ERAMCO SYSTEMS

ES 84081B
DAVID-ASSEM
MK - 2

OWNER'S MANUAL

July 1984

84081B-M004001

Printed in the Netherlands

W. van Alcmade str.[illegible]4
1785 LS Den Helder
The Netherlands

Table of Contents

APPENDICES

I_Introduction

From the moment that it was possible to build machine language extensions for the HP-41 calculator, a good assembler/disassembler proved to be indispensable. This is why DAVID-ASSEM has been developed.

DAVID-ASSEM is a ROM consisting of 5 functions, which is designed to provide an easy means of entering Machine Coded programs in a user-friendly format, and for maintaining and displaying thus written programs.

The main function, ASSM, provides a complete operating environment, that resembles the PRGM-mode the calculator provides for entering and editing ordinary programs. It features a redefined keyboard to reduce entering a step to a minimum of keystrokes (varying from 1 to 3 excluding parameters), while persecuing an easy-to-remember correspondence to the instruction mnemonics; program steps are displayed in Stephen Jacobs' notation. As an alternative a HEX-mode may be used for keying in and displaying hex-data-items. A powerful feature is the possibility to assign an alphanumerical label to any address (even in ROM), which may then be used in stead of it while entering jumps etc., and will also be used automatically whenever applicable in displaying program steps. It is even possible to refer to an as yet undefined label, which will automatically be resolved as soon as the label is defined. Also the ASSM function will perform the cumbersome computations needed to decide between the various forms of branches, gotos and (relocatable subroutine jumps. Text-stings following a ?NC XQ 07EF call are displayed as such. Finally it is possible to step forward and backward through a program. The most significant difference with the behavior of the calculator's PRGM-mode, is that ASSM will not relocate code-segments, and therefore will OVER-WRITE steps in the code rather than inserting them.

Because Machine Code knows no labels physically present within the code (like with user-code), this information is stored seperately by ASSM. This is done in storage space seperate from the user-registers, namely an I/O buffer, to achieve transparancy to the user. However, two functions BUF>REG and REG>BUF are provided to enable saving and restoring the buffer information using such devices as card-readers: they transfer the information from a buffer to the user-register or back. Actually the information of any I/O buffer can be handled by thes functions.

In order to enable printing the program in the same format as ASSM (including labels under user-program control, the functions BEG/END and DISTOA are provided.

David-Assem is a program, or operating system if you want, that can be extended in the future by users themselves or by ERAMCO SYSTEMS. Yet now there is a 4K ROM available through ERAMCO SYSTEMS that adds all mainframe entries as they are listed in the VASM listings of HP, but more information about this ROM can be obtained through ERAMCO SYSTEMS.

II Special Features

David-ASSEM provides five functions, of which the function ASSM is the most important. Therefore we shall treat this function first.

If you enter ASSM mode by executing the function ASSM, automaticaly the step is viewed where you were, while leaving ASSM mode for the last time. If it is the first time you enter ASSM mode, a beginning address is asked.

Every step in ASSeMbler mode is (in general) viewed according to Steven R. Jacobs's notation, however, there are some deviations, such as C=M ALL which is viewed as C=M, G=C @R, which is viewed as G=C.

II.1. SST- and BST-ing with ASSM

Being in ASSM mode you can "walk" through any machine level program by pushing SST and BST, which are located on the SST and BST key. One, two, three or multi-word instructions are viewed as one step, both SSTing and BSTing.

e.g. Put on USER mode, assign ASSM to [Σ+], and

Push	See	Comment
ASSM(Σ+)	BEGIN ____	here you enter ASSM mode
0000	?NC GO 0180	step 0000
SST	?NC GO 01AD	step 0002
SST	C=0 ALL	step 0004
SST	RAM SLCT	
SST,SST	M=C	step 0007
BST	READ 3(x)	step 0006
[<-]	CONT. ____	
[<-]	{x-register}	here you've left ASSM mode

You can see here that program steps consisting of more than one word are disassembled as one step.
?NC GO 0180 is actually 201, 006, two words long.

Auto repeat

A feature of SST and BST is that if one of these is pressed longer than one second, they will act like they are pressed every 1/2 second. In other words, if you keep pressing the SST key, every 1/2 second the next (or previous) line will be viewed.

e.g.

Push	See
ASSM	READ 3(x)
SST (hold)	M=C
¦	C=ST
¦	C=0 xs
¦	CLRF 7
¦	C<>ST
release	?FSET 6
BST (hold)	C<>ST
¦	CLRF 7
¦	C=0 xs
release	C=ST

II.2 Disassembling data words

ASSM provides steps being viewed as data words automatically in cases that the HP41 CPU treats them as such. This is after a LDI, after a SELP instruction and after a few mainframe subroutine calls.

	Push	See	Comment
e.g.	[<-]	CONT. ____	where do you want to continue?
	07F9	LDI	
	SST	HEX:0FD =,	step 07FA; this step is viewed as being a data word instead of an instruction.
	[<-] 02E0	?NC XQ 22F5	22F5 is ERROR subroutine needs a dataword, 038
	SST	HEX:038 8	

II.3 Disassembling characters

Sometimes ROM words are meant as character data. In one particular case, after a ?NC XQ 07EF (MESSL), ASSM places the characters one after another between quotes in the display. All characters will be treated as one step.

	Push	See	Comment
e.g.	<- 2FEE	?NC XQ 07EF	07EF needs a character string after its call
	SST	"XROM"	
	SST	RTN	

II.4 The USER-off mode

ASSM knows a special mode for the case that instructions should be disassembled or assembled (in tables etc.) as data words: the user-off mode. You can enter or leave this mode by pressing USER key, which toggles the annunciator too.

	Push	See	Comment
e.g.	[] BST	"XROM"	this is step 2FF0 again
	user	2FF0 018 X	
	SST	2FF1 012 R	
	SST	2FF2 00F O	
	[] BST	2FF1 012 R	
	USER	A=0 P-Q	

This last step has no meaning, for it means "R" of "XROM". You probably have noticed that in USER-off mode also a 4-digit hexadecimal address and a display character appear in the display.

II.5 Labels

A very helpful tool is the label option. To any given address a label can be assigned, and you can store up to 254 labels and their locations in HP 41CV memory. Labels themselves will be viewed and all kinds of jumps to addresses to which a label is assigned will be disassembled as jumps to those labels.

e.g.

Push	See	Comment
<- 000D	JNC +0B 0018	000D + 0B = 0018
<- 0018	A<>C ALL	assign to step
[] LBL ALPHA	LBL' _	0018 label "HERE"
HERE Alpha	LBL'HERE	
<-000C	?FSET 6	
SST	JNC +HERE	remember, this was a JNC +0B 0018 !
SST	?FSET 5	
SST	JNC +HERE	
<- 0025	jnc -HERE	here stood a JNC -0D 0018
<-ALPHA HERE ALPHA	LBL 'HERE	step 0018
SST	A<>C ALL	still step 0018!
LBL ALPHA ALPHA	A<>C ALL	clear label
<-ALPHA HERE ALPHA	NONEXISTENT	you just deleted it!

You can see here that a label can be assigned in a simpel way, and that it can be purged in a comparable way.

Also you will have noticed that error messages may appear in ASSM mode; however, you stay in ASSM mode.

II.6 The Redefined keyboard

In ASSM mode the entire keyboard is redefined, although we only used keys with a comparable function as yet. This feature provides a better and faster editing of machine language programs. Let's assume that in the following examples your free MLDL RAM page is A000-AFFF.

e.g.

Push	See	Comment
<- AA00	NOP	this may be another step if you didn't clear this area.
STO (WRITE DATA)	WRITE DATA	this is step AA01 !
R/S (RAM SLCT)	RAM SLCT	step AA02 !
R↓ (POP)	POP	keep it pressed, see after a while
	NULL	annulling key sequence works in the same way as in user code and after you released the key.
	RAM SLCT	
[] BST	WRITE DATA	
√x (C=)	C = _	
+	C = C+ _	
Σ+ (A)	C = C+A _	some instructions take more than one keystroke.
EEX	C = C+A S&X	
[]e[x] (LDI)	LDI	after a LDI, SELP, or special subroutine call, you are supposed to enter hex data
1	HEX:1__	
23	HEX:123 #	
<- AA00	NOP	See above
SST	WRITE DATA	
SST	C= C+A S&X	

Something you ought to get used to is the fact that when you key in new program steps you write them over the next step, not over the step that is viewed.

You can find how the keyboard is redefined in Appendix A on page -47-, you can find the key sequenses for all instructions in Appendix C on page -51-.

II.7 Keying in Data words

In USER-off mode the Hexadecimal keyboard is active: Hexadecimal datawords in 244 format (0-3 key, 0-F key, 0-F key) will be programmed just like instructions mentioned in II.6.

e.g.

Push	See	Comment
USER	AA02 206 F	206 is code of C=C+A S&X
1	HEX:1__	
2	HEX:12_	
3	AA03 123 #	HEX 123 is written over the next line
4	AA03 123 #	You can't key-in a dataword beginning with a 4-F ! The display will blink.
USER	JNC +24 AA27	123 is the code for JNC +24.

II.8 Jumps

Jump instructions can be keyed in in quite a lot ways. The most powerful is the one using labels. The following examples will give you an idea of the capacity of the label feature.

e.g.

Push	See	Comment
<- 0000	?NC GO 0180	assign label "STRTUP" to address 0000
LBL ALPHA STRTUP ALPHA	LBL'STRTUP	
<- AA00	NOP	continue in scratch area
[] GTO ALPHA	GTO _	
STRTUP ALPHA	NC GO 'STRTUP	now a ?NC GO 0000 has been programmed.
XEQ []	XEQ C ____	
0000	?C XQ 'STRTUP	
<- AA10	NOP	
LBL "TRY"	LBL'TRY	
[] GTO []	GTO C ____	jump to try
"TRY"	JC- TRY	ASSM computes the distance
[] GTO "QWERTY"	JNC? QWERTY	You didn't say yet where QWERTY is located!
<- AA20	NOP	
LBL "QWERTY"	LBL'QWERTY	Now ASSM can know it,
<- AA12	JNC+ QWERTY	And it does!

These are just a few things that can be done with labels, but with these examples you can see that ASSM keeps all information: where labels are located, and which labels have not yet been defined.

All these features will be discussed later, in chapter IV, how they work, what they do, etc.

II.9 BUF>REG and REG>BUF

This pair of functions allows you to save any I/O buffer at any time you need them. When the contents of an I/O buffer has been moved in user-registers, you may save them on magnetic cards or tape.

These functions have been incorporated in DAVID-ASSEM because the machine-language labels, or ASSeMbler labels are stored in an I/O buffer, namely a buffer with ID#1. This means that you can save your own labels for later use.

II.10 BEG/END and DISTOA

This pair of functions allows you to make a hardcopy of your progams on your printer, by making a simple user-code program.

The advantage of these functins compared to other disassemblers is that DISTOA also translates labels, just as they are viewed in ASSM mode, and since there is a ROM available that adds all mainframe-entries, the listings will be very easy to read.

II.11 Possible extensions

DAVID-ASSEM can easily be extended by one or more 4K blocks which can add big features to the yet powerful 4K of DAVID-ASSEM.

At this moment a ROM has been written, namely the one mentioned above.

III Warnings

- David-Assem provides the functions BUF>REG and REG>BUF. These functions should be executed with much care. Read before you use them, the chapter "User-instructions" carefully, a crash could be the result of a wrong use.

- In ASSM mode nearly every key stroke is a write-action in RAM. Be careful when you are in ASSM mode not to be too nonchalant with pushing keys. A program you made earlier could be destroyed partially by doing this.

- There are some deviations from the Steven R. Jacob's notation for disassembling in ASSM mode; look them over! (they are listed on page 16).

- Your HP41 remembers all labels in machine language when you leave ASSM mode, even if you turn the calculator off. However, if you turn the calculator on without having DAVID-ASSEM plugged in, all labels are purged.
  With BUF>REG and REG>BUF you can save the labels, even if you carry no MLDL box with you!

- ASSM may go PACKING the memory. Then the message TRY AGAIN will appear in display. if this message appears again after repeating the instruction that caused ASSM to pack, you have to leave ASSM mode and change the size such, that there is memory free again.

IV User instructions

First we have to make a few agreements :

- You are in ASSM mode (Assembler mode) if the function ASSM is being executed.
- By "instruction" we mean a machine language instruction such as LD@R 6, ?NC GO 1234 etc.
- By "function" we mean a user-code function, such as ABS, ASSM, DISTOA
- In examples, the alpha key will be represented as a quote, " , the shift key as []
- Something between {} should not be copied to the letter.
- Examples are given in chronological order, so earlier ones can influence later ones. You should do them all at least once!
- In the Examples it is assumed that you have not plugged in "MNFR-LBLS"
- Most examples take place at address AA00 and further. It is assumed that this is filled with NOP's at the beginning. If you can't switch a free ram page to A or your ram is filled at XA00, you may chose of course an other ram address, but in that case the examples won't be exactly correct.

A. The function ASSM XROM 02.01

The not-programmable function ASSM enables you to enter ASSM mode. ASSM uses no data memory (user registers) or program memory, or stack registers or extended memory. When you leave ASSM mode, everything will be the same as it was before you entered ASSM mode.

	Do	Comment
e.g.	CLX, STO 00, XEQ "REG>BUF"	Now you have initialised ASSM mode

Do/Push	See	Comment
1 enter 2 enter 3 enter		fill the stack
4	4_	
"TEST 1,2,3"	4,00	fill alpha register
XEQ "ASSM"	BEGIN ____	
0000	?NC GO 0180	now you entered ASSM
<-	cont. ____	continue....?
<-	4,00	now you left ASSM mode
R↓, R↓, R↓	1,00	you can see the stack
ALPHA	TEST 1,2,3	you can see the alpha
ALPHA	1,00	register.

IV A1 How to enter ASSM mode

The right way to do this is to execute "ASSM", or to push the key to which ASSM is assigned.

Because you will switch very often from normal mode to ASSM mode it is nice to have ASSM assigned to +, for instance.

e.g.

Push	See	Comment
[] ASN		
"ASSM" Σ+	1,00	
ASSM (Σ+)	?NC GO 0180	Now you are in ASSM mode

Sometimes it happens that executing ASSM will cause a prompt "BEGIN ____" to be displayed, or that PACKING - TRY AGAIN will appear in the display, but this does not happen very often. It will be discussed later when and why this happens.

IV A2. How to leave ASSM mode

There are two ways in which you can leave ASSM mode:

a. by twice pushing the back-arrow key ([<-])

e.g.

Push	See	Comment
[<-]	CONT. ____	you're still in ASSM mode
[<-]	1,00	you left ASSM mode

It could happen that an error message "? LBL' {label name} appears in display. This is to remind you of the fact that there still is a pending jump to a nonexistent label. In that case you have left ASSM mode too. (see IV A 4.8 d, page 41)

b. by turning the calculator off. If you turn the machine on again, you are not in ASSM mode any more.

e.g.

ASSM	?NC GO 0180	you are in ASSM mode again
ON	machine is off	
ON	{X-register}	you have left ASSM mode

IV_A3_The_ASSM_mode_as_disasembler

The ASSM mode has its own program counter, that is a 4 digit hexadecimal address that points to the step which is viewed in display. This counter shall be called "Assembly Program Counter", or APC.

The ASSM mode knows two sub-modes concerning both assembling and disassembling:

a. User-on : The machine-language program steps are disassembled according to Steven R. Jacobs's notation, and the redefined keyboard (see appendix A) is active.
b. User-off : Steps will be shown as HEXadecimal values, and a hexadecimal keyboard is active.

You can toggle this sub-mode by pressing the USER key. The USER annunciator shows in what mode you are.

IV_A3.1_User-off in disassembling

e.g.

```
ASSM          ?NC GO 0180
USER          0000 201 A

APC           10-bits ROM word        Display character
```

You see that the display is divided in three parts in this mode:

1. The 4-digit hexadicemal value of the current APC. (the address on which the ROM word is that is viewed).
2. After two spaces followed by a 3-digit hexadecimal value of the 10-bits ROM word located at the APC.
3. After two spaces followed by a display character & punctuation, where the lower 6 bits (0-3) of the ROM word indicate the character itself (rows 2 - 6 ascii) and the lower 2 of the upper 4 bits (6-7) of the ROM word indicate the punctuation.

   (00=nothing, 01=".", 10=":" and 11=",").
   The upper 2 bits (8-9) of the ROM word are not used.

The value of the APC can be changed by using

- SST : APC := APC+1
- BST : APC := APC-1, if APC<0 or >FFFF, APC := 0
- [<-] (CONTinue at) abcd (hex) : APC := 0
- [<-] (CONTinue at "{label}" : APC := {address of label}

The defenitions behind the key only apply in user-off mode.

e.g.

Push	See	Comment
SST	0001 006 F	APC : = 0000 + 1 = 0001
SST	0002 2B5 5:	APC : = 0002
[<-] (CONT)	CONT. ____	Where do you want to continue
2BF7	2BF7 130 0	APC : = 2BF7
BST	2BF6 28B K:	APC : = 2BF7 - 1 = 2BF6

IV A3.2 The User-on mode in dissasembling

In this mode one or more words are disassembled as one single step in the display, for 95% according to Steven R. Jacob's notation.

e.g.

Push	See	Comment
USER	JNC- 2F 2BC7	the address to which is to be jumped is added.
<-	CONT. ____	
0000	?NC GO 0180	APC : = 0000
SST	?NC GO 01AD	APC : = 0000 + 2 = 0002 because the last step was 2 words long
SST	C=0 ALL	
SST	RAM SLCT	
SST	READ 3(x)	APC = 0006

There are quite a few exceptions on Jacob's notation :

a. - sometimes a field is omitted, in cases that there is no alternative These cases are listed below.

Jacobs

C=M ALL	->	C=M	C=N ALL	->	C-N
M=C ALL	->	M=C	N=C ALL	->	N=C
C<>M ALL	->	C<>M	C<>N ALL	->	C<>N
G=C @R,+	->	G=C	POP ADR	->	POP
C=G @R,+	->	C=G	PUSH ADR	->	PUSH
C<>G @R,+	->	C<>G	LDI S&X	->	LDI

b. - with short form jumps (JC and JNC) the address to which is to be jumped is added. (see example above)

c. - Relocatable jumps and jump subroutines are disassembled as respectively as GOTO {addres/label} and GOSUB {address/label}.

d. - Datawords are viewed as : HEX: {3-digit hexadecimal value} {display character & punctuation}.

e. - strings are viewed in one step between quotes
f. - READ 0(T) is viewed as READ DATA

Now a few examples of these will follow:

	Push	See	Comment
e.g.			
(a)	SST	M=C	ALL is omitted
(b)	CONT. 000D	JNC +0B 0018	0018(=000D+0B) is added

Assumed X the page of DAVID-ASSEM:

	Push	See	Comment
e.g.			
(c)	CONT.{X}3BB	GOSUB {X}FAF	relocatable XQ.
	SST	LD@R 0	what is the next step
	USER	{X}3BE 010 P	APC is incremented by 3BE - 3BB = 3!
(a)	CONT. 07F6	LDI	S&X is omitted
	USER		
(d)	SST	HEX: 010 P	data word
	CONT. 2FEE	?NC XQ 07EF	
(e)	SST	"XROM "	5 words in one step
	SST	RTN	APC : = APC + 5

In USER-on mode the APC can be changed by using:

- SST : APC : = APC + {length of step at old APC}
- BST : APC : = APC - {length of step at new APC}
  if APC < 0, APC = 0 and if APC > FFFF, APC : = 0
- CONT. {address}/"{label}" : APC : = address/address of label

e.g.	BST	"XROM "	APC : = APC - 5
	BST	?NC XQ 07EF	APC : = APC - 2

Note: Because in machine-language a BST can not be defined, a wrong APC value may be computed by executing BST. In that case "the previous step" is implemented wrong by ASSM, but this happens in only 1% (or less) of the cases you do a BST.

IV A3.3 Instructions after a SELP instruction

The SELP instruction (SELect Peripheral) is used to control smart peripherals. After a SELP instruction is executed by the HP41 processor, it ignores all following instructions up to and including the word of which the hexadecimal value is odd (in other words: bit 0 of the 10-bit word is 1), because these instructions are disassembled as data words, for the cannot know how the selected peripheral implements those instructions.

This may sound a little bit complex, and therefore we shall show you some examples.

	Push	See	Comment
e.g.	CONT.100D	SELP 0	this is actually data, but it will work as well
	SST	HEX:021 !	odd value, so it must be the last data word.
	SST	XQ->GO	and it is!
	CONT.1399	SELP 2	
	SST	HEX: 0AA *:	even
	SST	HEX: 0B2 2:	even
	SST	HEX: 0BB ':	odd; last word
	SST	C=B @R	normal instructions

If we do a BST, ASSM recoqnizes the SELP instruction four words earlier, so a DATA word is viewed.

e.g.	BST	HEX: 0BB ':

IV A3.4 The label option

It is possible in ASSM mode to assign labels up to six characters to an address.

The procedure of assigning a label is as follows

- go to the step to which the label is to be assigned using SST, BST or CONT.
- push [] LBL ALPHA {label name} ALPHA

Then the label and its location is kept in memory until it is changed or deleted.

	Push	See	Comment
e.g.			
	CONT.0018	A<>C ALL	suppose you want to assign "TEST" to 0018
	[] LBL "TEST"	LBL'TEST	finished.

If we pass a label by SST-ing, the label will be viewed before the step to which it is assigned.

	Push	See	Comment
e.g.			
	CONT.0016	READ DATA	APC = 0016
	SST	JNC+ 09 0020	APC = 0017
	SST	LBL'TEST	APC = 0018
	SST	A<>C ALL	APC = 0018 still
	SST	READ 13(c)	APC = 0019

If we pass the label by BST-ing, the label won't be viewed:

	Push	See	Comment
e.g.			
	BST	A<>C ALL	APC : = APC - 1 = 0018
	BST	JNC +09 0020	APC = 0017, label is skipped
	BST	READ DATA	APC = 0016

If we continue at an address to which a label is assigned this will be viewed.

e.g. CONT 0018 LBL'TEST

Jumps to addresses with labels are dissembled as jumps to those labels :

Push	See	Comment
e.g.		
CONT. 0011	JNC+TEST	Jump in positive direction to a label called "TEST"
USER	0011 03B	03B is code for JNC +07
CONT. 0025	0025 393 C:	code for JNC- 0D
USER	JNC- TEST	jump in negative direction to a label
CONT. 0004	C=0 ALL	create a mainframe label
LBL"ADRFCH"	LBL'ADRFCH	
CONT. 004C	NC XQ ADRFCH	a ?NC XQ 0004

A label can be purged simply :

- go to the label using SST,BST, or CONT.
- Push [] LBL ALPHA ALPHA

Push	See	Comment
e.g.		
CONT. ALPHA	CONT._	say want label name
TEST	CONT.TEST_	
ALPHA	LBL'TEST	continue at label test.
USER	0018 0AE }:	check address
USER []LBL	LBL '_	
ALPHA ALPHA	A<>C ALL	
[] BST SST	A<>C ALL	there is no label indeed

In a similar way a label name can be changed :

	Push	See	Comment
e.g.			
	CONT "ADRFCH"	LBL'ADRFCH	
	[] LBL "CHANGE"	LBL'CHANGE	you changed ADRFCH to CHANGE
	CONT "ADRFCH"	NONEXISTENT	error message
	USER	0004 04E N.	
	USER	LBL'CHANGE	it really is CHANGE!

IV_A3.5_Auto_repeat

SST and BST are auto-repeat keys, if they are pushed a longer while, you will SST or BST through your program automatically. This feature is incorporated in order to save your [SST] key. Try it. You'll see how it works.

IV_A3.6_Unused_instructions

Unused instructions are normally disassembled as ???, but there is one exception: 040 is disassembled as WRIT S&X, because this is the normal MLDL RAM write.

e.g.	CONT 1002	???	code is 100

IV A4 ASSM mode as assembler

Now we have come to the point why you bought DAVID-ASSEM : the powerful assembler of ASSM.

ASSM redefines the entire keyboard, but maybe you have not noticed it yet, because we only used keys with a similar function : USER, ALPHA, SST, [] BST, [] LBL

In figure 1 of appendix A the redefining of the keyboard is shown. Most mnemonics you will recognise as Jacobean mnemonics, but a few seem incomplete, such as [C=] and [C<>].

It is very advisable to make your own keyboard overlay on which you mark the meaning of the keys as shown in appendix A, figure 1. You can write the unshifted mnemonics for instance from top to bottom next to the keys.

All instructions exist of one or more keystrokes. A few things can be said about the keyparser :

- a key sequence is still incomplete if a prompt is in display.
- a key sequence can be annulled by

a. holding the last key of a sequence (this is the key which causes no prompt any more) for about a second: NULL will appear in display.

b. pushing one (or more if it is needed) time(s) the back arrow key, until the display will be cleared while holding the key.

This also applies for CONT. and LBL.

	Push	See	Comment
e.g.			
	CONT. "CHANGE"	CONT. CHANGE_	sequence not ended yet
	ALPHA (hold)	CONT. CHANGE_ NULL	
	release key	{same step as before CONT.}	annulled
	√x (C =)	C = _	start sequence
	+	C = C̄ + _	
	[<-] (hold)		display is cleared
	release key	{same step as before [C=] key}	annulled

Before we go further into details about how instructions should be keyed in, you must become aware of the following:

Every programstep you key-in, will be written OVER the NEXT program step.

This may differ from what you are used to, with other programs, but the way in which the over-writing happens here is really the most logical. (compare for instance with HP 25; there you write also over the next step!)

Like the disassembler, the assembler (it is not so easy to distinguish the disassembler and assembler, for they both appear in ASSM mode, but they will be treated apart, as you may have noticed) has two sub-modes: User-on and User-off.

We will treat them one after another.

a. User-off: Now a hexadecimal keyboard is active. You can key-in a 10 bit ROM word by pushing three digits repectively in the range 0-3, 0-F, and 0-F.

	Push	See	Comment
e.g.			
	CONT. AA00	NOP	APC = AA00
	USER	AA00 000 @	turn to USER-off mode.
	2	HEX: 2__	expected two more digits
	A	HEX: 2A_	
	4	AA01 2A4 $:	Remember? NEXT step!

Do you still remember the annulling sequences?

	Push	See	Comment
e.g.			
	4 > BLINK >	AA01 2A4 $:	first key must be 0-3
	1	HEX: 1__	
	2	HEX: 12_	
	<-	HEX: 1__	
	<-	AA01 2A4 $:	No result; annulled
	123 (hold)	HEX: 123	
		NULL	No result; annulled

b. User-on: Now all instructions really will be assembled: All keys have an own function now, except for PRGM and ALPHA. In appendix A, figure 1 you can see the assignments of the keys.

There are several kinds of instructions. We shall treat them one by one.

IV A4.1 Direct Programmable instructions

These are almost all miscellaneous instructions; this group takes the biggest part of the keyboard.
The keying-in sequence simply exists of pushing the key to which the key is assigned:

	Push	See	Comment
e.g.	USER [] BST	NOP	
	R=R+1 (X<>Y)	R=R+1	one key stroke!
	USER	AA01 3DC \,	3DC is opcode of R=R+1
	USER	R=R+1	
	READ DATA (RCL)	READ DATA	READ 0(T) is READ DATA
	[] ST=0 (10^x)	ST=0	

Here you have a list of all direct programmable instructions and the keys related to them :

[] ST=0 (10^x)
[] R=R-1 (CLΣ)
[] WRIT S&X (SIN^{-1})
WRITE DATA (SST)
[] M=C (CAT)
RTN (EEX)
[] ST=C (CF)
SETHEX (5)
[] SLCT Q (R-P)
[] T=ST (FIX)
[] CLRKEY (ENG)
RAM SLCT (R/S)

[] LDI (e^x)
POP (R↓)
FETCH S&X (COS^{-1})
READ DATA (RCL)
G=C (CHS)
[] ?NC RTN (RTN)
ST<>T (4)
[] SLCT P (P-R)
?P=Q (*)
[] DSPOFF (SCI)
[] POWOFF (PI)
PRPH SLCT (VIEW).

R=R+1 (X<>Y)
[] PUSH (%)
[] XQ->GO (ASN)
N=C (ENTER↑)
[] ?C RTN (ISG)
[] NOP (CLX)
[] ST=T (BEEP)
SET DEC (6)
[] ?LOWBAT (X>Y)
?KEY (3)
[] DSPTOG (LASTX)

IV A4.2 LD@R and SELP

These instructions need a hexadecimal digit as parameter. If LD@R (LN) or []SELP (SF) is hit, the instruction name appears, followed by one prompt. After that you can key-in the parameter,[0] to [9] or [A] to [F]

e.g.

Push	See	Comment
[] SELP	SELP _	needs a hex digit.
A	SELP A	instruction is programmed now
001	HEX: 001 A	selp needs a data word
LD@R	LD@R	
3	LD@R 3	sequence finished

IV A4.3 RCR, R=, ?R=, SETF, CLRF, ?FSET, ?FI

These instructions need a decimal number between 0 and 13. However, after pushing one of these keys, only one prompt appears in display. So it is easy to key-in a number between 0 and 9. If you want to have a 10-13 parameter, you should push first EEX, and after that you can push 0 to 3. (compare this with the normal user code function GTO. 1234, it should be keyed-in as GTO [.] EEX 234).

e.g.

Push	See	Comment
SETF (7)	SETF _	want a digit
8	SETF 8	finished
[] ?R= ($\tan^{-1}$)	?R= _	want a 0 to 9
EEX	?R= 1	want a 0 to 3
3	?R= 13	finished
RCR (LOG)	RCR _	suppose you had in mind 11
EEX	RCR 1_	and you made a mistake,
<-	RCR _	you wanted a 3
3	RCR 3	press <- and 3!

IV A4.4 WRIT and READ

These instructions need a decimal number between 0 and 15, or a stack register name :

a. decimal 0-15 :

This goes like 4.3, exept that the range is 0-15 :

- push instruction key (WRIT (sin) or READ (cos))

- push
  - a number in the range of 0 to 9, or
  - EEX followed by a number in the range 0 to 5.

e.g.

Push	See	Comment
WRIT	WRIT _	want a 0-9
EEX	WRIT 1_	want a 0-5
5 (hold)	WRIT 15	register # 15
release	WRIT 15(e)	is called (e)
READ	READ _	
5 (hold)	READ 5	don't hold too long!
release	READ 5(M)	finished

b. stack register name. (compare with STO [.] X)
- push instruction key (WRIT or READ)
- PUSH [.] (decimal point)
- push one of the keys shown in appendix A, figure 2, which mean register names :

A(a), B(b), B(b), D(d), E(d), K(!-), L(L), M(M), N(N), O(O), P(P), Q(Q), T(T), X(X), Y(Y), Z(Z).

	Push	See	Comment
e.g.			
	WRIT	WRIT _	Suppose you want a WRIT X
	[.]	WRIT ST _	now you must hit a reg. key
	X (hold)	WRIT X	don't hold too long!
	(release)	WRIT 3(X)	finished
	READ	READ _	suppose you want a READ d,
	EEX	READ 1_	but you forgot the number
	<-	READ _	PRESS <-, [.], D
	[.] D	READ 14 (d)	and your problem is solved
	WRIT	WRIT _	WRIT T
	[.]	WRIT ST _	
	T	WRIT 0(T)	
	WRIT	WRIT _	WRIT ⊢
	[.] K	WRIT 10(!-)	

The only register which is assigned in a strange way, is 10(-) that is assigned to the K-key. (APPEND character is [] K)

IV A4.5 Singular arithmetic instructions

These instructions are known as class 2 instructions according to Steven Jacob's work. They need a "field" parameter. The right key sequence is as follows :

- push the instruction key: the name appears in display with a prompt.
- push one of the "field" keys as shown in appendix A, figures 3 :

ALL (A), R<- (RDN), M (M), MS (CHS), S&X (EEX), P-Q (-), @R (R), XS (X).

	Push	See	Comment
e.g.			
	LSHFA (1)	LSHFA _	key in: LSHFA S&X
	S&X (EEX)	LSHFA S&X	finished
	[] B=0 (y^x)	B=0 _	key in: B=0 ALL
	ALL (A)	B=0 ALL	finished

The instructions that are part of this group are :

[] A<>B (Σ-)	[] B=0 (y^x)	[] ?A#0 (X=Y?)
?A#C (-)	?A<C (*)	[] ?A<B (X<=Y?)
LSHFA (1)	RSHFB (2)	[] ?B#0 (X=0?)
?C#0 (/)	RSHFA (0)	RSHFC (.)
B=A (1/x)		

IV A4.6 plural miscellaneous instructions

Because there are just 35 keys for a lot more instructions available, quite a few are gathered under a few initial keys. These initial keys are A=(Σ+), [] C<> (x^2) and C= ($\sqrt{x}$), and they can be followed by one, two or three key strokes. Some of the Jacobean class 2 instructions are reached one or two key touches after a A= or C=. Then they act like the singular arithmetic instructions mentioned above: they still need a field.

In the following enumeration of all the possible key sequences starting with A=, C<> or C= the class 2 instructions are marked with an asterix *.

	Key			Display	Instruction
a)	A=			A= _	
		B		A=B=C=0	A=B=C=0
		C		A=C _	A=C *
		-		A=A- _	
			B	A=A-B _	A=A-B *
			C	A=A-C _	A=A-C *
			1	A=A-1 _	A=A-1 *
		+		A=A+ _	
			B	A=A+B _	A=A+B *
			C	A=A+C _	A=A+C *
			1	A=A+1 _	A=A+1 *
		0		A=0 _	A=0 *
b)	C<>			C<> _	
		A		C<>A _	C<>A *
		B		C<>B _	C<>B *
		G		C<>G	C<>G
		M		C<>M	C<>M
		N		C<>N	C<>N
		S		C<>ST	C<>ST

	Push	See	Comment
e.g.			
	A=	A=	key in A=A-1 S&X
	-	A=A- _	
	1	A=A-1 _	
	EEX (S&X)	A=A-1 S&X	and A=A-1 S&X is programmed

c

key 1	key 2	key 3	Display	Instruction
C=			C= _	
	A		C=A _	
		-	C=A-C _	C=A-C *
		+	C=C OR A	C=C OR A
		*	C=C AND A	C=C AND A
	B		C=B _	C=B *
	G		C=G	C=G
	K		C=KEY	C=KEY
	M		C=M	C=M
	N		C=N	C=N
	CHS		C=0-C _	C=0-C *
	S		C=ST	C=ST
	-		C=C- _	
		-	C=-C-1 _	C=-C-1 *
		1	C=C-1 _	C=C-1 *
	+		C=C+ _	
		A	C=C+A _	C=C+A *
		C	C=C+C _	C=C+C *
		1	C=C+1 _	C=C+1 *
	0		C=0 _	C=0 *

	Push	See	Comment
e.g.			
	C=	C= _	key in C=-C-1 P-Q
	-	C=C- _	
	-	C=-C-1 _	
	- (P-Q)	C=-C-1 P-Q	finished

In appendix C you can find all instructions in alphabetical order, and there sequence of keying them in.

IV A4.7 Jump subroutines

There are 2 kinds of jump subroutines possible in machine language :
a : ?NC XQ and ?C XQ
b : GOSUB (this is a relocatable XQ, actually of the form
?NC XQ GOSUB/ GOSUB0/ GOSUB1/ GOSUB2/ GOSUB3 <location in 1K>)

a. The sequence for a ?NC XQ and ?C XQ start with

- pushing XEQ key, XEQ ____ will appear in display
- if you want a ?C XQ, pushing [], XEQ C ____ will appear.

You see that XEQ asks for a 4-digit hexadecimal address, however a label name can be keyed in too!

So after you have keyed in XEQ or XEQ [], you can choose between :

- keying in a a 4 digit hexadecimal number, or
- pushing ALPHA key, label name, and alpha key again

Then, a ?NC XQ or ?C XQ to the specified address or label will be programmed over the next two words

e.g.

Push	See	Comment
XEQ (XEQ)	XEQ ____	want 4 digits
1 2 3 4	?NC XQ 1234	finished
XEQ []	XEQ C ____	
"CHANGE"	?C XQ CHANGE	"change" was at 0004 so actually a ?C XQ 0004 is written in MLDL RAM.
alpha key		

b. The sequence for a GOSUB starts with pushing the XEQ key :
XEQ ____ will appear.
After that, one can choose between :
- keying in a 4-digit hexadecimal address, of which the first digit is equal to the first digit of the APC. This first digit must be greater than 6.
- pushing label name of which the location is at the same page as the APC is located (in other words: the first digit of both addresses must be equal and the page must be > 6).

	Push	See	Comment
e.g.			
	CONT. A000 LBL "SAME4K"	LBL'SAME4K	do some preparing work
	CONT. A900 LBL "SAME1K"	LBL'SAME1K	
	CONT AA00	NOP	
	XEQ A123	GOSUB A123	continue in scratch area
	USER	AA01 349 I.	address = A123, so reloc
	SST	AA02 08C L:	you can check the
	SST	AA03 123 #	programmed words
	USER [] BST	GOSUB A123	
	XEQ "SAME1K"	GOSUB SAME1K	
	USER	AA04 379 9.	these two words form a
	SST	AA05 03C <	?NC XQ GOSUB (0FDE)
	USER [] BST	GOSUB SAME1K	
	XEQ "SAME4K"	GOSUB SAME4K	nothing is easier than this

You see that it depends on the address or label you key in whether a ?(N)C XQ or a GOSUB is programmed:

If before a relocatable address [] is hit, it won't matter to what will be programmed :

e.g.	XEQ [] "SAME1K"	GOSUB SAME1K	[] is ignored

If you key in a label name that not yet exists an error appears.

e.g.	XEQ "QXT"	NONEXISTENT	you're still in ASSM mode

IV A4.8 All jumps

The [] GTO key is the most flexible key on the ASSM-keyboard. All jumps (relative, absolute and relocatable) start with this key. If you push it, GTO ____ will appear in display. After that you can choose one of the following jumps

a JNC and JC
The relative jumps can be keyed in in two ways :

- [] GTO, GTO ____ will appear
- if it should be a JC, press [], GTO C ____ will appear
- [+] or [-] key, depending on the jump direction GTO + __ / GTO - __ / GTO C + __ / GTO C - __ will appear
- a two digit hexadecimal number in the range of 0 - 3F. After a GTO - __ or GTO C - __ you can also hit [4], which will be assembled as 40.

Push	See		Comment
e.g. [] GTO	GTO ____		standard prompt
[]	GTO C ____		carry
+	GTO C + __		positive direction
25	JC + 25	AA32	finished
[] GTO + 4	GTO + __		4 is ignored
3F	JNC + 3F	AA4D	
[] GTO - 4	JNC - 40	A9CF	
[] GTO - 01	JNC - 01	AA0F	

- press [] GTO
- if you want a JC, press [] again
- key in a 4-digit hexadecimal address, or alpha, label name that is assigned to an address, alpha where the address is within relative jump distance (-40 to + 3F)

Push	See	Comment
e.g. LBL'AB	LBL'AB	create a label
CONT. AA00	NOP	
[] GTO []		
"AB"	JC+ AB	right distance is computed
CONT AA4F	NOP	
[] GTO "AB"	JNC - AB	
USER	AA50 203 C	203 is JNC - 40
USER		

b GOTO ADR and GOTO KEY

These two miscelaneous instructions must be keyed in like this:

GOTO ADR : [] GTO M (M is same key as GTO)
GOTO KEY : [] GTO K

Push	See	Comment
e.g. [] GTO	GTO ____	
K	GOTO KEY	
[] GTO M	GOTO ADR	adr field is part of mantissa

c ?NC GO, ?C GO, GOTO (relocatable GO)

The same conditions are in force here as in 4.7

Key in [] GTO ([]) 4-digit address / label, and ASSM checks whether a J(N)C , ?(N)C GO or GOTO to that addres or label should be programmed.

	Push	See	Comment
e.g.	[] GTO 0000	?NC GO 0000	
	[] GTO [] 0004	?C GO CHANGE	CHANGE still isn't purged
	[] GTO "SAME1K"	GOTO SAME1K	
	[] GTO "SAME4K"	GOTO SAME4K	
	[] GTO "AB"	GOTO AB	AB is more than 40 steps back
	[] GTO "CHANGE"	?NC GO CHANGE	CHANGE is not located on page A so a ?NC GO is programmed.

d JNC or JC to nonexisting labels

It is possible to key in a not yet existing label name after [] GTO. Instead of an error "NONEXISTENT" a J(N)C ? {label name} will be shown in the display.

The moment that the suiting label is keyed in, the right jumpdistance is computed and a real J(N)C +/- {label name} is programmed at the location where the jump was keyed in.

	Push	See	Comment
e.g.	[] GTO "XNOTYT"	JNC ? XNOTYT	XNOTYT is not known yet, so a ? is shown.
	[] GTO "XNTYT2"	JNC ? XNTYT2	other jump
	CONT AA80	NOP	
	LBL "XNOTYT"	LBL' XNOTYT	XNOTYT is assigned to AA80
	CONT AA24	NOP	
	LBL "XNTYT2"	LBL' XNTYT2	
	CONT AA60	NC GO CHANGE	
	SST	JNC + XNOTYT	this line is programmed
	SST	JNC - XNTYT2	

Just JNC's and JC's can be programmed in this way, because ASSM can not know whether it should reserve one, two or three words (for respectively J(N)C, ?(N)C GO and GOTO) for the jump. That is why it can happen that the message JUMP TOO FAR will appear in the display if you key in a label: JNC's may not go further than 3F forwards or 40 backwards.

	Push	See	Comment
e.g.	CONT AA00	NOP	well-known area!
	[] GTO "ABC"	JNC ? ABC	at AA01
	[] GTO [] "DEF"	JC ? DEF	
	[] GTO "DEF"	JNC ? DEF	
	[] GTO [] "ABC"	JC ? ABC	at AA04
	CONT AA40	NOP	
	LBL "DEF"	LBL'DEF	DEF is assigned at AA40
	[] NOP	NOP	now NOP is programmed at AA40 (remember: label is viewed before step)
	[] NOP	NOP	
	LBL "ABC"	JUMP TOO FAR	APC is AA41 now AA41 - AA01 = 40 too far!
	CONT AA01	NOP	a NOP is programmed instead of a JNC +40
	SST	JC +DEF	all pending jumps were programmed by keying in LBL"DEF"
	SST	JNC +DEF	
	SST	JC +ABC	also with ABC, exept that there was at least one JUMP TOO FAR

If there are still pending J(N)C ? 's and you try to exit ASSM mode, an error message ? LBL ' {still not yet existing label name} will appear in display to remind you of the fact that your program is not finished yet. However, you did leave ASSM mode.

	Push	See	Comment
e.g.	[] GTO "HIJ"	JNC ? HIJ	create a jump to nonexistent label
	<- (cont)	CONT. ____	still in ASSM mode
	<-	?LBL'HIJ	you left ASSM mode
	<-	{X-register}	

The JNC ? and JC ? 's actually are NOP's.
During execution they do nothing

	Push	See	Comment
e.g.	ASSM	JNC ?'HIJ	last viewed step
	USER	AA04 000 @	000 is code for NOP

IV A 5 Messages in ASSM

The following error messages may occur.

- NONEXISTENT : You keyed in a nonexistent label name after a XEQ, [] XEQ, or CONT. (<-)
- NO WRITE : You tried to write in ROM.

	Push	See	Comment
e.g.	USER	JNC ? HIJ	last viewed step
	cont 0000	?NC GO 0180	
	N=C	NO WRITE	you can't write in ROM!

- NULL : you held the last key of a sequence too long. (any key sequence, exept for BEGIN ____)
- PACKING-TRY AGAIN :
  - you tried to execute ASSM for the first time without free RAM (i.e. 00 REG 00 in PRGM mode).
  - you tried to key in a label without free RAM.
  - you tried to program a jump to a nonexistent label without free RAM.
- ?LBL'{name} : still pending jumps to nonexistent labels.
- JUMP TOO FAR : a new label caused a JNC to that label to be too far.
- TRY AGAIN : this error occurs in the theoretical case you try to create a 255st label or 255st jump to a not yet existing label.

IV_B BUF>REG and REG>BUF

The function ASSM must keep a list of assigned labels and jump-to nonexistent label locations. For this ASSM uses two I/O buffers.

An I/O buffer is a reserved part of user code memory, located directly above the key assignments, which may expand or decrease. A buffer exist of one buffer header, in which the buffer identity (a number between 1 and 14) and the length of the buffer (in registers, header included) are stored.

For execution of ASSM, the buffer with identity 1 should always exist, for important ASSM variables are saved in a part of the header (such as the APC), and a list of labels is saved in this buffer, using one register per label.

Why all this information in this chapter?

Since all buffers will be purged by turning on the HP41, unless special ROM's, as DAVID-ASSEM does, keep one or more of them alive, all your label assignments would be purged if you would turn on the calculator without DAVID-ASSEM plugged in.

This is why BUF>REG and REG>BUF have been incorporated. These functions provide that buffers could be copied into normal user register; these could be copied with the normal functions WRTX (for the cardreader) or WRTRX (for the mass storage) onto magnetic cards respectively tape. Then, if you need the buffers again, you can read them from the magnetic medium into the registers, and then with REG>BUF a buffer with the original contents will be created.

IV_B1 BUF>REG

To copy a buffer with ID# (identity number) p, p must be in X and the size should be minimum n, where n is the buffer length. Then BUF>REG must be executed, after execution n will be in the format: 0. {3-digit n-1}, so that immediate WRTX or WRTRX can be executed.

e.g. Suppose you want to save the eight labels you assigned in chapter IV A (CHANGE, SAME1K, SAME4K, AB, XNOTYT, XNTYT2, ABC, DEF) on a magnetic card.

(Size must be at least 9)

Do	See	Comment
1	1_	buffer ID#1 is used for
XEQ "BUF>REG"	0,008	labels assumed you have FIX 3 the buffer is 8 labels + 1 head = 9 registers long 9 - 1
XEQ "WRTX"	etc.	= 8 labels are saved on card.

IV_B1.1_Error_messages:

NONEXISTENT - the buffer with ID# specified by X does not exist
- X > = 1000 (guess what routine is used)
- the size is too small

DATA ERROR - X = 0 or 15 <= X <= 999 (too small or too big X)

ALPHA DATA - X is of type ALPHA DATA

IV_B1.2_Warning

If a buffer is copied into user registers, you should not recall the contents of a register with RCL, because this could affect the contents of the register.

IV_B2_REG>BUF

This function can be used for two different purposes:

a. To copy buffer contents in registers into a real buffer. No parameters are used nor results are given : REG>BUF gets its information out of REG 00, in which the "header" should be located. Therefore, R00 should have the following format :

```
 ! 13 ! 12 ! 10    9 !  8    7    6    5    4    3    2    1    0 !
 !    !    !         !                                            !
 ! ID#! ID#! buffer- !              buffer data                   !
           ! length  !
```

Before REG>BUF starts copying, first all existing buffers with the same ID# as in R00 is specified are purged (mostly it is "the buffer", for more than 1 buffer with the same ID# is not allowed actually), in order to assure there is maximum 1 buffer with that specified ID#.

e.g. Turn calculator off, plugg DAVID-ASSEM out, press ON twice plugg in DAVID-ASSEM again, turn calculator on again :
Now both buffers #1 and 2 (for repectively assigned labels and nonexistent labels) are purged.

Do	See	Comment
ASSM	BEGIN ____	there is no APC known yet.
<-	{X-register}	this was to show you that both buffers are cleared.
XEQ "REG>BUF"		now buffer 1 (labels & APC) must be created:
ASSM	?NC GO 0180	APC was 0000 when BUF>REG was executed.
cont. "ABC"	LBL'ABC	label is back again!
<- <-		exit.

IV B2.1 Error mesages

NONEXISTENT: there are not enough free registers available to create the buffer with length, specified in R00.

IV B2.2 Warning

Be sure that R00 contains the right information. If you've done no RCL 00's or anything else special, you may assume R00 is good; but if you have created R00 using CODE routines and NonNormalized STOre routines, you must double check its contents. A crash could be the result of a wrong use.

b. A special code in R00 is recognised trough REG>BUF as the function : clear both buffer ID#1 and buffer ID#2. This special code is very simpel : it is the value zero. 0 can be used for this purpose since ID#0 does not exist, nor a buffer length 00.

	Do	Comment
e.g.	0 STO 00	special code in R00
	XEQ "REG>BUF"	now buffer ID#1 and ID#2 have been purged
	ASSM	BEGIN ____; APC is not known any more
	<-	exit.

IV C BEG/END and DISTOA.

With these two functions you can make printed listings of your machine-language programs, in which labels will make the listings more readable.

With BEG/END you can define the beginning and the ending address, where they must be the last 8 characters in ALPHA, the first 4 characters the 4-digit beginning address, the last 4 characters the ending address.

With DISTOA you can move the address & instruction string as it is shown in display during ASSM to the alpha register, compute the location of the next step, and test whether the end address is reached.

BEG/END sets the APC to the specified beginning address, and execution of DISTOA causes the APC to point to the next step, and check APC >= END, where the next user code step will be skipped if the equasion is not true.

Knowing these facts, you can write the following user code program that will print a certain part of machine-language program :

```
01   LBL'PRINT
02   'BEGIN/END?
03   AON
04   STOP              prompt for begin and end
05   AOFF
06   BEG/END           define begin and end
07   LBL 01            begin loop
08   DISTOA            put line in ALPHA
09   GTO 03            if reached end
10   PRA               print address & line
11   GTO 01
12   LBL 03
13   BEEP              signal
14   PRA               this may be added if the end address itself
                       should be printed
15   END
```

All different kinds of loop counters may be inserted between steps 10 and 11, such as form feed counters etc.

IF you write the function FNC TO A, as listed in appendix E, using ASSM, and you have the rom MNFR-LBLS, your listings cannot be beaten by other disassemblers!

There are a few characters in ASSM mode which don't exist on the printer.

These are translated this way :

Discription	Display code	Printer chr.	Printer code
west goose	02C	<	03C
east goose	02E	>	03E
starburst	03A	[]	01F

If flag 27 (the user flag) is cleared, DISTOA won't disassemble, but put just the rom words display character behind the address into ALPHA, and the APC will be incremented by 1.

During DISTOA's execution the display is turned off, because actually the characters are read from display into alpha, which also has the effect that just 12 characters after the address & spacing can be placed (excluding ":",";" and ","), but it doesn't often happen that a line is more than 12 characters long.

This only occurs when you have a message string that contains 11 characters. Since the message string is included in two quotes , " , the resulting string is 13 characters long. The right hand quote will be cut from the listing, this also happens in normal ASSM mode

V Technical Details.

In this chapter we will go further into the operation of the functions

V 1 Register use with ASSM

ASSM uses just the scratch areas of the stack registers. However, this is not enough to remember all information during light or deep sleep. Therefore the 5 free bytes in the header of the buffer with ID#1 (in which the labels are stored) are used to save information.

An effect of this is that this buffer always exists during execution of ASSM. This is why ASSM will cause the message "PACKING-TRY AGAIN", to appear if there are no free registers of user RAM and the buffer with ID#1 does not exist yet: At least 1 register is needed for the header of the buffer.

ASSM will prompt for a beginning address with BEGIN ____ if for what ever reason the buffer #1 is purged, because the APC is stored in the header.

ASSM uses the following scratch areas :

1. REG 8(P) [13:6]
2. REG 9(Q)
3. REG 15(e) [4:3]
4. Buffer ID#1, header [9:0]

REG 8(P) [13:6]

Both in [13:0] and in [9:6] return adresses used by ASSM are saved during a light sleep, because in machine language you should call the mainframe subroutine NEXT (at 0E50) without any subroutine level on the CPU return stack. ASSM often calls NEXT two levels deep.

REG 9(Q)

This register is used for temporary alpha scratch of label names when these are keyed in. However, it is not done in the same way as in the mainframe routines. The formatting will be discussed later.

REG 15(e)[4:3]

This part of the register is used automatically by calling NEXT. In here the CPU register G is saved, which is used by ASSM as a digit counter in cases that the <- key will clear the last keyed in character. When this is not the case, its value is set to 0 to avoid a bug in the key-information-flag-set PTEMP1, as HP calls it in the VASM listings.

Header of buffer with ID#1 [9:0]

a. Part [3:0] This is always the APC, the address on which the instruction is located that is viewed.

b. Part [7:4] This part is used for different purposes :

1. When CONT, GTO(C) or XEQ(C) is in display, NEXT is called three subroutine levels deep: In that case another return address is saved there.
2. If BEG/END has been executed, the ENDing address is saved there.

c. Part [9:8] This part is used for a special ASSM flagset, in which 8 flags are saved, we shall call them flags 7-0 They are used as follows :

flag 0 - scratch flag, used by keyparser
1 - " "
2 - if this flag is set, a SST or BST is auto repeating.

flags 3 and 4:
- if flag 3 is set, and 4 is cleared, a data word is viewed, except if you are in USER off mode.
- if flag 4 is set and 3 is cleared, a text string (after a ?NC XQ MESSL (at 07EF)) is viewed.
- if both flags are set, a selp instruction has been passed and still is in force.

flag 5 - This flag is not used by ASSM for historical reasons.
6 - This flag is set if a label is displayed LBL'{label name} ASSM decides from this flag whether the next step may be a label or not : if this flag is set, the next step is disassembled not to be a label.
7 - If this flag is set, the disassemble routine (at X3FA) acts like a normal subroutine. This flag is used by DISTOA

	DO	COMMENT
e.g.	0 STO 00 REG>BUF	clear buffers 1 and 2
	ASSM	see BEGIN ---- buffer #1 clear
	07F7	see HEX:010 P data word
	<- <- 1 BUF>REG	move header to reg 00
	RCL 00	This is allowed because first digit is 1: alpha text
	DECODE	we assume you'll have such a routine: see 11 01 08 X5 C2 07 F7

11 is the buffer ID number
01 is the length of the buffer
08 flag 3 is set
X5C2 return address for ASSM (X is page of DAVID-ASSEM)
07F7 Assem Program Counter

V_B_Format_of_a_label_in_a_buffer

A buffer register that is used for label storage will look like this

```
  13  : 12    11    10    9    8    7    6    5    4  :  3    2    1    0
      :                                               :
carry :         condensed 6-character string          :       address
```

With assigned labels [13] = 0, and [3:0] points to the address to which the label is assigned.

With nonexisting labels [13] says whether a JNC ? or JC ? is stored there. A 1 means JC ?, a 0 means JNC ?. [3:0] points to the address on which the JC ? or JNC ? is located.

Both with assigned and nonexisting labels part [12:4] is used to store a 6-character string. There are 64 characters possible, so one character takes 6 bits. A label is maximum 6 characters long, so 6*6 = 36 bits are needed. Since [12:4] is 9 nibles long (9*4 = 36 bits) it fits exactly.

The characters are stored in reverse order, right justified, so the first character is in [4] and bits 0 and 1 of [5], the second character in bits 2 and 3 of [5] and [6], etc. The character with value 0 is the end of string character. If this is missing, the string is 6 characters long.

For the characters the following table applies

upper 2 \ lower 4	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	end	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	P	Q	R	S	T	U	V	W	X	Y	Z	['	\	]'	^	_'
2		!'	"'	#'	$	%	&'	''	('	)'	:	+	<-'	-	->'	/
3	0	1	2	3	4	5	6	7	8	9	bs	,	<	=	>	?

The characters marked with ' cannot be keyed in from the keyboard.

e.g. If you have assigned the label "MESSL" to 07EF, somewhere in buffer #1 a register will have the value; 00 0C 4D 31 4D 07 EF, which is in bits

```
 13 |  12   11    10    9     8     7     6    5     4  |  3    2    1    0
    |          |           |         |         |         |
0000| 0000 00|00 1100| 0100 11|01 0011| 0001 01|00 1101| 0000 0111 1110 111
    |          |           |         |         |         |
zero| end str|    L    |    S    |    S    |    E    |    M    |  0    7    E    F
```

V C BST and CONT

As mentioned on page 18, a BST can not be defined correctly in machine language for the disassembler cannot know whether words are just data in a table, or instructions.

Therefore BST looks 7 steps back, then acts as if it is the beginning of an instruction, and computes every new step, until the original APC has been reached. The new APC will be the last but one computed step. This assures for 99% that the disassembler sorts itself out, but there are exeptions of course. More than 7 steps would cause a BST to last longer than would be nice. (compare a BST in USER code in a program that is more then hundred steps long). If you want to change this constant (7) you can find it in V E.

A CONT. (<-) (and also a BEGIN ____) does actually a APC := {address}, a BST and a SST to assure that an instruction is disassembled well. Therefore it could happen that you continue one word further than you typed in.

	PUSH	SEE	COMMENT
e.g.	ASSM <- 0001	?NC GO 01AD	try to continue in 2 word instruction.
	USER	0002 2B5 S:	it didn't work out
	USER		
	<- 2FF3	RTN	try to continue in text string
	USER	2EF5 3E0 ,	it didn't work out

V D Extensions of DAVID - ASSEM

Every time ASSM searches for a suiting label or a suiting adress, all plug-in ROM's on pages => 5 are scanned for a ROM with a XROM# 100. This is a too high XROM#, and therefore no functions can be incorporated in this ROM.

If such ROM is found, ASSM will jump somewhere in the ROM (at X080 or X0D3). At these locations extension routines may be located, and they may choose between letting ASSM search further for other XROM#100's and returning directly to ASSM.

Here the in and out conditions follow.

1 Given an address, search for a suiting label

in - the entry address is X080. If you would not use this entry, a RTN should be on that location.
- the address, with which a label must suit, is in CPU register N[3:0]
- the DSP is off, RAM is selected (not chip 0), G[0] is page of the ROM itself, flag 9 is cleared.

in & out - M(CPU) may not be affected. APC is in M[6:3], ASSM status is in M[1:0]

out - if no suiting label is found, a simple RTN is enough (flag 9 must be cleared) to cause ASSM to look further for XROM#100's.
- if a label is found, one return address should be skipped by POP or better by XQ->GO, flag 9 must be set, and the label should be in condensed form (see B) in C[12:4], and RTN will return to ASSM, and will show the label.

2. Given a label, search for suiting address

in - the entry address is X0D3. If you would not use this entry, a RTN should be on that location.
- The label is in condensed format (see B) in CPU reg. N[12:4], Q=12, P is selected.
- The DSP is on, RAM is selected (not chip 0), G[0] is page of the ROM itself, flag 9 is cleared.

in & out - M(CPU) may not be affected. APC is in M[6:3], ASSM status is in M[1:0].

out - if no suiting address is found, a RTN is enough. (flag 9 must be cleared)
- if an address is found, one return address shoud be skipped by POP or XQ->GO, flag 9 must be set, and the address must be in B[3:0].

Both in case 1. and 2. one must keep two subroutine levels on the stack, G and M may not be used.

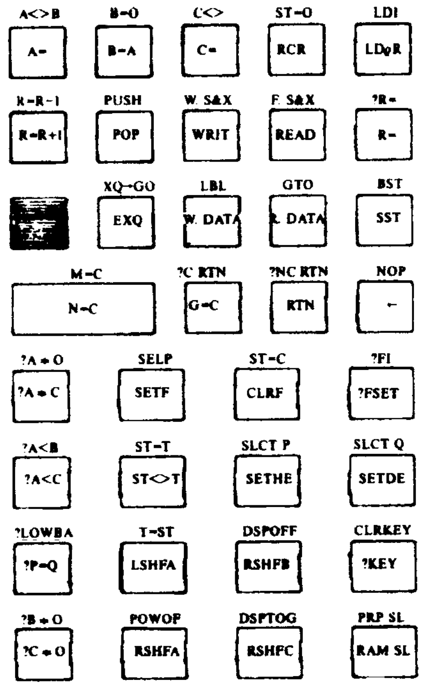
This all may look a bit complex, but there is already one 4K ROM that adds all mainframe entries as listed in HP's VASM listings called MNFR-LBLS. These are c. 750 labels, of which the search routines take in case 1. less than .1 second, and in case 2. maximum 1.5 second with an average of .4 second. More about this rom in appendix G

V_E_Important_Addresses

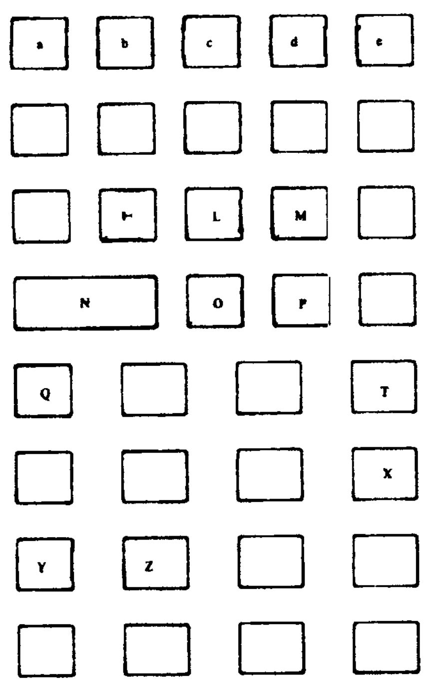
If you find a few constants not good, you may change them for you own use.

X624 : Autorepeat constant. Normally 3FF. This constant determines how long a step is viewed in autorepeat mode.
X64C : wait-to-autorepeat constant. Normally 3FF. This constant determines how how long SST/SST should be pressed before they will go autorepeating.
XEA0 : BST constant. Normally 007. This constant determines where is started when the disassembler tries to sort itself out during a BST. This value is the number of words back.
XC64 : XROM constant. Normally 100. This is the XROM# for which is searched in other 4K ROM's. You can change this to a XROM < 40 if you want also functions in your extension(s).
X974 : relocatable page minimum. Normally LD@R 7. This instruction determines from which page relocatable goto's and GOSUB's are possible. (normally from page 7)
X000 : the XROM#. Normally 002
XFFF : check sum. Normally 35F

Appendix A Keyboard definition figures

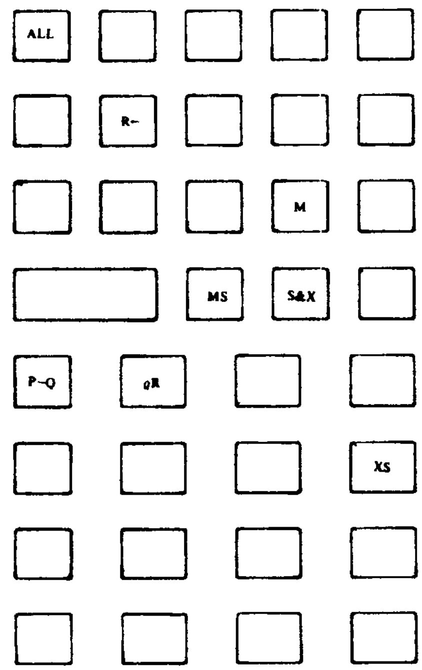


keyboard 1

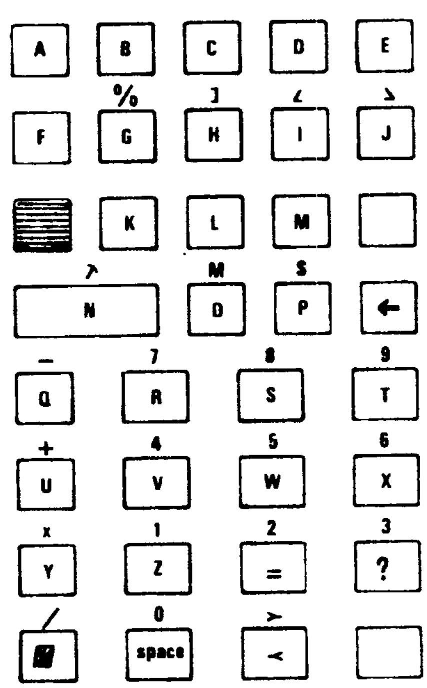


keyboard 2

Appendix A Keyboard definition figures



keyboard 3



alpha on

Appendix B Instructionset of the HP41 CPU

The HP41 CPU has three main arithmetic registers: A,B and C. These are 56 bits long (14 nibbles) and instructions can operate in various "fields" of the register.

```
nibbles  | 13 | 12  11  10   9   8   7   6   5   4   3 | 2  | 1    0 |
         |    |                                         |    |        |
         |    |                                         | XS |        |
         |    |                  ALL                    |<-->|        |
         |<---+-----------------------------------------+----+------->|
         | MS |                   M                     |    S & X    |
         |<-->|<--------------------------------------->|<----------->|
```

ALL : The whole register
M   : Mantissa
MS : Mantissa Sign
XS : eXponent Sign
S&X : eXponent and Sign of exponent

@R : At specified pointer
R<- : from digit R to digit 0
PQ : Between P and Q

There are two pointers P and Q, of which the value is 0-13. One of them is selected at the time (through slct p or slct q), the selected pointer is called R. These are three extra fields, which depend on the value of the pointer), R<- (up to R, from digit R to digit 0) and P-Q (between pointer P and Q, Q must be greater than P).

There is a register G, 8 bits long, that may be copied to or from or exchanged with the nibbles R and R+1 of register C. (R<=12). There are 14 flags, 0-13, of which flags 0-7 are located in the 8-bits ST (status) register, and there is a 8-bits TONE register T, of which the contents floats every machine cycle through a speaker.

Then there are two auxilary storage registers, M and N, which can operate only in the field ALL. They are 56 bits long.

There is a 16-bit program counter, which addresses the machine language, and a KEY register of 8-bits, which is loaded when a key is pressed. The returnstack is 4 addresses long and is situated in the CPU itself.

The CPU may be in HEX or DEC mode. In the latter mode the nibbles act as if they can have a value from 0 to 9.

The USER-code RAM is selected by C[s&x] through RAM SLCT, and can be written or read through WRITE DATA or READ DATA. If chip 0 is selected (RAM address 000 to 00F) the 16 stack registers may be addressed by WRIT and READ 0 to 15.

Peripherals (such as display, card reader, printer) may be selected by C[s&x] through PRPH select or by SELP (see page 19).

The mnemonics are a kind of BASIC structure.

Arithmetic instructions (operate on a specified field)

A=0	C=B	C=C+1	?A<B
B=0	A=A+1	C=C+A	?A#C
C=0	A=A+B	C=A-C	?A#0
A<>B	A=A+C	C=0-C	RSHFA
B=A	A=A-1	C=-C-1	RSHFB
A<>C	A=A-B	?B#0	RSHFC
A=C	A=A-C	?C#0	LSHFA
C<>B	C=C+C	?A<C	

CLRF, SETF, ?FSET, ?R=. ?FI (peripheral flag set?) , RCR (rotate right) have a parameter 0-13.

LD@R (load C at R) and SELP (select peripheral) have a parameter 0-F.

WRIT and READ have a parameter 0-15, called
0(T), 1(Z), 2(Y), 3(X), 4(L), 5(M), 6(N), 7(O), 8(P), 9(Q), 10(|-), 11(a), 12(b), 13(c), 14(d), 15(e).

Jumps:
There are two classes jumps:

a. JNC (jump if no carry) and JC (jump if carry). These instructions provide to jump relative 3F in positive direction or 40 in negative direction.
b. ?NC GO and ?C GO. These instructions provide to jump to an absolute 16 bits address.

?NC XQ and ?C XQ are jump-subroutine instructions to absolute addresses. (remember the return stack is just 4 addresses long).

Miscelaneous instructions:

ST=0	C=G	ST=T	POWOFF
CLRKEY	C<>G	ST<>T	SLCT P
?KEY	C=M	ST=C	SLCT Q
R=R-1	M=C	C=ST	?P=Q
R=R+1	C<>M	ST<>C	?LOWBAT
G=C	T=ST	XQ->GO	A=B=C=0
GOTO ADR (C[6:3])	?C RTN	PUSH (C[6:3])	
C=KEY	?NC RTN	POP (C[6:3])	
SETHEX	RTN	GOTO KEY	
SETDEC	N=C	RAM SLCT	
DSPOFF	C=N	WRITE DATA	
DSPTOG	C<>N	READ DATA	
FETCH S&X	C=C or A	PRPH SLCT	
WRIT S&X (for MLDL)	C=C and A		

Note: various arithmetic and all test instructions may set the carry flag. This flag keeps set only one machine cycle, so a jump dependent on this flag must be immediate after the arithmetic or test instruction, otherwise the carryflag will always be cleared.

The HP41 CPU cannot execute all combinations of instructions correctly. Those that are located are listed in appendix E.

Appendix C Instructions and their keysequences

All possible instructions are listed below with their keysequences (in ASSM and normal keyboard), The parameter sort, and the page where information can be found. The parameters are :

F	:	field
4K	:	4K hex addres
L	:	label
-	:	nothing
d3	:	decimal 0-13
d5	:	decimal 0-15
STK	:	stack reg name
H	:	hexadecimal digit
+/-	:	plus or minus hex

Instruction	Key sequence		Parameter	Page
?A#0	[] ?A#0	(X=Y?)	F	27
?A#C	?A#C	(-)	F	27
?A<B	[] ?A<B	(X<=Y)	F	27
?A<C	?A<C	(+)	F	27
?B#0	[] ?B#0	(X=0?)	F	27
?C GO	[] GTO []	(GTO [])	4H / L	32
?C RTN	[] ?C RTN	(ISG)	-	24
?C XQ	XEQ []	(XEQ [])	4H / L	30
?C#0	?C#0	(/)	F	27
?FI	[] FI	(FS?)	d3	25
?FSET	?FSET	(9)	d3	25
?KEY	?KEY	(3)	-	24
?LOWBAT	[] LOWBAT	(X>Y?)	-	24
?NC GO	[] GTO	(GTO)	4H / L	32
?NC RTN	[] ?NC RTN	(RTN)	-	24
?NC XQ	XEQ	(XEQ)	4H / L	30
?P=Q	?P=Q	(*)	-	24
?R=	[] ?R=	(TAN^{-1})	d3	25
A<>B	[] A<>B	(Σ-)	F	27
A<>C	[] C<> A	(X^2 Σ+)	F	28
A=0	A= 0	(Σ+ 0)	F	28
A=A+1	A= + 1	(Σ+ + 1)	F	28
A=A+B	A= + B	(Σ+ + 1/X)	F	28
A=A+C	A= + C	(Σ+ + $\sqrt{X}$)	F	28
A=A-1	A= - 1	(Σ+ - 1)	F	28
A=A-B	A= - B	(Σ+ - 1/X)	F	28
A=A-C	A= - C	(Σ+ - $\sqrt{X}$)	F	28
A=B=C=0	A= B	(Σ+ 1/X)	-	28
A=C	A= C	(Σ+ $\sqrt{X}$)	F	28
B<>A	[] A<>B	(Σ-)	F	27
B<>C	[] C<> B	(X^2 1/X)	F	28
B=0	[] B=0	(Y^X)	F	27
B=A	B=A	(1/X)	F	27
C<>A	[] C<> A	(X^2 Σ+)	F	28
C<>B	[] C<> B	(X^2 1/X)	F	28
C<>G	[] C<> G	(X^2 R↓)	-	28
C<>M	[] C<> M	(X^2 RCL)	-	28
C<>N	[] C<> N	(X^2 ENTER)	-	28
C<>ST	[] C<> S	(X^2 8)	-	28
C=-C-1	C= - -	($\sqrt{x}$ - -)	F	29
C=0	C= 0	($\sqrt{x}$ 0)	F	29
C=0-C	C= CHS	($\sqrt{x}$ CHS)	F	29
C=A-C	C= A -	($\sqrt{x}$ Σ+ -)	F	29

Instruction	Key sequence		Parameter	Page
C=A and C	C= A *	($\sqrt{x}$ $\Sigma+$ *)	-	29
C=C or A	C= A +	($\sqrt{x}$ $\Sigma+$ +)	-	29
C=B	C= B	($\sqrt{x}$ 1/x)	F	29
C=C+1	C= + 1	($\sqrt{x}$ + 1)	F	29
C=C+A	C= + A	($\sqrt{x}$ + $\Sigma+$)	F	29
C=C+C	C= + C	($\sqrt{x}$ + $\sqrt{x}$)	F	29
C=C-1	C= - 1	($\sqrt{x}$ - 1)	F	29
C=G	C= G	($\sqrt{x}$ R↓)	-	29
C=KEY	C= K	($\sqrt{x}$ XEQ)	-	29
C=M	C= M	($\sqrt{x}$ RCL)	-	29
C=N	C= N	($\sqrt{x}$ ENTER)	-	29
C=ST	C= S	($\sqrt{x}$ 8)	-	29
CLRF	CLRF	(8)	d3	25
CLRKEY	[] CLRKEY	(ENG)	-	24
DSPOFF	[] DSPOFF	(SCI)	-	24
DSPTOG	[] DSPTOG	(LASTX)	-	24
FETCH S&X	[] FETCHS&X	(COS^{-1})	-	24
G=C	G=C	(CHS)	-	24
GOSUB	XEQ	(XEQ)	4H / L	30
GOTO	[] GTO	(GTO)	4h / L	32
GOTO ADR	[] GTO M	(GTO RCL)	-	32
GOTO KEY	[] GTO K	(GTO XEQ)	-	32
JC	[] GTO []	(GTO [])	+/- / 4H / L	31
JNC	[] GTO	(GTO)	+/- / 4H / L	31
LD@R	LD@R	(LN)	H	25
LDI	[] LDI	(e^x)	-	24
LSHFA	LSHFA	(0)	F	27
M=C	[] M=C	(CAT)	-	24
N=C	N=C	(ENTER)	-	24
NOP	[] NOP	(CLX/A)	-	24
POP	POP	(R↓)	-	24
POWOFF	[] POWOFF	(PI)	-	24
PRPH SLCT	[] PRPHSLCT	(VIEW)	-	24
PUSH	[] PUSH	(%)	-	24
R=	R=	(TAN)	d3	25
R=R+1	R=R+1	(X<>Y)	-	24
R=R-1	[] R=R-1	(CLΣ)	-	24
RAM SLCT	RAMSLCT	(R/S)	-	24
RCR	RCR	(LOG)	d3	25
READ	READ	(COS)	d5 / STK	26
READ DATA	READDATA	(RCL)	-	24
RSHFA	RSHFA	(0)	F	27
RSHFB	RSHFB	(2)	F	27
RSHFC	RSHFC	(.)	F	27
RTN	RTN	(EEX)	-	24
SELP	[] SELP	(SF)	H	25
SETDEC	SETDEC	(6)	-	24

Instruction	Key sequence		Parameter	Page
SETF	SETF	(7)	d3	25
SETHEX	SETHEX	(5)	-	24
SLCT P	[] SLCTP	(P-R)	-	24
SLCT Q	[] SLCTQ	(R-P)	-	24
ST<>T	ST<>T	(4)	-	24
ST=0	[] ST=0	(10)	-	24
ST=C	[] ST=C	(CF)	-	24
ST=T	[] ST=T	(BEEP)	-	24
T=ST	[] T=ST	(FIX)	-	24
WRIT	WRIT	(SIN)	d5 / STK	26
WRIT S&X	[] WRITS&X	(SIN)	-	24
WRITE DATA	WIRTEDATA	(STO)	-	24
XQ->GO	[] XQ->GO	(ASN)	-	24

Appendix D Error messages

The following Error messages can occur using DAVID-ASSEM:

1.ALPHA DATA: - BUF>REG X is of type ALPHA

2.DATA ERROR: - BUF>REG: x=0 or x>=15

3.JUMP TOO FAR: - ASSM mode: At least one jump to the label you keyed in cannot be realized. Label is located too far.

4.NONEXISTENT
- one of functions: DAVID-ASSEM is not plugged in
- ASSM mode: the label you keyed in after CONT or XEQ (C) does not exist.
- BUF>REG: the SIZE is to small to contain the entire buffer
- the buffer with ID# X does not exist
- DISTOA: - buffer #1 does not exist

5.NO WRITE: - ASSM mode: you tried to write in ROM or your MLDL is wrong.

6.PACKING TRY AGAIN
- ASSM: buffer #1 does not exist yet and there is no room to create a header. (00 reg 00).
- ASSM mode: there is no room to save a label jump to nonexistent label.
- REG>BUF: there is no room enough to create an entire buffer in the i/o area.
- BEG/END: buffer with ID#1 does not exist yet and there is no room to create a header.

7.TRY AGAIN - you wanted to key in a 255st label or jump to a nonexistent label.

8.? LBL {name} - ASSM: you still have not completed your machine language program. You forgot to key in this label.

Appendix_E_Example_program

This program will show you how to use labels.

The function itself, FNCTOA is very useful in programs for listings of machine language. You can change parts of it to make it suitable for your own demands.

First, append the function of the FAT of your RAM page, using the USER off mode. CONT.X001, see what is written there do [] BST and increment the word on X001. Add the FAT address and continue 7 words before that address, still in user off mode. Then key in

```
 081  A    last char of functionname
 00F  O
 014  T
 003  C
 00E  N
 006  F  function name: FNCTOA
USER  A=0  (S&X)  turn on assembler
```

For more information about the structure of the FAT and the ROM itself see your manual of the ERAMCO SYSTEMS MLDL-box.

You will need ERRNE, DOSKP, CLA, APNDNW. If you don't have the rom MNFR-LBLS you could first assign these 4 labels to the adresses respectively 02E0, 1631, 10D1, 2D14. This will make programming easier and nicer, but of course you can use the addresses themselves.

```
LABEL   INSTRUCTION             COMMENT

        READ        13(c)       first search buffer #1
        C<>B        S&X         put up chainhead in B[S&X]
        LDI
        0BF
        R= 12
        A=0         @R
        A=A+1       @R          compare buffer ID# in A[12]
        A=C         S&X         ram start address in A[S&X]
NXT     A=A+1       S&X         point to next ram register
NXTBUF  ?A<B        S&X         reached chainhead ?
ERRNEA  ?NC GO      ERRNE       yes, say nonexistent
        A<>C        S&X
        A=C         S&X         copy A[S&X] to C[S&X]
        RAM SLCT
        READ DATA               get register
        ?C#0        ALL         empty ? ( reached free memory ? )
        JNC-        ERRNEA      yes
        C=C+1       MS          key assignment
        JC-         NXT         yes
        ?A#C        @R          right buffer
        JNC+        FOUND       yes, header is in C
        RCR 10                  no, move buffer length to C[3:0]
        C=0         XS
        A=A+C       S&X         skip this buffer
        JNC-        NXTBUF
FOUND   RCR 11                  APC to C[6:3]
        C=C+1       M           for compare later
        M=C                     save APC in M
        A=0         ALL
        R= 6
        A=C         @R          copy page of APC
        LD@R 5                  set up compare value
        SLCT P
        R= 3
        SLCT Q
        R= 6                    put up address field, set R= 6
        ?A<C        @R          page < 5
        JC+         DOSKPA      yes, say not found
        C=0         P-Q
        C=C-1       P-Q         put FFFF in B
        B=A         P-Q         and FAT counter in C
        C<>B        P-Q         B is lowest function address > APC
        C=C+1       P-Q
        FETCH       S&X         get # of functions
        A=C         S&X         save in A[S&X]
LOOP    A=A-1       S&X         out of functions ?
        JC+         LOWEST      yes
        C=C+1       P-Q         increment FAT counter
        FETCH       S&X         get word
        ?C#0        XS          user code function ?
```

```
        JNC+        03          no
        C=C+1       P-Q         increment FAT counter
        JNC-        LOOP        try next FAT address
        RCR 12
        A=C         XS          save 3rd digit in A[XS]
        RCR 2
        C=C+1       P-Q         get 2nd word
        FETCH       S&X         C[XS] is always 0
        A<>C        XS          put 0 in A[XS], add 3rd digit
        RCR 11                  move to C[5:3]
        A=C         P-Q         copy address
        RCR 3
        A=C         @R          add page
        C<>M                    get APC+1
        ?A<C        P-Q         APC <= FNC address
        JNC+        03          yes
        C<>M                    no, put back, try next function
        JNC-        LOOP
        C<>M                    put APC+1 back
        ?A<B        P-Q         FNC address < lowest as yet ?
        JNC-        LOOP        no
        B=A         P-Q         make B lowest
        JNC-        LOOP
LOWEST  C=B         P-Q
        C=C+1       P-Q         still FFFF ?
        JC+         DOSKPA      yes, not found, do skip
        C=C-1       P-Q         restore address
LOOP1   C=C-1       P-Q         compute begin of FNC name
        FETCH       S&X
        ST=C
        ?FSET 7                 last character ?
        JNC-        LOOP1       no
        A=C         P-Q         save in A
        C=M                     get APC+1
        ?A<C        P-Q         within function name ?
DOSKPA  ?NC GO      DOSKP       no, do a skip
        READ DATA               get header #1 again
        RCR 11
        C=B         P-Q         copy new address to APC
        RCR 3                   APC = start address of function
        WRITE DATA
        C=0         S&X
        RAM SLCT                select chip 0
        ?NC XQ      CLA         clear alpha
        R= 9                    leaves C = 0
        LD@R 4                  put string "FNC: " in alpha
        LD@R 6
        LD@R 4
        LD@R E
        LD@R 4
        LD@R 3
        LD@R 3
        LD@R A
        LD@R 2                  leaves R= 0
```

```
        WRIT        S(M)
LOOP2   C=B         M           get function address
        C=C-1       M           decrement
        FETCH       S&X         get display character
        C<>B        M           put address in B again
        ST=C                    save character in ST
        ?FSET 6                 special character ?
        JNC+        NOSPCL      no
        RCR 1                   save collumn in C[MS]
        LDI
        2C0                     special character table is at 2C00
        RCR 10                  move to C[6:3]
        FETCH       S&X         get ascii byte
        JNC+        APPND
NOSPCL  ?FSET 5                 row 0-1
        JC+         02          no
        SETF 6                  make row 4-5
        CLRF 7                  clear last character bit
        C<>ST                   save last char. bit in ST and char. in C[1:0]
APPND   G=C                     R= 0 left by loading constants
        ?NC XQ      APNDNW      append ascii character to alpha
        ?FSET 7                 last character ?
        JNC-        LOOP2       no, do next character
        RTN                     finished, don't skip
```

The use of FNCTOA.

FNCTOA checks whether the APC, stored in a buffer with ID#1, is located within a functionname. If it isn't, the next line is skipped, if it is, FNC: {function name} is put into ALPHA, and the next line is executed, and moreover the APC is set to the first word of the function.

Here follows a routine in which FNCTOA suits perfect

```
01      LBL 'PRINT
02      BEGIN/END?
03      AON
04      PROMPT
05      AOFF
06      BEG/END
07      LBL 01
08      FNCTOA
09      GTO 02
10      DISTOA
11      GTO 03
12      LBL 02
13      PRA
14      GTO 01
15      LBL 03
16      BEEP
17      PRA
18      END
```

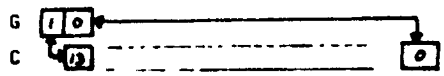
Appendix F Errors of HP41 CPU

The hardware of the HP41 is not bugfree.

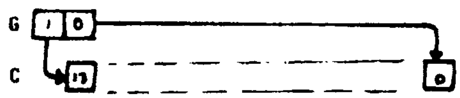
A. In particular the instructions G=C, C=G and C<>G.
You can guess that something must go wrong if R=13 (what is R+1). You should not use these instructions with R=13, unless you studied the following.

1 if you did not change R to 13 one step befor you use C<>G, G=C or C=G, it is rather simpel.

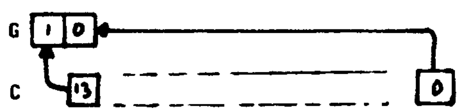
a. C<>G exchanges C[13] with G[1] (!) and C[0] with G(0), or in symbols



b. C=G

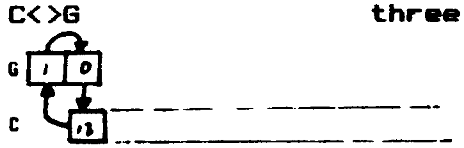


c. G=C

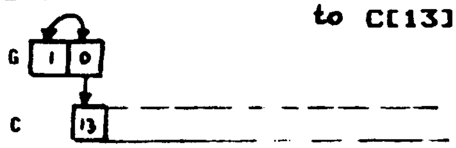


2 if you changed R to 13 one step before you use C<>G, C=G or G=C, by R=13, R=R+1 or R=R-1, strange things happen:

a. C<>G three nibbles circulate



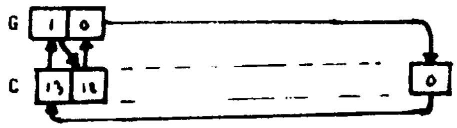
b. C=G the nibbles in G are exchanged, G(0) is *copied* *to* C[13]



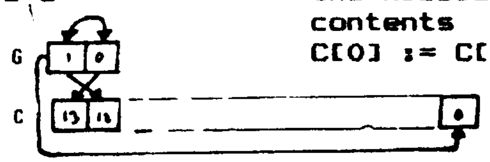
c. G=C G[0]:=G[1], G[1]:=C[13]

3. if R=13 and you change R to 12 using R=R-1 one step before use C=G, G=C or C<>G very strange things happen!

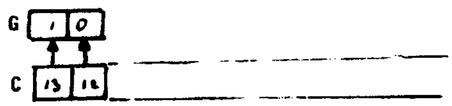
a. C<>G — 5 nibbles are involved in a circulation



b. C=G — the nibbles of G are exchanged, then its contents is copied to C, and at last C[0] := C[13]



c. G=C acts normally:



B. The instructions C=C AND A and C=C OR A do not always work right:

If you have used an arithmetic instruction that is able to yield a carry, and where [13] (MS) is part of the field (so in ALL, MS, @R R=13, P=Q Q=13, R<- R=13), the C=C AND A or C=C OR A is executed, and after that C[13]:=C[0], and A[13]:=C[0].

Therefore, if you want to let these instructions work normally, you should insert a NOP just before them.

Appendix_G_Mainframe_label_rom

For the convenience of the user we have developed a 4K rom, that contains all the important entry points within the VASM listings. These entry points are compatible with the official NOMAS HP listings of the HP-41 operating system.

We haven't included all the entry points, for a lot of them are never used by HP themself. The entry points that are supported by the rom, are those entry points that are called from another 1K block of code in the operating system.

For example: There are four entry points concerning locating the rom head address of a program in rom. Of these four entry points one is used only within the routine itself. The other three are also called from another part of the operating system, and are therefore included. So you will find ROMHED, ROMH05 and ROMH35 included in the label rom but ROMH05 isn't.

The mainframe label rom behaves in the same manner as the user defined labels, but you can not delete them. Therefore they are always present when the label rom is switched on line. Memory lost will not influence the state of the rom

Included in this appendix you will find a printout of all the entry points that are included in the label rom. These entry points are listed in alphabetical order to ease searching if a specific label is included in the rom. After the label name you will find its address in the operating system.

With the help of the label rom you will get very readable printouts of your programs, for you never have to look up which mainframe routine is called. If you have ordered the complete set, you can start writing programs in an easy and simple way now. If you have only ordered the assembler rom itself, we can advice you to order the seperate mainframe label rom as soon as is possible, for it is really easing up writing your programs.

ABS	ADDR:	1076
ABTS10	ADDR:	0D16
ABTSEQ	ADDR:	0D12
ACOS	ADDR:	107D
AD1-10	ADDR:	1809
AD2-10	ADDR:	1807
AD2-13	ADDR:	180C
ADD1	ADDR:	1CE0
ADD2	ADDR:	1CE3
ADDONE	ADDR:	1800
ADRFCH	ADDR:	0004
ADVNCE	ADDR:	114D
AFORMT	ADDR:	0628
AGTO	ADDR:	1085
AJ2	ADDR:	0DD4
AJ3	ADDR:	0DD0
ALCL00	ADDR:	06C9
ALLOK	ADDR:	02CD
ALPDEF	ADDR:	03AE
ANN+14	ADDR:	075B
ANNOUT	ADDR:	075C
AOFF	ADDR:	1345
AON	ADDR:	133C
AOUT15	ADDR:	2C2B
APHST*	ADDR:	2E62
APND-	ADDR:	1FF3
APND10	ADDR:	1FF5
APNDDG	ADDR:	1FFA
APNDNW	ADDR:	2D14
APPEND	ADDR:	2D0E
ARCL	ADDR:	108C
ARGOUT	ADDR:	2C10
ASCLCD	ADDR:	2C5D
ASHF	ADDR:	1092
ASIN	ADDR:	1098
ASN	ADDR:	109E
ASN15	ADDR:	27C2
ASN20	ADDR:	27CC
ASRCH	ADDR:	26C5
ASTO	ADDR:	10A4
ATAN	ADDR:	10AA
AVAIL	ADDR:	28C4
AVAILA	ADDR:	28C7
AVIEW	ADDR:	10B2
AXEQ	ADDR:	10B5
BAKAPH	ADDR:	09E3
BAKDE	ADDR:	09A5
BCDBIN	ADDR:	02E3
BEEP	ADDR:	10BB
BIGBRC	ADDR:	004F
BKROM2	ADDR:	2A91
BLANK	ADDR:	05B7
BLINK	ADDR:	0899
BRT100	ADDR:	1D80
BRT140	ADDR:	1DEC
BRT160	ADDR:	1DA8
BRT200	ADDR:	1E0F
BRT290	ADDR:	1DAC
BRTS10	ADDR:	1D6B
BST	ADDR:	10C2
BSTCAT	ADDR:	0BBA
BSTE	ADDR:	290B
BSTE2	ADDR:	2AF2
BSTEP	ADDR:	28DE
BSTEPA	ADDR:	28EB
CALDSP	ADDR:	29C3
CAT	ADDR:	10C8
CAT$$1	ADDR:	0BC3
CAT$$2	ADDR:	0B53
CAT$$3	ADDR:	1383
CF	ADDR:	10CC
CHK$S	ADDR:	14D8
CHK$S1	ADDR:	14D4
CHK$S2	ADDR:	14D9
CHKAD4	ADDR:	1686
CHKADR	ADDR:	166E
CHKFUL	ADDR:	05BA
CHKRPC	ADDR:	0222
CHRLCD	ADDR:	05B9
CHS	ADDR:	123A
CHSA	ADDR:	1CDA
CHSA1	ADDR:	1CDC
CLA	ADDR:	10D1
CLCTMG	ADDR:	039C
CLDSP	ADDR:	10E0
CLLCDE	ADDR:	2CF0
CLP	ADDR:	10E7
CLR	ADDR:	1733
CLREG	ADDR:	10ED
CLRLCD	ADDR:	2CF6
CLRPGM	ADDR:	228C
CLRREG	ADDR:	2155
CLRSB2	ADDR:	0C00
CLRSB3	ADDR:	0C02
CLSIG	ADDR:	10F3
CLST	ADDR:	10F9
CLX	ADDR:	1101
CNTLOP	ADDR:	0B9D
COLDST	ADDR:	0232
COPY	ADDR:	1109
COS	ADDR:	127C
CPGM10	ADDR:	06F7
CPGMHD	ADDR:	067B
D-R	ADDR:	110E
DAT106	ADDR:	2D4C
DAT231	ADDR:	2D77
DAT260	ADDR:	2D94
DAT280	ADDR:	2D98
DAT300	ADDR:	2D9B
DAT320	ADDR:	2DA2
DAT400	ADDR:	2E05
DAT500	ADDR:	2E10
DATENT	ADDR:	2D2C
DATOFF	ADDR:	0390
DCPL00	ADDR:	2EC3
DCPLRT	ADDR:	2F0B
DCRT10	ADDR:	2F0D
DECAD	ADDR:	29C7
DECADA	ADDR:	29CA
DECMPL	ADDR:	2EC2
DECOCT	ADDR:	1330
DEEXP	ADDR:	088C
DEG	ADDR:	1114
DEGDO	ADDR:	172A
DEL	ADDR:	1124
DELETE	ADDR:	1127
DELLIN	ADDR:	2306
DELNNN	ADDR:	22A8
DEROVF	ADDR:	08EB
DEROW	ADDR:	04AD
DERUN	ADDR:	08AD
DERW00	ADDR:	04B2
DF060	ADDR:	0587
DF150	ADDR:	0482
DF160	ADDR:	0485
DF200	ADDR:	04E7
DFILLF	ADDR:	0563
DFKBCK	ADDR:	0559
DFRST8	ADDR:	0562
DFRST9	ADDR:	0561
DGENS8	ADDR:	0836
DIGENT	ADDR:	0837
DIGST*	ADDR:	08B2
DIV110	ADDR:	18A5
DIV120	ADDR:	18AF
DIV15	ADDR:	18A9
DIVIDE	ADDR:	106F
DOSKP	ADDR:	1631
DOSRC1	ADDR:	24E3
DOSRCH	ADDR:	24E4
DROPST	ADDR:	00E4
DROWSY	ADDR:	0160
DRSY05	ADDR:	0161
DRSY25	ADDR:	0173
DRSY50	ADDR:	0190
DRSY51	ADDR:	0194

DSE	ADDR:	112D
DSPCA	ADDR:	0B35
DSPCRG	ADDR:	0B26
DSPLN+	ADDR:	0FC7
DSWKUP	ADDR:	01AD
DTOR	ADDR:	1981
DV1-10	ADDR:	189A
DV2-10	ADDR:	1898
DV2-13	ADDR:	189D
ENCP00	ADDR:	0952
END	ADDR:	1132
END2	ADDR:	03B6
END3	ADDR:	03BE
ENG	ADDR:	1135
ENLCD	ADDR:	07F6
ENTER^	ADDR:	113E
ERR0	ADDR:	18C3
ERR110	ADDR:	22FB
ERR120	ADDR:	22FF
ERRAD	ADDR:	14E2
ERRAM	ADDR:	2172
ERRDE	ADDR:	282D
ERRIGN	ADDR:	00BB
ERRNE	ADDR:	02E0
ERROF	ADDR:	00A2
ERROR	ADDR:	22F5
ERRPR	ADDR:	2184
ERRSUB	ADDR:	22E8
ERRTA	ADDR:	2F17
EXP10	ADDR:	1A0A
EXP13	ADDR:	1A0D
EXP400	ADDR:	1A21
EXP500	ADDR:	1A61
EXP710	ADDR:	1A4C
EXP720	ADDR:	1A50
EXSCR	ADDR:	192A
E^X	ADDR:	1147
E^X-1	ADDR:	1163
FACT	ADDR:	1154
FC?	ADDR:	115A
FC?C	ADDR:	116B
FDIG20	ADDR:	0E3D
FDIGIT	ADDR:	0E2F
FILLXL	ADDR:	00EA
FIND$1	ADDR:	1775
FIX	ADDR:	1171
FIX57	ADDR:	0AC3
FIXEND	ADDR:	2918
FLGANN	ADDR:	1651
FLINK	ADDR:	2928
FLINKA	ADDR:	2927
FLINKM	ADDR:	2929
FLINKP	ADDR:	2925
FNCTBL	ADDR:	1400
FNDEND	ADDR:	1730
FORMAT	ADDR:	0A7B
FRAC	ADDR:	117C
FS?	ADDR:	1182
FS?C	ADDR:	1188
FSTIN	ADDR:	14C2
GCP112	ADDR:	2BB5
GCPK04	ADDR:	2BBC
GCPK05	ADDR:	2BBE
GCPKC	ADDR:	2B80
GCPKC0	ADDR:	2B89
GENLNK	ADDR:	239A
GENNUM	ADDR:	05E8
GETLIN	ADDR:	1419
GETN	ADDR:	1CEA
GETPC	ADDR:	2950
GETPCA	ADDR:	2952
GETX	ADDR:	1CEF
GETXSQ	ADDR:	1CEE
GETXY	ADDR:	1CEB
GETY	ADDR:	1CED
GETYSQ	ADDR:	1CEC
GOLNGH	ADDR:	0FD9
GOSUBH	ADDR:	0FDD
GOTINT	ADDR:	02F8
GRAD	ADDR:	111A
GSB000	ADDR:	23FA
GSUBS1	ADDR:	23C9
GT3DBT	ADDR:	0FEB
GTACOD	ADDR:	1FDB
GTAI40	ADDR:	0341
GTAINC	ADDR:	0304
GTBYT	ADDR:	29B0
GTBYTA	ADDR:	29BB
GTBYTO	ADDR:	29B2
GTCNTR	ADDR:	0B8D
GTFEN1	ADDR:	20EB
GTFEND	ADDR:	20E8
GTLINK	ADDR:	224E
GTLNKA	ADDR:	2247
GTO	ADDR:	1191
GTO.5	ADDR:	29AA
GTOL	ADDR:	118C
GTONN	ADDR:	2959
GTRMAD	ADDR:	0800
GTSRCH	ADDR:	24DF
H-HMS	ADDR:	1199
HMS+	ADDR:	1032
HMS-	ADDR:	1045
HMS-H	ADDR:	1193
HMSDV	ADDR:	19E5
HMSMP	ADDR:	19E7
IN3B	ADDR:	2865
INBCHS	ADDR:	2E0A
INBYT	ADDR:	29E6
INBYT0	ADDR:	29E3
INBYT1	ADDR:	29EA
INBYTC	ADDR:	29E4
INBYTJ	ADDR:	2E0C
INBYTP	ADDR:	29E5
INCAD	ADDR:	29CF
INCAD2	ADDR:	29D3
INCADA	ADDR:	29D6
INCADP	ADDR:	29D1
INCGT2	ADDR:	0286
IND	ADDR:	0DB2
IND21	ADDR:	0DC4
INEX	ADDR:	2A4A
INLIN	ADDR:	2876
INLIN2	ADDR:	29F6
INPTDG	ADDR:	08A0
INSHRT	ADDR:	2A17
INSLIN	ADDR:	29F4
INSSUB	ADDR:	23B2
INSTR	ADDR:	2A73
INT	ADDR:	1177
INTARG	ADDR:	07E1
INTFRC	ADDR:	193B
INTINT	ADDR:	02FB
INTXC	ADDR:	2A7D
IORUN	ADDR:	27E4
ISG	ADDR:	119E
KEYOP	ADDR:	068A
KYOPCK	ADDR:	0693
LASTX	ADDR:	1228
LBL	ADDR:	11A4
LD90	ADDR:	1995
LDD.P.	ADDR:	0B1D
LDDP10	ADDR:	0B1E
LDSST0	ADDR:	0797
LEFTJ	ADDR:	2BF7
LINN1A	ADDR:	2A93
LINNM1	ADDR:	2A90
LINNUM	ADDR:	2A8B
LN	ADDR:	11A6
LN1+X	ADDR:	1220
LN10	ADDR:	1B45
LN560	ADDR:	1BD3
LNAP	ADDR:	1A8A
LNC10	ADDR:	1AAE
LNC10*	ADDR:	1AAD
LNC20	ADDR:	1ABD

LNSUB	ADDR:	19F9
LNSUB-	ADDR:	19F8
LOAD3	ADDR:	14FA
LOG	ADDR:	11AC
LSWKUP	ADDR:	0180
MASK	ADDR:	2C88
MEAN	ADDR:	11B9
MEMCHK	ADDR:	0205
MEMLFT	ADDR:	05A7
MESSL	ADDR:	07EF
MIDDIG	ADDR:	0DE0
MINUS	ADDR:	1054
MOD	ADDR:	104F
MOD10	ADDR:	195C
MODE	ADDR:	134D
MODE1	ADDR:	134F
MOVREG	ADDR:	215C
MP1-10	ADDR:	184F
MP2-10	ADDR:	184D
MP2-13	ADDR:	1852
MPY150	ADDR:	1865
MSG	ADDR:	1C6B
MSG105	ADDR:	1C80
MSG110	ADDR:	1C86
MSGA	ADDR:	1C6C
MSGAD	ADDR:	1C18
MSGDE	ADDR:	1C22
MSGDLY	ADDR:	037C
MSGE	ADDR:	1C71
MSGML	ADDR:	1C2D
MSGNE	ADDR:	1C38
MSGNL	ADDR:	1C3C
MSGNO	ADDR:	1C64
MSGOF	ADDR:	1C4F
MSGPR	ADDR:	1C43
MSGRAM	ADDR:	1C67
MSGROM	ADDR:	1C6A
MSGTA	ADDR:	1C5F
MSGWR	ADDR:	1C56
MSGX	ADDR:	1C75
MSGYES	ADDR:	1C62
NAM40	ADDR:	0F34
NAM44$	ADDR:	0F7D
NAME20	ADDR:	0EE6
NAME21	ADDR:	0EE9
NAME33	ADDR:	0EEF
NAME37	ADDR:	0F09
NAME4A	ADDR:	0FA4
NAME4D	ADDR:	0FAC
NAMEA	ADDR:	0ED9
NBYTA0	ADDR:	2D04
NBYTAB	ADDR:	2D06
NEXT	ADDR:	0E50
NEXT1	ADDR:	0E45
NEXT2	ADDR:	0E48
NEXT3	ADDR:	0E4B
NFRC	ADDR:	00F1
NFRENT	ADDR:	00C4
NFRFST	ADDR:	00F7
NFRKB	ADDR:	00C7
NFRNC	ADDR:	00A5
NFRN10	ADDR:	015B
NFRPR	ADDR:	00EE
NFRPU	ADDR:	00F0
NFRSIG	ADDR:	00C2
NFRST+	ADDR:	0BEE
NFRX	ADDR:	00CC
NFRXY	ADDR:	10DA
NGRKB1	ADDR:	00C6
NLT000	ADDR:	0E91
NLT020	ADDR:	0EA0
NLT040	ADDR:	0EAA
NM44$5	ADDR:	0F7E
NOPRT	ADDR:	0106
NOREG9	ADDR:	095E
NOSKP	ADDR:	1619
NOTFIX	ADDR:	0ADD
NRM10	ADDR:	1870
NRM11	ADDR:	1871
NRM12	ADDR:	1872
NRM13	ADDR:	1884
NROOM3	ADDR:	28C2
NULT$	ADDR:	0E65
NULT$3	ADDR:	0E7C
NULT$5	ADDR:	0E8F
NULTST	ADDR:	0EC6
NWGOOS	ADDR:	07D4
NXBYT3	ADDR:	29B7
NXBYTA	ADDR:	29B9
NXBYTO	ADDR:	2D0B
NXL1B	ADDR:	2B23
NXL3B2	ADDR:	2B63
NXLCHN	ADDR:	2B49
NXLDEL	ADDR:	2AFD
NXLIN	ADDR:	2B14
NXLIN3	ADDR:	2B5F
NXLINA	ADDR:	2B1F
NXLSST	ADDR:	2AF7
NXLTX	ADDR:	2B77
NXTBYT	ADDR:	2D07
OCTDEC	ADDR:	132B
OFF	ADDR:	11C8
OFSHFT	ADDR:	0749
ON/X10	ADDR:	188B
ON/X13	ADDR:	188E
ONE/X	ADDR:	11D6
OPROMT	ADDR:	2E4C
OUTLCD	ADDR:	2C80
OUTROM	ADDR:	2FEE
OVFL10	ADDR:	1429
P-R	ADDR:	11DC
P10RTN	ADDR:	02AC
P6RTN	ADDR:	1670
PACH10	ADDR:	03EC
PACH11	ADDR:	03F5
PACH12	ADDR:	03FC
PACH4	ADDR:	03E2
PACK	ADDR:	11E7
PACKE	ADDR:	2002
PACKN	ADDR:	2000
PAK200	ADDR:	2055
PAKEND	ADDR:	20AC
PAKSPC	ADDR:	20F2
PAR111	ADDR:	0CED
PAR112	ADDR:	0CF5
PARA06	ADDR:	0D22
PARA60	ADDR:	0D35
PARA61	ADDR:	0D37
PARA75	ADDR:	0D49
PARB40	ADDR:	0D99
PARS05	ADDR:	0C34
PARS56	ADDR:	0C93
PARS75	ADDR:	0CCD
PARSDE	ADDR:	0C90
PARSE	ADDR:	0C05
PARSEB	ADDR:	0D6D
PATCH1	ADDR:	21DC
PATCH2	ADDR:	21E1
PATCH3	ADDR:	21EE
PATCH5	ADDR:	21F3
PATCH6	ADDR:	1C06
PATCH9	ADDR:	1C03
PCKDUR	ADDR:	16FC
PCT	ADDR:	1061
PCTCH	ADDR:	11EC
PCTOC	ADDR:	00D7
PGMAON	ADDR:	0956
PI	ADDR:	1242
PI/2	ADDR:	199A
PKIOAS	ADDR:	2114
PLUS	ADDR:	104A
PMUL	ADDR:	1BE9
PR10RT	ADDR:	0372
PR14RT	ADDR:	1365
PR15RT	ADDR:	22DF
PR3RT	ADDR:	0EDD

PROMF1	ADDR:	05CB
PROMF2	ADDR:	05D3
PROMFC	ADDR:	05C7
PROMPT	ADDR:	1209
PSE	ADDR:	11FC
PSESTP	ADDR:	03AC
PTBYTA	ADDR:	2323
PTBYTM	ADDR:	2921
PTBYTP	ADDR:	2328
PTLINK	ADDR:	231A
PTLNKA	ADDR:	231B
PTLNKB	ADDR:	2321
PUTPC	ADDR:	2337
PUTPCA	ADDR:	2339
PUTPCD	ADDR:	232C
PUTPCF	ADDR:	2331
PUTPCL	ADDR:	2AF3
PUTPCX	ADDR:	232F
PUTREG	ADDR:	215E
QUTCAT	ADDR:	03D5
R-D	ADDR:	120E
R-P	ADDR:	11C0
R/S	ADDR:	1218
R/SCAT	ADDR:	0BB7
RAD	ADDR:	111F
RAK06	ADDR:	0C7F
RAK60	ADDR:	06FA
RAK70	ADDR:	070A
RCL	ADDR:	122E
RCSCR	ADDR:	1934
RCSCR*	ADDR:	1932
RDN	ADDR:	1252
RDNSUB	ADDR:	14E9
REGLFT	ADDR:	059A
RFDS55	ADDR:	0949
RG9LCD	ADDR:	08EF
RMCK05	ADDR:	27EC
RMCK10	ADDR:	27F3
RMCK15	ADDR:	27F4
RND	ADDR:	1257
ROLBAK	ADDR:	2E42
ROMCHK	ADDR:	27E6
ROMH05	ADDR:	066C
ROMH35	ADDR:	0678
ROMHED	ADDR:	066A
ROUND	ADDR:	0A35
ROW0	ADDR:	2766
ROW10	ADDR:	02A6
ROW11	ADDR:	25AD
ROW12	ADDR:	0519
ROW120	ADDR:	0467
ROW933	ADDR:	0487
ROW940	ADDR:	1598
RST05	ADDR:	0098
RSTANN	ADDR:	0759
RSTKB	ADDR:	03BE
RSTMS0	ADDR:	0390
RSTMS1	ADDR:	0392
RSTMSC	ADDR:	0384
RSTSEQ	ADDR:	08A7
RSTSQ	ADDR:	0108
RSTST	ADDR:	08A7
RTJLBL	ADDR:	14C9
RTN	ADDR:	125C
RTN30	ADDR:	272F
RTOD	ADDR:	198C
RUN	ADDR:	07C2
RUNING	ADDR:	011D
RUNNK	ADDR:	04E9
RW0110	ADDR:	04F1
RW0141	ADDR:	037E
R^	ADDR:	1260
R^SUB	ADDR:	14ED
SAR021	ADDR:	2640
SAR022	ADDR:	2641
SAROM	ADDR:	260D
SAVR10	ADDR:	27D5
SAVRC	ADDR:	27DF
SAVRTN	ADDR:	27D3
SCI	ADDR:	1265
SCROL0	ADDR:	2CDE
SCROLL	ADDR:	2CDC
SD	ADDR:	1D10
SEARC1	ADDR:	2434
SEARCH	ADDR:	2433
SEPXY	ADDR:	14D2
SERR	ADDR:	24E8
SETQ=P	ADDR:	0B15
SETSST	ADDR:	17F9
SF	ADDR:	1269
SGT019	ADDR:	25C9
SHF10	ADDR:	186D
SHF40	ADDR:	186C
SHIFT	ADDR:	1348
SIGMA	ADDR:	1C88
SIGMA+	ADDR:	126D
SIGMA-	ADDR:	1271
SIGN	ADDR:	1337
SIGREG	ADDR:	1277
SIN	ADDR:	1288
SINFR	ADDR:	1947
SINFRA	ADDR:	194A
SIZE	ADDR:	1292
SIZSUB	ADDR:	1797
SKP	ADDR:	162E
SKPDEL	ADDR:	2349
SKPLIN	ADDR:	2AF9
SNR10	ADDR:	243F
SNR12	ADDR:	2441
SNROM	ADDR:	2400
SQR10	ADDR:	18BE
SQR13	ADDR:	18C1
SQRT	ADDR:	1298
SRBMAP	ADDR:	2FA5
SST	ADDR:	129E
SSTBST	ADDR:	22DD
SSTCAT	ADDR:	0BB4
STATCK	ADDR:	1CC8
STAYON	ADDR:	12A3
STBT10	ADDR:	2EA3
STBT30	ADDR:	2FE0
STBT31	ADDR:	2FE5
STDEV	ADDR:	11B2
STFLGS	ADDR:	16A7
STK	ADDR:	0DF3
STK00	ADDR:	0DFA
STK04	ADDR:	0E00
STMSGF	ADDR:	03A7
STO	ADDR:	10DA
STO*	ADDR:	12A8
STO+	ADDR:	12B0
STO-	ADDR:	12B9
STO/	ADDR:	12C1
STOLCC	ADDR:	2E5B
STOP	ADDR:	1215
STOPS	ADDR:	03A9
STOPSB	ADDR:	013B
STORFC	ADDR:	07E8
STOSTO	ADDR:	124C
STSCR	ADDR:	1922
STSCR*	ADDR:	1920
SUBONE	ADDR:	1802
SUMCHK	ADDR:	1667
SUMCK2	ADDR:	1669
TAN	ADDR:	1282
TBITMA	ADDR:	2F7F
TBITMP	ADDR:	2F81
TENTOX	ADDR:	12CA
TEXT	ADDR:	2CAF
TGSHF1	ADDR:	1FE7
TIMES	ADDR:	105C
TODEC	ADDR:	1FB3
TOGSHF	ADDR:	1FE5
TONE	ADDR:	12D0
TONE7	ADDR:	1716
TONE7X	ADDR:	16DB

TONEB	ADDR:	16DD
TONSTF	ADDR:	0054
TOOCT	ADDR:	1F79
TOPOL	ADDR:	1D49
TOREC	ADDR:	1E75
TRC10	ADDR:	19A1
TRC30	ADDR:	1E38
TRCS10	ADDR:	1E57
TRG100	ADDR:	1E78
TRG240	ADDR:	1ED1
TRG430	ADDR:	1F5B
TRGSET	ADDR:	21D4
TSTMAP	ADDR:	14A1
TXRW10	ADDR:	04F6
TXTLB1	ADDR:	2FC6
TXTLBL	ADDR:	2FC7
TXTROM	ADDR:	04F5
TXTROW	ADDR:	04F2
TXTSTR	ADDR:	04F6
UPLINK	ADDR:	2235
VIEW	ADDR:	12D6
WKUP10	ADDR:	0184
WKUP21	ADDR:	01A7
WKUP25	ADDR:	01BA
WKUP70	ADDR:	01F5
WKUP80	ADDR:	01FF
X$0?	ADDR:	12DC
X$Y?	ADDR:	12E2
X/Y13	ADDR:	1893
X10TOX	ADDR:	1BF8
X<0?	ADDR:	12E8
X<=0?	ADDR:	12EF
X<=Y?	ADDR:	12F6
X<>	ADDR:	124C
X<>ROW	ADDR:	0026
X<>Y	ADDR:	12FC
X<Y?	ADDR:	1308
X=0?	ADDR:	130E
X=Y?	ADDR:	1314
X>0?	ADDR:	131A
X>Y?	ADDR:	1320
XARCL	ADDR:	1696
XASHF	ADDR:	1748
XASN	ADDR:	276A
XASTO	ADDR:	175C
XAVIEW	ADDR:	0364
XBAR	ADDR:	1CFE
XBAR*	ADDR:	1D07
XBEEP	ADDR:	16D1
XBST	ADDR:	2250
XCAT	ADDR:	0B80
XCF	ADDR:	164D
XCLSIG	ADDR:	14B0
XCLX1	ADDR:	1102
XCOPY	ADDR:	2165
XCUTB1	ADDR:	0091
XCUTE	ADDR:	015B
XCUTEB	ADDR:	0090
XDEG	ADDR:	171C
XDELET	ADDR:	22AF
XDSE	ADDR:	159F
XECROM	ADDR:	2F4A
XEND	ADDR:	2728
XEQ	ADDR:	1328
XEQC01	ADDR:	24EA
XFS?	ADDR:	1645
XFT100	ADDR:	18EC
XGA00	ADDR:	248D
XGI	ADDR:	24C7
XGI07	ADDR:	24DA
XGI57	ADDR:	24C1
XGNN10	ADDR:	2512
XGNN12	ADDR:	2514
XGNN40	ADDR:	255D
XGOIND	ADDR:	1323
XGRAD	ADDR:	1726
XGTO	ADDR:	2505
XISG	ADDR:	15A0
XLN1+X	ADDR:	1B73
XMSGPR	ADDR:	056D
XPRMPT	ADDR:	03A0
XR/S	ADDR:	079D
XRAD	ADDR:	1722
XRDN	ADDR:	14BD
XRND	ADDR:	0A2F
XROM	ADDR:	2FAF
XROMNF	ADDR:	2F6C
XROW1	ADDR:	0074
XRS45	ADDR:	07BE
XRTN	ADDR:	2703
XR^	ADDR:	14E5
XSCI	ADDR:	16C0
XSF	ADDR:	164A
XSGREG	ADDR:	1659
XSIGN	ADDR:	0FF4
XSIZE	ADDR:	1795
XSST	ADDR:	2260
XSTYON	ADDR:	1411
XTOHRS	ADDR:	19B2
XTONE	ADDR:	16DE
XVIEW	ADDR:	036F
XX$0?	ADDR:	1611
XX$Y?	ADDR:	1629
XX<0?	ADDR:	15FA
XX<=0?	ADDR:	160D
XX<=0A	ADDR:	1609
XX<=Y?	ADDR:	1601
XX<Y?	ADDR:	15EF
XX=0?	ADDR:	1606
XX=Y?	ADDR:	1614
XX>0?	ADDR:	15F1
XX>Y?	ADDR:	15F8
XXEQ	ADDR:	252F
XY^X	ADDR:	1B11
X^2	ADDR:	106B
Y-X	ADDR:	1421
Y^X	ADDR:	102A

CARE AND WARRANTY

Eprom care

Store the eprom set in a dry and clean place. Make sure that the feet of the eprom's are protected against bending. If any brakes of, the eprom would be worthless. Do not connect any external power supply to the eproms. Protect the eproms against static charges, otherwise irreparrable damage to the eproms can result. Do not under any circumstances remove the labels on the eproms, as these labels protect the eproms against losing their data by accidental wipe-out through to much U.V. light on the eproms.

Limited 180 day's warranty

For 180 day's from the date of original purchase the 84081B DAVID-ASSEM-Eprom set is warranted against defects in materials and workmanship affecting electronic performance, but not software content. If you sell your unit or present it as a gift, the warranty is automatically transferred to the new owner and remains in effect for the original 180 day's period. During the warranty period, we will repair or, at our option, replace at no charge a product that proves to be defective, provided that you return the product, shipping prepaid, to ERAMCO SYSTEMS, or their official service representative.

WHAT IS NOT COVERED

This warranty does not apply if the product has been damaged by accident or misuse or as the result of service or modification by other than ERAMCO SYSTEMS or their official service representative.

No other express warranty is given. Any other implied warranty of mechantabillity or fitness is limited to the 180 day's period of this written warranty. In no event shall ERAMCO SYSTEMS be liable for consequential damages. This liability shall in no way excede the catalog price of the product at the moment of sale.

Obligation to Make Changes

Products are sold on the basis of specifications applicable at the time of manufacture. ERAMCO SYSTEMS shall have no obligation to modify or update products once sold.

INDEX

Special thanks to Peter van Swieten, Michael Markov, and the people of ERAMCO SYSTEMS, for their most appreciated stimulating criticisms.

David A. van Leeuwen
Noordweg 54
2641 AM Pijnacker
the Netherlands.