label	operation	operand	comments	identification
MOVBUF	EQU	0		
	LDXL	A1&0,BUFFSIZ	BUFF ADDRESS	
	LDXL	A1&2,0		
MOVLOP	LDXL	A1&2,BUFFSIZ		
	ADR	A1&4,A1&4		
	LDR	A0,A1&2		
	LDXL	A1&2,MOVE		

LABEL FIELD

< label > :: = < identifier >
< identifier > :: =
< letter > | < identifier > < letter > | < identifier > < digit > | < identifier >

Labels (or identifiers) in a module are used for reference purpose to other statements in a module.
The Assembler assigns, in most cases, to each label a word address value which is the numerical equivalent (absolute or relocatable) of the label.

The maximum number of characters in a label recognised by the Assembler is six. The first of those must always be a letter. A label, however, may contain more than six characters but the additional characters will not be taken into account. If the label has already been allocated to another statement an error message is output.
Period signs in a label are not significant, e.g.

L.A.B.E.L. has the same meaning as LABEL.

The value of a label is normally regarded as relocatable, except when:

- an absolute address is equated by an EQU directive
- the label appears in an absolute program section (defined by the AORG directive and which is not equated by an EQU directive to a label previously defined as relocatable).

OPERATION FIELD

< operation code > :: = < mnemonic > [S|(< end >)|L][=] < assembly directive >

where:

< mnemonic >

The operation field normally contains the mnemonic of a standard instruction. It is possible, however, to generate one's own instruction mnemonic by means of the FORM, XFORM and GEN directives (only with the monitor controlled Assembler).

S

Allowed after the mnemonic of certain register to register and memory reference instructions. It indicates that the result of the operation must be stored in a memory word (bit 15 of the instruction is set to 1). In fact, S has to be considered as a part of the instruction mnemonic.

e.g. CIR and CIRS instructions are to be considered as two different instructions.

NOTE: It is allowed to have the S preceded by a period sign though the Assembler does not take this sign into account.
e.g. AD.S␣ = ADS␣

(< **cnd** >):: = < numerical condition value > | < condition mnemonic >
< numerical condition value > :: = 0/1/ . . . . . /7
< condition mnemonic > :: =
Z|P|N|O|E|G|L|A|R|U|NA|NR|NZ|NP|NE|NG|NL|NN

This indicator specifies the condition under which a conditional branch instruction is to be performed. The table below shows how in the Assembler the conditional mnemonics and numerical condition values may be used.

COND. REG CONTENTS	GENERAL	ARITHM.
0	(0)	(Z) ZERO
1	(1)	(P) POS.
2	(2)	(N) NEG.
3	(3)	(O) OVERFL.
		NOT - CONDITION
≠ 0	(4)	(NZ) NOT ZERO
≠ 1	(5)	(NP) NOT POS.
≠ 2	(6)	(NN) NOT NEG.
≠ 3	(7)	UNCONDITIONAL

.L

Allowed after the instruction mnemonic of a constant instruction. It specifies that the operand is contained in 16 bits i.e. that the instruction must be assembled as a "long" instruction.

•
Indicates the indirect addressing mode in a register to register or a memory reference instruction.

OPERAND FIELD

The operand field may contain an address expression, a register expression or constants associated with the current machine instruction or assembly directive or a combination of those.
The structure and meaning of the operand depends on the type of instruction and directive and is explained below.
All operand expressions must be separated by a comma.

Expression

< expression > :: = [+ | −]< term >[[+ | −]< term >[[+ | −]< term >]]
< term > :: = < constant > | < symbol >

NOTE: • is considered to be a symbol.

An expression may not refer to more than 2 symbols and may not refer to more than one register name. In the latter case it may not contain any other term.

(<CND>)

COMPARE	I/O
(E) EQUAL (G) GREATER (L) LESS —	(A) ACCEPTED (R) REFUSED — (U) UNKNOWN
(NE) NOT EQUAL (NG) NOT GREATER (NL) NOT LESS	(NA) NOT ACCEPTED (NR) NOT REFUSED —

Address expression
The address specified in a memory reference instruction can be either absolute or relocatable.
An *absolute address* is the actual address in memory where the information the user needs can be found.
A *relocatable address* is relative to the beginning of the program in which it appears.

The address expression may contain any of the following terms or a combination of them:

*	asterisk, which is a predefined expression representing the current value of the location counter. This counter is incremented by two or a multiple of two depending on the length of the instruction.
symbol	used to refer to an instruction or data word with the same identifier in its label field. The Assembler will convert the symbol to a relative address.
displacement value	which can be attached to * or <symbol> to indicate a word not labeled by an identifier.

Predefined expression
A predefined expression is an expression consisting of not more than two symbols, each of which is defined i.e. has been assigned a value. Some symbols are implicitely predefined in the Assembler (see page 1-49).

An expression may contain only one external reference. The remainder, if any, of such an expression must have a predefined absolute value. The combination of an external reference and a predefined absolute value may only be used for specifying the value of a 16-bit field. The table below shows the result of a combination of positive and negative absolute or relocatable values:

2nd term \ 1st term	+ R	− R	+ A	− A
+ R	E	A	R	R
− R	A	E	E	E
+ A	R	E	A	A
− A	R	E	A	A

where:

R = relocatable
A = absolute
E = erroneous

Register expression
Register expressions are regarded as predefined expressions and consist of one or two characters. The register expressions recognised by the Assembler are:

P	P-register or instruction counter
A1 . . . A14	Registers 1 to 14 (general purpose registers)
A15	Register 15 (stackpointer)

Constants
A variety of constant types may be specified in the operand of an instruction or directive.

< constant > :: = < decimal constant > | < hexadecimal constant > | < character constant >

Decimal constants
< decimal constant > :: = < digit > | < integer >

The decimal constant is a digit or integer contained in an 8-bit character or 16-bit word whose value may range from 0 to 32767.

Hexadecimal constants
< hexadecimal constant > :: = / < hexa integer > | X' < hexa integer > '

The hexadecimal constant is considered to be hexadecimal value or bit string in the range from 0 to /FFFF.

Character constants
< character constant > :: = '< character > [< character >]'

A character constant is composed of a character string enclosed in single quotation marks. The string is composed of the characters described in the character set on page 1-5.

A character constant can be used with a machine instruction only if the constant consists of either one character (short constant) or two characters (long constant). Longer strings can be specified in a DATA directive. A single quote mark (') used as a character is specified by two consecutive single quote marks.

COMMENT FIELD

Comments are only for the programmer's benefit. They are included in the assembly listing but not in the generated object program.
A line is considered to be a comment line when the first 10 characters of that line are blanks or when the line starts with an asterisk.

INPUT OF SOURCE STATEMENTS AND CORRECTIONS

The user may type in the statements and corrections from the operator's typewriter. He may do so by counting the number of characters to obtain a neat output on the listing device.

Example:

1st col		10th col		19th col		40th col
label	␣	opcode	␣	operand	␣	comments

may be typed as follows:

label\opcode\operand\comments

without having to count for the first column of each field.

Example:

```
DATAF\LDK\A4,4
\ABL(7)\HALT
DEVUN\LDK\A4,5
\ABL(7)\HALT
ADDIT\LDK\A1,0\SET INDEX REGISTER FOR BUFFER.
\LDK\A3,00FF\LOGICAL CONSTANT INTO A3
```

ADDRESSING MODES

In Volume II we see how addressing takes place from a hardware point of view. The condition an instruction must fullfil to meet the requirements of the Assembler is explained on the preceding pages. Specific examples, with source statements and explanation concerning the arithmetic instructions AD and ADR are given to show the operation within the CPU.
See for the hardware operation of those instructions Volume II. The order in which the examples are given is in accordance with the description on those pages.

Direct addressing

AD A1,LABEL	The contents of the memory location with symbolic address LABEL are added to the contents of register A1. The result is placed in A1.
ADS A1,LABEL	The contents of the memory location with address LABEL are added to the contents of register A1. The result is stored in LABEL.

Indexed addressing

AD A2,LABEL,A10	The contents of register A10 are added to the address LABEL. The result gives an address whose contents are added to the contents of A2. The result of the latter operation is placed in A2.
ADS A2,LABEL,A10	The contents of register A10 are added to the address LABEL. The result gives an address whose contents are added to the contents of A2. The result of the latter operation is stored in the address: LABEL + contents of A10.

Indirect addressing

AD* A2,LABEL	The contents of LABEL point to an address whose contents are added to the contents of register A2. The result is placed in A2.
ADS* A2,LABEL	The contents of LABEL point to an address whose contents are added to the contents of register A2. The result is placed in the contents of LABEL.

Indexed Indirect addressing

AD* A2,LABEL,A10	LABEL is added to the contents of register A10. The result points to an address whose contents are added to the contents of register A2. The result hereof is placed in register A2.
ADS* A2,LABEL,A10	LABEL is added to the contents of register A10. The result points to an address whose contents are added to the contents of register A2. The result hereof is placed in the address obtained of A10.

Register to Register operation

ADR A1,A2	The contents of A2 are added to the contents of A1. The result is placed in A1.

Register addressing

ADR* A1,A2	The contents of the address pointed to by A2 are added to the contents of register A1. The result is placed in A1.
ADRS A1,A2	The contents of the address pointed to by A2 are added to the contents of A1. The result is stored in the address pointed to by A2.

LOAD AND STORE INSTRUCTIONS

Load Instructions
Before the programmer can perform an operation on the contents of a memory location or a register its contents must be placed in one of the registers A1 thru A15.
Two load instructions are provided, allowing to load a 16-bit word from anywhere in memory or from any register into a specified register where the operation will take place, and one instruction to load a constant into a register.

Store instructions
Companion to the load instruction is the store instruction which may store the contents of a register, containing the result of an operation, into a memory location or a register.

ARITHMETIC INSTRUCTIONS

Arithmetic instructions perform the normal arithmetic functions such as add, subtract. The instruction operand operates upon the contents of the specified register.

LOGICAL INSTRUCTIONS

Instructions described under this heading are called logical instructions because they operate on binary information according to the rules of logic. The first operand which may be a memory location, a register (R1 or R3) or a constant is compared with the second operand, register R2. The result is placed in a register or possibly in memory. In the instruction set each logical instruction is given a description in which way the contents of a memory location is ANDed or ORed.

CHARACTER HANDLING INSTRUCTIONS

Character handling instructions operate on a character level. Characters may be exchanged, compared or 8 bits of a constant may be placed in 8 bits of a register.

BRANCH INSTRUCTIONS

These instructions cause a branch to an address in memory either when a certain condition is fulfilled or unconditionally.

In branch instructions on condition the instruction mnemonic is followed by a number ranging from 1 thru 6, enclosed in brackets. When the number is (7) or omitted, the branch is unconditionally.
These numbers are compared with the contents of the condition register set by the previous instruction.
The condition number has the following meanings:

(0) branch if CR	= 0	(4) branch if CR	≠ 0
(1)	= 1	(5)	≠ 1
(2)	= 2	(6)	≠ 2
(3)	= 3	(7) unconditional branch	

Example:

```
          —
          —
          LDK     A2,4
LABEL.    SUK     A2,1
          RB(4)   LABEL.
          —
          —
```

The Assembler allows to use, instead of a number, a condition mnemonic e.g. Z, E, A (see page 1-12).
Unconditional branches are made by the following instructions:
— absolute branch instruction or relative branch instruction without a condition indicator or when (7) is specified.
— CF, RTN, EX instructions.

Long format absolute branch instructions permit to branch, forward as well as backwards, to any address in the program. Short format absolute branch instructions may only branch to locations 0000 to 00FE. Relative forward and relative backward instructions may not skip backwards more than 127 locations and 128 locations forward.
The Assembler gives an error indication if the permissible branch range is exceeded.
The address to which control is to pass may be indicated in various ways:

1. By means of a symbolic address expression:
   ABL(3) LABEL

2. By an absolute address held in a register:
   ABR (7) A5

3. By using a constant to indicate an absolute memory address (short constant):
   AB /84

4. By means of a displacement value added to or subtracted from the instruction counter value (RB and RF instructions only). This displacement value is computed by the Assembler from an address expression used in the operand and may not exceed more than 128 words forward or 127 backwards:
RB(0) ZERO

Another group of branch instructions are the Call Function and Return from Function instructions. The Call Function instruction provides a link to a subroutine by branching to the first instruction of the subroutine. To be able to resume the execution of the main program after the subroutine has been executed the contents of the P-register and the Program Status Word are stored in the stack. When the last instruction of the subroutine (RTN) is executed the contents of P and PSW are restored.
A special group within the branch instructions is formed by the instructions EX, EXK and EXR.
These instructions allow to address a memory location of which the contents is the binary representation of another instruction. The latter instruction is executed before the program continues with the next instruction in sequence.

Example:

```
        —
        LDKL    A3,CIO
        LDKL    A4,SST
        —
        —
CIO     CIO     A1,1,TY
        EXR*    A4          EXECUTE SST
        RB(4)   *—2
        —
        EXR*    A3          EXECUTE CIO
        —
SST     SST     A7,TY
        RB(4)   *—2
```

The Execute instruction may not refer to other EX, EXK or EXR instructions or to Call Function, RTN or double format instructions.

SHIFT INSTRUCTIONS

Shift instructions operate on a bit level. These instructions allow to rotate the contents of one of the registers A1 thru A7 n positions in the direction and manner specified in the instruction.

CONTROL INSTRUCTIONS

These instructions perform the control of the program by allowing the program to be interrupted or not, or to reset an internal interrupt. Except for the LKM instruction, control instructions should only be used in Stand Alone programming.

INH and ENB are two companion instructions. The program part between these instructions is not interrupted as INH inhibits all interrupts. ENB sets the machine status to permit interrupts.

Example:

```
        IDENT   TEST
OUT     EQU     *
        RORG    OUT + /600
START   HLT
        INH
        LDK     A5,0
        LDKL    A11,BUF
        LDK     A2,0
AGAIN   CIO     A2,1,/30        } program inhibited
        RB(NA)  AGAIN
        LC      A3,BUFPTP,A5
        —
        —
        ENB
```

The RIT instruction is used to reset an internal interrupt which was previously set by an interrupt from the control panel, power failure/automatic restart, real-time clock or by a program error.
The programmer may specify a 5-bit hexadecimal value in the operand of this instruction to clear specific interrupts.

```
INTRTC  RIT     /1B     Reset the real-time clock interrupt
```

I/O INSTRUCTIONS

I/O instructions handle the data transfer between the CPU and peripherals, the operation of control units for these peripherals and status control.
In monitor controlled programs the I/O functions, initiated by these instructions, are taken over by a general I/O routine which is called each time a LKM instruction followed by a DATA directive is used.

The user need therefore not to write his own I/O routines. When the programmer is to write a Stand Alone program he has to write his own I/O routines.

EXTERNAL TRANSFER INSTRUCTIONS

Two of these instructions, WER and RER, may be used for programming the I/O Processor by addressing an external register. The function of these instructions is described on page 1-54. The other instructions of this group are only useful when working with the Memory Management Unit (MMU) on the P857M and permit to load 16 registers on the MMU with 16 consecutive memory locations, or to replace these locations with the contents of the 16 registers. The 16 registers are called *segment table*.

MOVE TABLE INSTRUCTIONS

These instructions can only be used on the P857M. They allow to move a table either to an area higher or lower in memory or to move a table from a user to a system area, and vice versa.

Directives are used to provide a framework for a program and to guide the assembly process. The directives are written in the program and are printed on the assembly listing if the listing option is specified in the Assembler option message (see page 2-5).
The two versions of the Assembler accept either all directives (monitor controlled Assembler) or part of the directives (Stand Alone Assembler).
The table below gives a survey of which directives are accepted by which Assembler.

Directive	Meaning	Stand Alone Assembler	Monitor controlled Assembler	page
IDENT	Program identification	X	X	1-27
END	End of assembly	X	X	1-28
ENTRY	Define entry point name	X	X	1-30
EXTRN	Define external references	X	X	1-31
COMN	Define common blocks	–	X	1-32
STAB	Define internal symbol table	–	X	1-36
AORG	Assign absolute origin	X	X	1-37
RORG	Assign relative origin	X	X	1-37
IFF	If false	–	X	1-35
IFT	If true	–	X	1-35
XIF	End of condition	–	X	1-35
DATA	Data generation	X	X	1-38
EQU	Equate symbol to value	X	X	1-40
RES	Reserve memory area	X	X	1-41
EJECT	Continue listing on new page	–	X	1-42
LIST	Resume listing output	–	X	1-42
NLIST	Suspend listing output	–	X	1-42
FORM	Format definition	–	X	1-43
XFORM	Extension of FORM directive	–	X	1-47
GEN	Generation directive	–	X	1-48

The directives can be divided in the following groups according to their function:

- Program framework : IDENT, END
- Linkage control : ENTRY, EXTRN, COMN
- Assembly control : IFT, IFF, XIF, STAB, AORG, RORG
- Value definition : EQU, DATA
- Area reservation : RES
- Listing control : NLIST, LIST, EJECT
- Symbol generation : FORM, XFORM, GEN

PROGRAM FRAMEWORK

The directives IDENT and END form respectively the first and last statements in the module. They are mandatory. The module punched on tape must be followed by :EOS or :EOF.
The IDENT directive is used for identification purposes and the END directive generates the END cluster after which the assembly process is stopped and a symbol table is printed.

The IDENT directive specifies the name to be given to the object module output by the Assembler. It is used for identification purposes in selective loading or updating (see parts on Linkage Editor and Update Package).
This directive must always be present and must be the first statement in a source module.

Syntax

␣IDENT␣ < module name >

where:

< module name >	A symbol which is specified according to the rules for a label.

END of assembly

END

This directive must be the last statement in a module and terminates the assembly process by punching an :EOS mark.

Syntax

[< label >]␣END␣[< predefined expression >][, < symbol >]

where:

< label >	The tabel is given a relative value equal to the length of the relative section of the generated object program. This length includes the length of the optional symbol table (see STAB directive, page 1-36). The value is 0 if this module is absolute.
< predefined expression >	This expression, if present, gives the address of the first instruction to be performed in the program after loading.
< symbol >	This parameter gives an entry point name to the internal symbol table of the generated object program when the STAB directive has been assembled.

LINKAGE CONTROL

Some modules which have to be grouped into one larger program contain references to identifiers defined in other modules.
By means of the directives ENTRY and EXTRN the user is able to refer to certain parts in other modules whereas the directive COMN allows to transfer data among several modules either written in Assembly Language or in FORTRAN.
By using a COMN the programmer can define one or more common blocks. Each common block may be divided in a number of subfields of varying length, each having a symbolic name which can be referred to directly but only in the module in which they are declared.
COMN blocks may be labeled or blank; a COMN block is labeled if a name is attached to it.
The Linkage Editor allocates a space to the blank common block at the end of the link-load or link-edit run (see Linkage Editor). This block is placed at the end of the entire program.
Labeled commons are placed at the end of the first module that refers to it.

The ENTRY, EXTRN and COMN directives must always follow immediately after the IDENT directive and in this order, though it is not necessary that the ENTRY as well as EXTRN and COMN are specified.

So: IDENT, ENTRY, EXTRN, COMN or
IDENT, EXTRN, COMN or
IDENT, ENTRY, COMN etc.

The ENTRY directive is used to declare entry points, i. e. labels which are defined in the current module and used as operands of another module.
The directive must follow, if present, the directive IDENT.

Syntax

␣ENTRY␣<entry point name>[,<entry point name>, ...,<entry point name>]

where:

<entry point name> Can be referred to by an operand of an instruction in another module. The maximum number of entry points which can be specified in one ENTRY directive is determined by the length of one line.

Example (see also EXTRN)

```
          IDENT   PROG
          ENTRY   NUMB1, NUMB2, NUMB 3
          —
          —
          —
NUMB1     LDKL    A3, LABEL
          —
          —
NUMB2     ST      A6, REFER
          —
          —
NUMB3     CF      A14, EOS
          —
          —
          —
          END     START
```

The EXTRN directive is used to declare externals i.e. operands which are used in the current module and defined as labels in another module.
The directive must follow ENTRY, or IDENT when the directive ENTRY is not present.

Syntax

␣EXTRN␣ <external name>[,<external name> ...,<external name>]

where:

<external name> Name of external reference (label in other module). The maximum number of external names which can be specified in one EXTRN directive is determined by the length of one line.

Example (see also ENTRY)

```
        IDENT   ASMPRO
        EXTRN   NUMB2
        -
        -
        -
        CF      A14, NUMB2
        -
        -
        -
        END     START
```

The COMN directive facilitates communication between modules written in Assembly Language or FORTRAN. The directive is written as follows:

Syntax

[< label >]␣COMN␣ < common field definition list >

where:

< common field definition list > ::= < common field definition > [, < common field definition list >]

where:

< common field definition > ::= < common field name > [< common field length >]

where:

< common field name > ::= < identifier >
< common field length > ::= < predefined absolute expression >

If the parameter < common field length > is omitted the default value assumed by the Assembler is 1. The field length must be given in words.

Example
A␣COMN␣FVAL1 (3), FVAL2 (3), INTGV (10)

which defines a labeled common, named A, having the length

3 + 3 + 10 = 16 words.

A is defined as an external reference and common block name. Either the common block name itself or the subfield names may be referred to in the same module. The subfield names are then considered to be equivalent to:

< common block name > + < absolute displacement >

so.

LD␣A1, FVAL2 is equivalent to LD␣A1, A + 6

and

ST␣A2, INTGV + 18 is equivalent to ST␣A2, A + 30

The displacements in this example are counted in characters.
Blank commons can only be referred to by the subfield names defined in the operand field.

␣COMN␣VAL1 (3), VAL2 (4)
␣COMN␣VAL3 (9), VAL4 (10)

These directives define a blank common of 3 + 4 + 9 + 10 = 26 words.

VAL2, for instance, may be used in symbolic expressions and is equivalent to:

<blank common "name"> + 6

ASSEMBLY CONTROL

When it is neccessary to check whether a certain condition is satisfied before assembling a number of source lines. the user may include the directives IFT, IFF and XIF. The assembly of the IDENT – END – XIF directives are never bypassed by IFT or IFF.

By means of the STAB directive the user may specify one or more internal symbols which are to be used for Debugging purposes. All these symbols must have been defined previously in the current module.
Common block names are handled as externals.

The RORG and AORG directives are used to reset the location counter to a relocatable or absolute value indicated in the operands of those two directives.

Those directives are only used in combination with the directive XIF to indicate that a block of instructions is to be assembled only if a certain condition is fulfilled. The assembly of the IDENT – END – XIF directives are never bypassed.

IFT (IF True)

The IFT directive specifies that the Assembler has to assemble the next source lines only if the condition stated by this directive is fulfilled.

Syntax
␣IFT␣ < predefined absolute expression > = < predefined absolute expression >

If the first parameter ≠ second parameter the source line(s) following IFT up to the next XIF directive are not assembled.

IFF (IF False)

Syntax
␣IFF␣ < predefined absolute expression > = < predefined absolute expression >

If the first parameter = the second parameter the source lines following IFF will not be assembled.

Syntax
␣XIF␣

This directive allows all subsequent statements to be assembled until a new IFT or IFF statement is encountered.

The STAB directive outputs at the end of the relocatable program section of the generated module one or several internal symbols to be used for debugging purposes (internal symbol is the address given to a symbol in the program after assembly). All symbols must have been declared previously in the current module. STAB must immediately precede the END directive.

Syntax

␣STAB␣ < internal symbol list >

where:

< internal symbol list > :: = < internal symbol > [, < internal symbol list >]

If the STAB directive does not contain a parameter in the operand field all internal symbols of the module will be included.
The programmer may not specify entry points, external reference names or commons. This directive is only taken into account when in the END directive the parameter < symbol > is specified which gives the name of the internal symbol table.

AORG — Assign absolute ORiGin — AORG

This directives assigns an even absolute value to the location counter. The location counter receives that value by specifying < predefined absolute expression >.
From the time AORG is given and until a RORG directive is given the location counter is incremented in the same way as if it were relative, i.e. by increments of 2 and 4 depending on the length of the instruction. All labels in an absolute module are given an absolute value unless they are equated to a predefined relative value by an EQU directive.
RB and RF instructions in an absolute program cannot refer to an address in a relocatable program section as the place from where this section will be loaded is not known.

Syntax

␣AORG␣ < predefined absolute expression >

RORG — assign Relative ORiGin — RORG

The RORG directive allows the user to specify the beginning of a relocatable module by assigning a relative value, which must always be even, to the location counter. Its value may never become negative. If RORG has no operand the location counter is given the last relocatable value it has previously received. This value is equal to the length of the relocatable module at the time this directive is assembled.

Syntax

␣RORG␣ [< predefined relocatable expression >]

VALUE DEFINITION

The directives DATA and EQU are used to define certain values in a module.

DATA **DATA generation** **DATA**

The DATA directive is used to assign a value to one or more words in the module, for inclusion in the object module.

Syntax

[< label >]␣DATA␣ < data expression >

where:

< data expression > :: = [< expression > | ' < character string > ']
[< data expression >]

< label > refers to a symbol in the operand field elsewhere in the module.

< data expression > the data expression may be:
- a decimal or hexadecimal constant
- an address expression
- a character string consisting of one to thirty-two ASCII characters enclosed by single quote marks. A series of words is generated, of two characters each, which are left justified. When the number of characters is odd the rightmost character of the last word is a space.

Example
The expression may contain a number of parameters which, in total, may generate no more than 16 words in memory.

␣DATA␣'ABC',/0A0D, 1,/A, 2,'DEF

will generate the following words:

4142	'AB
4320	C'␣
0A0D	/0A0D
0001	1
000A	/A
0002	2
4445	'DE
4620	F␣

Example
When the user wishes to make an ECB he may do so as follows:

ECB␣DATA␣1, BUF2, 6, 0, 0, 0,

Example
DATA␣−0128, + 12,/3AB,−/A, LABEL, 'TEXT:'

will generate the following:

FF80	-128
000C	+ 12
03AB	/3AB
FFF6	−/A
< value >	LABEL
5445	'TE
5954	XT'
3A20	:␣

Identifiers are normally defined by being assigned memory values as they appear in the label field of an instruction. The EQU directive may be used to define an indentifier in a direct manner by assigning to it the value of an expression in the operand field. The symbol in the label field is made equivalent to the value in that operand field. This value may be absolute or relocatable.
A symbol, provided it differs from standard mnemonics and FORM-defined mnemonics, may be used as an operation mnemonic but may not be followed by an operand. The Assembler generates one code word each time this mnemonic appears in the operand field.

Syntax

< label > ␣EQU␣ < predefined expression >

Example

```
CT␣     EQU␣/41C4
        —
        —
        CT
        LDKL␣A1, CT
```

CT may now be used anywhere in the program to represent the value /41C4.

Example

```
VAL␣    EQU␣10
        —
        —
        —
        LDK ␣ A1, VAL
```

Example

```
LAB ␣ EQU ␣ *
```

LAB receives the vaue of the location counter. (equal to: LAB RES 0)

Example:

```
C: 1␣EQU␣25
REG: 3␣EQU␣A3
```

Each time the Assembler encounters C:1 or REG:3 they are replaced by "25" and A3 respectively.
LDK A1, C:1→ = LDK A1,25
LDK REG:3,1→ = LDK A3,1
LDK REG:3, C:1→ = LDK A3,25

AREA RESERVATION

The directive RES can ce used to skip over an area in memory. The RES directive saves a memory area of a given length, specified in the operand, advancing the location counter by twice the number of words specified.

RES **REServe memory area** **RES**

The RES directive is used to reserve a number of memory words. The programmer may specify this number in the parameter. The location counter is incremented or decremented depending on the positive or negative value of that parameter. If positive, a memory area of the specified value is reserved. If negative, a memory area of the specified size before the place identified by < label >.
The value of the latter is not changed but the location counter is reset to a lower value by subtracting twice the value specified.

[< label >]␣RES␣ < predefined absolute expression >

where:

< label >	receives the address of the first word of the reserved area.
< predefined absolute expression >	specifies the length of the area to be reserved.

If < predefined absolute expression > is 0 the location counter is not updated and, if < label > is specified, the statement is equivalent to

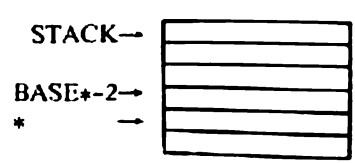
< label > ␣EQU␣*

Examples:

```
        RES␣4      Reserve 4 words
LAB1␣   RES␣-2     Reserve 2 words before LAB1
INS␣    RES␣0      INS receives the value of the location counter.
```

Examples of stack reservation:

```
STACK  RES 4
BASE   EQU *-2
```



LISTING CONTROL

The Assembler normally produces an output listing for each assembly. By means of the directives EJECT, NLIST and LIST the programmer may determine which parts of the modules do not need to be listed.

EJECT **Continue listing on new page** **EJECT**

This directive causes the remainder of the current page of the line printer paper to be left blank and the listing to be continued at the top of next page.

Syntax
␣EJECT␣

NLIST **Suspend listing** **NLIST**

The NLIST directive causes the Assembler listing to be suspended from the point where this directive is given until either the END directive or a LIST directive.
Lines which contain errors will continue to be printed during this phase.

Syntax
␣NLIST␣

LIST **Resume listing** **LIST**

The LIST directive causes the Assembler to resume the listing after it has been suspended by a NLIST directive.

Syntax
␣LIST␣

SYMBOL GENERATION

Three directives allow the user to make a number of special instructions for a specific purpose or program, namely FORM, XFORM and GEN. In the FORM directive the user may define the bit configuration and the mnemonic of the special instruction.
If two FORM-defined instructions are to be specified which differ only in the contents of certain fields the programmer may use the XFORM directive.
The GEN directive allows to include the instructions, defined by FORM and XFORM, in the existing Assembler by extending the Assembler's symbol table.
A particular useful pseudo-instruction or system macro can be defined once for all times instead of having to be generated by a FORM directive in every program where it is used.
Symbol generation is only possible with the monitor controlled assembler.

FORM **FORMat definition** **FORM**

This directive is used to define the format of a word or a group of up to 8 words named by an identifier which can be used as an instruction mnemonic later in the program.
The directive is written as follows:

Syntax
< label > ␣FORM␣ < field definition >[,< field definition >.
< field definition > ...< field definition >][/ < field number list >]

where:

< field definition > ::= < field length definition >[[= |:]field value definition >]
< field number list > ::= < field number >[,< field number list >]

and

< field number > ::= < decimal integer >

< **field length definition** > specifies the number of bits to be allocated to a field of the word and may range from 1 through 16. If several fields are defined inside a word the sum of the field lengths must be 16. The maximum number of consecutive words defined by a single FORM directive is 8.

< **field value definition** > can be used to place a value in the field to which it refers when the value is preceded by an equal sign (=).
If the value is preceded by a colon (:) the value indicates the address of a word in relation to the first word of the expansion defined by FORM. The value definition itself may be a predefined expression, an external reference without any displacement or a predefined absolute or relocatable expression. If a particular field has not received a value definition the field will be filled wit zeroes.

< **label** > defines the instruction mnemonic. The operand field of the directive must then contain values to be placed in any non-predefined fields. The last non-predefined value is default value.

Example
MNEM␣FORM␣16 = /85A0,16:14,16 = /8141,16 = INST, 16, 16, 16

/85A0	→arithmetic or logical value
MNEM + 14	→address of word following this block
/8141	→arithmetic or logical value
INST	→identifier
0 - 0	
0 - 0	3 words containing zeroes
0 - 0	

The parameter 16:14 indicates a word address seven words form the beginning of the expansion defined by FORM. The programmer has to specify this address as the last three words are left zero.

Example
This example shows how the programmer may make an ECB if not all parameters are known. By using the FORM directive he does not have to write the instruction sequence:

```
LDK    A7, –
LDKL   A8, DECB
LKM
DATA   1
```

```
00000                          IDENT  FORM
00001              INOUT       FORM   8=/07,8,16=/80A0,16,16=/2804,16=1
00002  0000        BUFFER      RES    10
00003  0014  0008  DECB        DATA   8,BUFFER,20,0,0,0
       0016  0000 R
       0018  0014
       001A  0000
       001C  0000
       001E  0000
00004  0020  0782  START       INOUT  /82,DECB
       0022  80A0
       0024  0014 R
       0026  2804
       0028  0001
00005  002A  2804                LKM
00006  002C  0003                DATA   3
00007                          END    START
       SYMBOL TABLE
BUFFER  0000 R   DECB  0014 R  START  0020  R
ASS.ERR.  00000
.EOF
A::EOF
EXIT
```

From now on the programmer may use INOUT␣/82, DECB instead of LDK␣A7,...

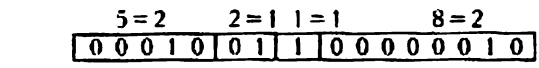
Field number list

If the programmer wishes to put the values of the operand field of the FORM defined mnemonic in an order different from that of the non-predefined field they are to occupy, or if the user wishes to alter the values held by any of the predefined fields, he must use the field number list parameter in the FORM directive.

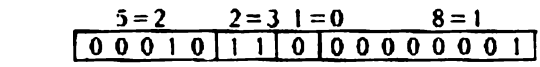
Each field that is generated is given a number, beginning with 0 for the first field, 1 for the second field, n-1 for the nth field (n may not exceed 15).

The field number list must be preceded by a / (slash) and be placed after the last field definition of the FORM directive.
All non-predefined fields specified in the field definition list must also be specified in the field number list.
A field number is represented as a decimal integer.
If a field number list is specified after a FORM directive, the operand expressions following the pseudo-mnemonic will occupy the fields specified in the field number list in the given order. In this way, the contents of predefined fields may be altered while blank fields may be left blank.

Example:
Suppose the user has specified in his program, by means of a FORM directive, a 16-bit word of the following format:



He wishes to have this word changed in:



He may do so by using the following instruction sequence in his program using the field number list in the FORM directive

```
        IDENT   EXAM
        —
        —
WORD    FORM    5=2, 2=1, 1=1, 8=2/2,1,3
        —
        —
        WORD    0,3,1
        —
        —
        END
```

The Assembler will now change the fields as follows:
field no 2 (1 = 1) will be changed to contain the value 0
field no 1 (2 = 1) will be changed to contain the value 3
field no 3 (8 = 2) will be changed to contain the value 1
field no 0 (5 = 2) will keep the value 2

The operand expressions following a pseudo-mnemonic are positional parameters. If one parameter is omitted (other than the rightmost one), its position must be indicated by a comma.

If a FORM defined mnemonic is identical with a standard instruction mnemonic, the pseudo-mnemonic is given priority.

Syntax

< label > ␣XFORM␣ < FORM-defined pseudo-mnemonic >, < field list >

The XFORM may be used each time two FORM-defined pseudo-mnemonics have to be defined which do not differ in the format but only in the values of the predefined fields.
The field list is a series of field definitions giving the format of the new pseudo-mnemonic and the contents of its fields.
The field length definitions must be the same as those of the FORM-directive referred to and appear in the same order.

Example
INST1␣FORM␣8=/FF, 4, 4, 16/1, 3, 2
INST2␣FORM␣8=/33,4,4,16/1,3,2

The XFORM directive combines the two and generates an INST2 instruction as follows:

INST2␣XFORM␣INST1,8=/33,4,4,16

The GEN directive allows to extend the Assembler symbol table so that it recognizes and assembles a number of non-standard symbols in any program in which they are used.

Syntax

␣GEN␣

Restrictions
The GEN directive may only be used in the source program in which it appears if it fulfills the following conditions:

- GEN must immediately precede END
- only the FORM, XFORM, EQU and EXTRN directives are allowed in this program.

The Assembler does not verify if those conditions are fulfilled. It checks only if:

- object code is produced
- assembly errors have occurred.

Example

```
        IDENT␣FORM
INOUT␣  FORM␣8 = /07,8,16 = /80A0,16,16 = /2804,16 = 1
        GEN
        END
```

The following procedure must be followed to include the features provided by GEN:

- load Assembler
- place on the reader the user source module with GEN directive
- assemble this module to produce object output
- load Linkage Editor
- place the Assembler in the reader and have it processed by the Linkage Editor (P)
- place the object user program in the reader and have it processed (P)
- next Terminate (T).

The punched output of this link-editing is the original Assembler extended with one or more new mnemonics.

NAME	MEANING	PREDEFINED VALUE	INTERNAL VALUE
P	Instruction Counter	0	0
A1	Register 1	1	2
A2	Register 2	2	4
A3	Register 3	3	6
A4	Register 4	4	8
A5	Register 5	5	10
A6	Register 6	6	12
A7	Register 7	7	14
A8	Register 8	8	1
A9	Register 9	9	3
A10	Register 10	10	5
A11	Register 11	11	7
A12	Register 12	12	9
A13	Register 13	13	11
A14	Register 14	14	13
A15	stack pointer	15	15

Note: P, A1, A2, A3 etc. can only be used to call the registers. If they are used for other purposes an error message will be output.

Data transfers between input/output devices and the central processor are controlled by device control units each of which may have one or several devices attached to it, depending on the type of device. Control units are attached to the central processor by an interrupt or break line, by address lines and other signal lines which are used by the computer to determine whether a data transfer can be performed.
Data transfers take place through a channel, the General Purpose Bus. The actual programming of the data transfers may be on a character or word basis, where each word or character is programmed and transferred individually via the Programmed Channel or the user may program blocks of words or characters via the I/O Processor. In the latter case external registers may be addressed.

Stand Alone or Monitor controlled programming
The basic difference between Stand Alone programming and Monitor controlled programming is caused by the fact that in Stand Alone programming the user has to write his own input/output routines whereas in Monitor controlled programming the user may call certain monitor functions by means of *links to monitor* which execute the input/output.
For information on programming in either mode refer to the P800M Software Training Manual (Pub. No 5122 991 1243×) and to page 1-55 of this manual.

Interrupt system
When working in interrupt mode each interrupt program may be connected to an interrupt level. As the actioning of an interrupt involves the direct accessing of the interrupt level's start address from its hardware interrupt location, the contents of this location must have been previously loaded with the correct address.
The start addresses loaded in these locations are not fixed and must be defined by the programmer.

interrupt level	*hardware interrupt location*
0 to 62	/0000 to /007C

where level 0 has the highest priority and 62 the lowest. The levels are defined at SYSGEN time (see Volume I).

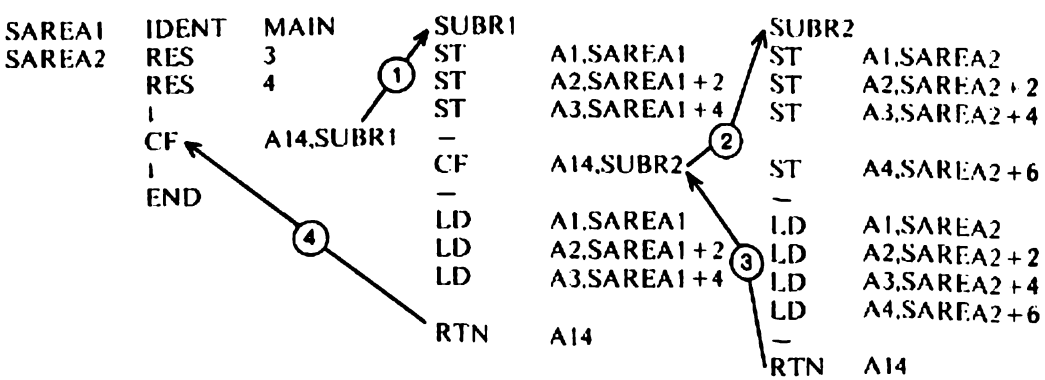
System stack
To save the contents of registers when an interrupt is made into the main program, the hardware interrupt routine automatically uses register A15. This

register addresses the stack which is to hold the contents of the P-register and the Program Status Word at the time the program was interrupted. It is therefore necessary to reserve sufficient space for the stack and to load register A15 with its start address. This may be done by using the appropriate assembly directives and by defining the start address by means of an identifier. The start address is the highest address reserved as the stack is filled from the high towards the lower addresses.
Apart from the contents of the P-register and PSW, the stack may be used to save the contents of other registers as required by the program. These registers are saved by means of Store instructions (1 for each register). Before returning to the main program, Load instructions are required to restore the contents of the stack, prior to RTN. During the hardware action further interrupts are inhibited. If the user wishes to allow the specific routine to be interrupted he must give an ENB instruction.

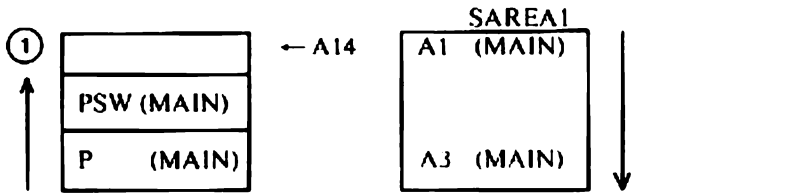
User stack
We have seen that with the A15 stack the P-register, the PSW and any other registers are saved with Store instructions in this stack towards the lower addresses. Now, if a user calls a subroutine with a CF instruction the contents of the P-register and the PSW are automatically stored in a stack he has set up previously, for example as follows:

```
        RES     20
STB     EQU     *-2
        LDKL    A14,STB
                                    then the subroutine is called:
        CF      A14,SUBR
                                    and P and PSW are stored in the
                                    A14 stack
                                    (other registers may also be used
                                    as a stackpointer)
```

For example, for a program with two subroutines, one subroutine calling another one, the saving may be done as follows:

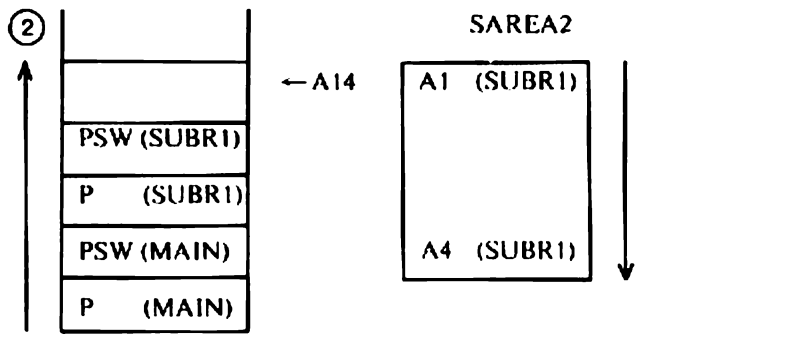


The following save operations take place in this example:



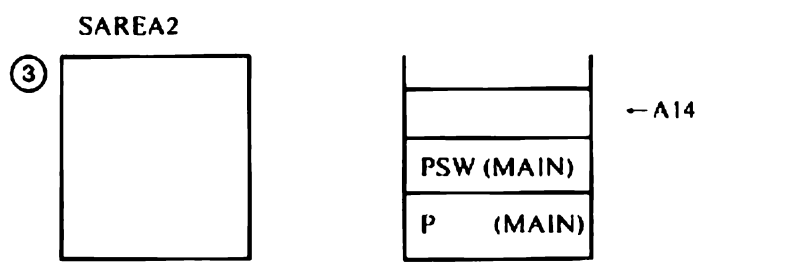
Stored automatically

Stored by user-written instruction



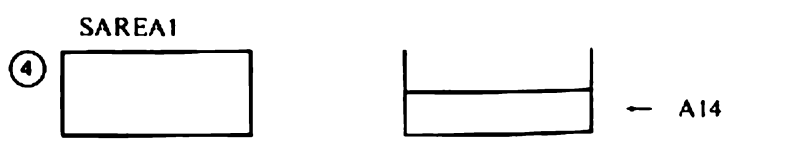
Stored automatically

Stored by user-written instruction



Registers restored for SUBR1

P and PSW restored for SUBR1



Registers restored for MAIN

P and PSW restored for MAIN

Note:
It is possible to return from SUBR2 directly to the main program but in such a case the user must update the A14 register content i.e. the stackpointer himself (with 4, in this case).

***Trap* action**
Instructions input to the P800M computer are checked and decoded by the CPU's Hardware.
If an unexecutable instruction is encountered a *trap* action is started which consists of a hardware and software operation. The hardware operation of the trap consists of the following actions:
- the CPU does not attempt to carry out the instruction
- interrupts are inhibited
- information which refers to the instruction's address and processor status (P and PSW) are saved
- an indirect branch is made to location /7E (start of trap routine).

The software operation of the trap consists of:
- save the address in P
- save the instruction's bit pattern and its second word, if any
- activate the Simulation routine (see below), if any.

Simulation routine
The simulation routine allows the P852M user to simulate the following instructions:

multiply	double shift
divide	multiple load
double add	multiple store
double subtract	

This routine, which is activated each time an illegal instruction code is met in the instruction sequence, consists of two parts. One part analyzing the bit pattern saved by the trap routine and one part executing the instruction listed above.
The routine may be interrupted.
See Appendix G for Stand Alone Simulation Package

Adaptation of P855M software to P800M software

When P855M programs are to be adapted and run on the P800M computer the following points must be taken into account:
1 the sequence ... ENB INH ... in the P855M software permits to have the program interrupted after ENB to see whether an external interrupt is pending. As in the P800M external interrupts are not scanned at the end of a short instruction, a dummy instruction must be included after ENB to allow for an interrupt scan.

The sequence may be altered in ... ENB/RF, +2 /INH ...

2 in the P800M a stack overflow interrupt is given as long as the register A15 contents remains < /100. For the P855M a stack overflow interrupt is generated when the contents of register A15 = /100.

Use of the RTN instruction

Operation of the RTN instruction is slightly different for the P852M on one hand and the P856M and P857M on the other hand. The RTN instruction on the P852M reloads from the system or the user stack (the system stack is pointed to by register A15 and the user stack by one of the registers A1 through A14) the contents of the P register and the PSW as saved when the interrupt routine or subroutine was entered.

On the P856M and P857M the return is as follows:

When one of the registers A1 through A14 is specified, the P register and the CR field of the PSW in the user stack are reloaded. When register A15 is used as a stack pointer, the P register, bits 0 through 7, bit 9 and bit 15 are reloaded from the system stack.

Stand Alone Input and Output Programming

Programmed Channel

To control the data transfer between the device and the CPU the following instructions are, in general, available:

CIO Start	Start input or output
CIO Stop	Stop the input or output
INR	Input one character
OTR	Output one character
SST	Send status of the control unit
TST	Test if the control unit is busy

The register <r3> used in the CIO instruction must always contain additional information for the control unit e.g. input, output, parity, echo etc. Which information must be loaded can be found in the relevant hardware manuals delivered with the system.

When the CIO Start instruction is accepted (test the condition register) it is followed by an INR or OTR instruction. When the last character is tranferred a CIO Stop instruction must be given. This instruction should be followed by an SST instruction which gives the status of the relevant control unit and may reset an interrupt and switch a control unit to the Inactive State.

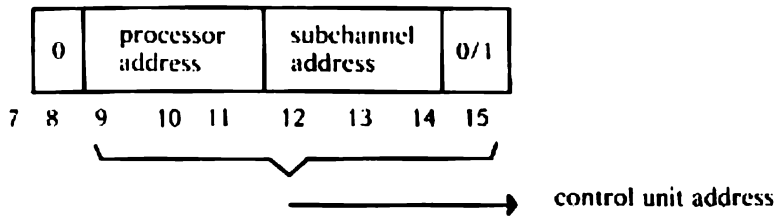
I/O Processor

The I/O processor allows the high speed transfer of variable length or fixed length data blocks between a suitable control unit and the processor.

Up to eight I/O processors may be connected to the General Purpose Bus each of which may control up to eight control units via eight subchannels.

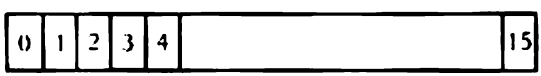
Each I/O processor has implemented two working registers which are used to

effect register to register exchanges with the CPU internal registers.
Before a data transfer can be realised the user has to specify two control words for two external registers. These external registers are addressed by 2 WER instructions in which the address part must be composed as follows:

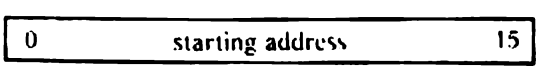


where processor and subchannel address are determined at system installation time. Both addresses, which may range from 0 thru 7, form together the attached control unit address. Bit 15 determines which control word is sent:
bit 15 = 0 1st control word
1 2nd control word

Format of control words
The format of the first control word is:



where:
bit 0 = 1 exchange is in word mode
0 exchange is in character mode
bit 1 = 1 exchange is from memory to control unit (output)
0 exchange is from control unit to memory (input)
bit 2 = 0
bit 3 = 0
bits 4 thru 15 specify the number of characters or words to be transferred.
The format of the second control word is:



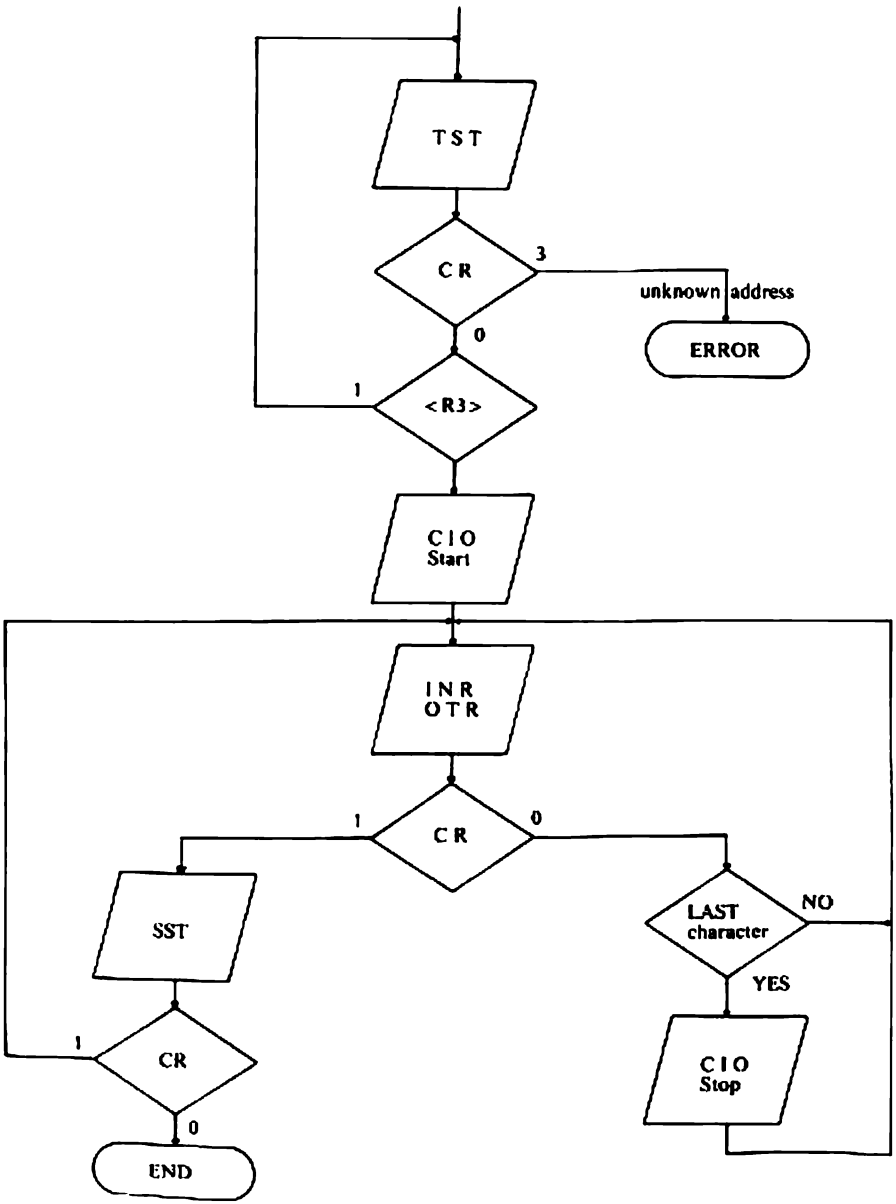
When operating in *word mode* the 1st word of the block is always even (bit 15 = 0)
In *character mode*, and bit 15 = 1, the right hand character is addressed (odd address). When bit 15 = 0 the left hand character is addressed (even address).

Example:

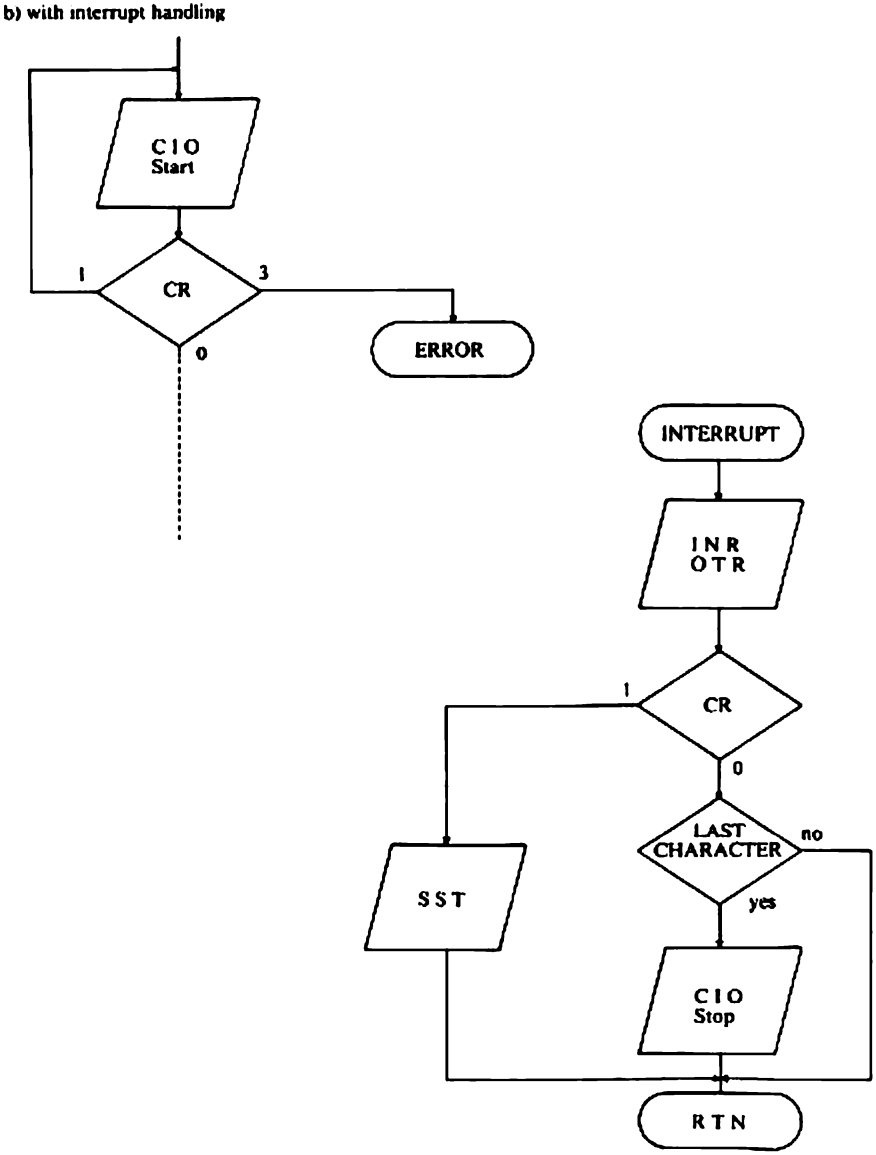
```
        —
        —
        LDKL    A1,/8032        word mode, input, 50 words
        LDKL    A2,BUF          starting address of block
        WER     A1,/A           send control words (1000010 and 1000011)
        WER     A2,/B
        —
        —
        CIO     A4,1,/01        start input (address: 000001)
        —
        —
```

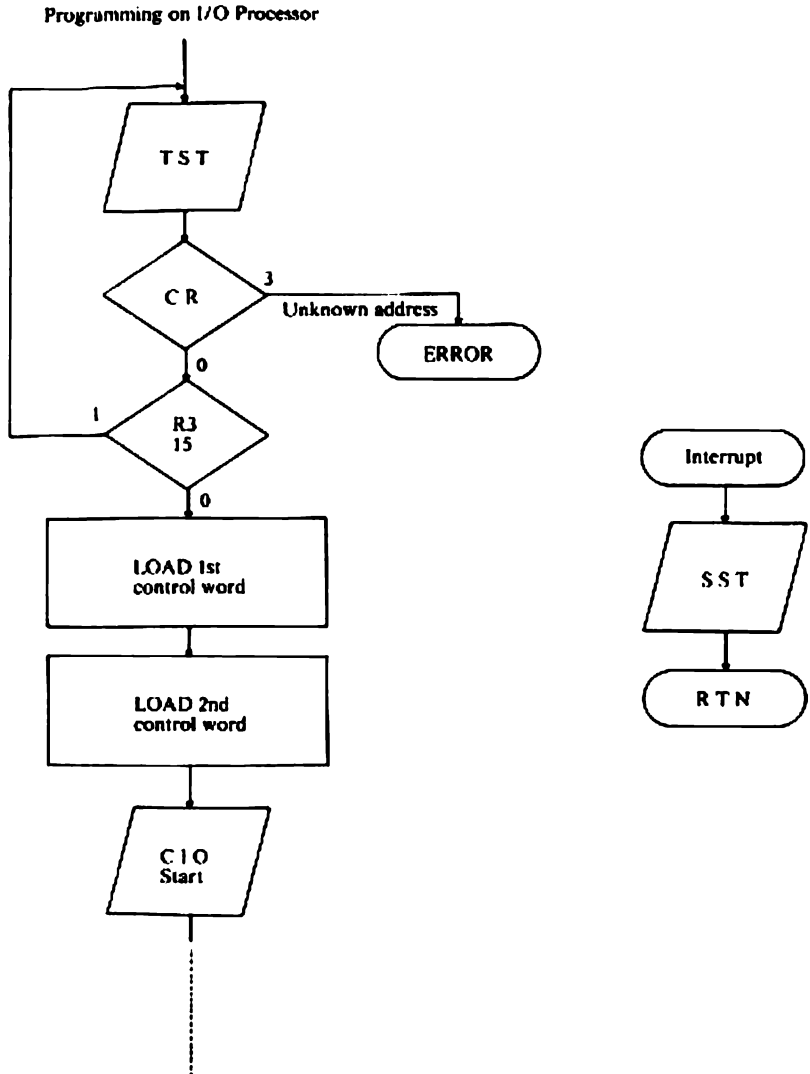
The RER instruction may now be used to read a transfer's effective length after termination of the I/O operation.
When the exchange is completed an SST instruction should check the status of the control unit and set it to the *inactive* state. The control unit may now be re-initialised for a new transfer.

Input/Output Programming on Programmed Channel
a) without interrupts



b) with interrupt handling




Programming on I/O Processor
TST
CR
3
Unknown address
ERROR
0
1
R3
15
0
LOAD 1st
control word
LOAD 2nd
control word
CIO
Start
Interrupt
SST
RTN

```
0000                    IDENT   OUTPUT
0001                    *
0002                    * EXAMPLE OF STAND ALONE PROGRAM TO OUTPUT A
0003                    * MESSAGE ON THE TELETYPE LOG AND NEXT HAVE THE
0004                    * SAME MESSAGE PUNCHED ON THE TELETYPE PUNCH UNIT.
0005                    *
0006                    *
0007 0000 2044          MESSGE  DATA    ' DIT IS EEN TEST',/80A
     0002 4934
     0004 2F49
     0006 532F
     0008 4545
     000A 4E20
     000C 5445
     000E 5354
     0010 802A
0008 0012 287F          START   MLT
0009 0014 208F                  INM
000B 0016 0112                  LDK     A1,18                   COUNTER FOR NO OF CHARACTERS
0011 0018 8840                  LDKL    A8,8
     001A 0008
0012 001C 8000                  LDK     A8,0
0013 001E 4808                  CIO     A8,1,/10                START TELETYPE IN OUTPUT
0014 0020 9C04                  RB(NA)  *-2                     ACCEPTED?
0015 0022 E542          ASR     LC      A5,MESSGE,A8            LOAD A CHAR IN A5
     0024 0000  R
0016 0026 4510                  OTR     A5,0,/10
0017 0028 9C04                  RB(NA)  *-2                     ACCEPTED?
0018 002A 8840                  ADKL    A8,1                    POINT TO NEXT CHARACTER
     002C 0001
0019 002E 1901                  SUK     A1,1
001A 0030 9C10                  RB(NZ)  ASR                     ALL CHARACTERS PRINTED?
0021 0032 4808                  CIO     A8,0,/10                YES, SWITCH TELETYPE OFF
0022 0034 9C04                  RB(NA)  *-2                     ACCEPTED?
0023 0036 4C08                  SST     A4,/10                  SEND STATUS
0024 0038 9C04                  RB(NA)  *-2                     ACCEPTED?
0025                    *
0026                    * PUNCH THE MESSAGE
0027                    *
0028 003A 8840                  LDKL    A8,8
     003C 0008
0029 003E 0112                  LDK     A1,18                   COUNTER FOR NO OF CHARACTERS
0030 0040 0600                  LDK     A6,0
0031 0042 4800                  CIO     A6,1,/10                SWITCH TELETYPE ON IN OUTPUT
0032 0044 9C04                  RB(NA)  *-2                     ACCEPTED?
0033 0046 0512                  LDK     A5,/12                  = SWITCH PUNCH UNIT ON
0034 0048 4510                  OTR     A5,0,/10
0035 004A 9C04                  RB(NA)  *-2
0036 004C E542          PTP     LC      A5,MESSGE,A8
     004E 0000  R
0037 0050 4510                  OTR     A5,0,/10                OUTPUT THE CHARACTER IN A5
0038 0052 9C04                  RB(NA)  *-2
0039 0054 8840                  ADKL    A8,1
     0056 0001
0040 0058 1901                  SUK     A1,1                    ALL CHARACTERS PUNCHED?
0041 005A 9C10                  RB(NZ)  PTP
0042 005C 0514                  LUK     A5,/14                  SWITCH PUNCH UNIT OFF
0043 005E 4510                  OTR     A5,0,/10
0044 0060 9C04                  RB(NA)  *-2                     ACCEPTED?
0045 0062 4808                  CIO     A8,0,/10
0046 0064 9C04                  RB(NA)  *-2
0047 0066 4C08                  SST     A4,/10
0048 0068 9C04                  RB(NA)  *-2
0049 006A 287F                  MLT
004A                            END     START
```

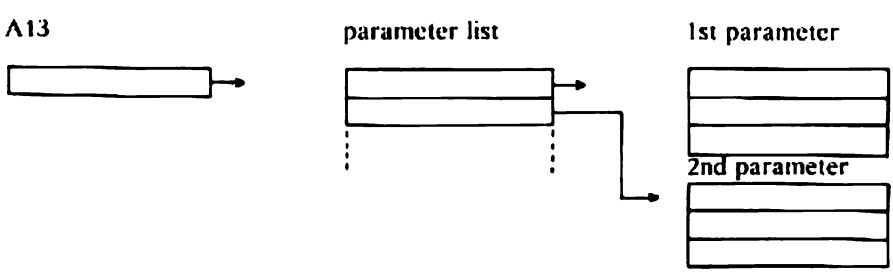
Source program calling a subroutine in FORTRAN library

When writing a program in Assembly Language it may be useful to have a certain operation performed by a subroutine which has been specifically included in the FORTRAN library to execute such a function.
The user may call this subroutine, in his Assembly program, in the following way:

Suppose the user wishes to multiply two floating point numbers. The FORTRAN library subroutine, which executes this multiplication, has F:RM as entry point. The framework of the Assembly program, with only the relevant details, is written as follows:

```
            IDENT   ASMPRO
            EXTRN   F:RM

FLNUM1      DATA    -
            DATA    -
            DATA    -

FLNUM2      DATA    -
            DATA    -
            DATA    -

            LDKL    A13, PARLIS
            CF      A14, F:RM

PARLIS      DATA    FLNUM1
            DATA    FLNUM2
```

Before the CF instruction is executed, register A13 must contain the address of a parameter list. This list must contain the address of floating point number 1 and the address of floating point number 2.



The subroutine in the library contains the following relevant items:

```
IDENT    FRTLIB
ENTRY    F:RM
-
-
-
-

RTN      A14
```

This subroutine does not use the stack of the calling program, except for the return. When values are to be returned to the main program an integer will be returned to A1 and a real value to the registers A1 to A3 inclusive (mantissa in A1, A2 and the exponent in A3).

The main program must now be link-edited or link-loaded with the called subroutine and the FORTRAN library.

The Linkage Editor selects those modules required for program execution.