ERAMCO
SYSTEMS

ES 86011A
ERAMCO MLDL-OPERATING SYSTEM EPROM

OWNER'S MANUAL

January 1986

Printed in the Netherlands

MLDL operating system eprom

CONTENTS

MLDL operating system eprom

INTRODUCTION

This manual deals with the ERAMCO MLDL operating system eprom. To get a full understanding of all the routines and functions in this eprom set, it is advisable to read through this manual carefully before operating any of the functions or routines.

INSTALLATION

Follow the instructions of your ERAMCO MLDL-box carefully when installing the eprom set in your box. It may be necessary to bend the feet of the two eproms slightly inward to make them fit easily into the eprom sockets. Do not forget to enable the page on which you insert the eproms (for more detailed information on how to insert the eproms, consult your hardware manual of the ERAMCO MLDL-box). A lower address is the most appropiate page for insertion of the eprom. This provides a quick access to the routines and functions available in the ERAMCO MLDL-eprom set.

ORGANISATION OF THE INSTRUCTION SET

As you will soon discover the routines and functions in this eprom set are divided into two sections. The first section contains all the functions and routines that will change anything in the MLDL-ram you are working on. So always be careful when using any of these functions. A single mistake can destroy the whole 4K ram block that is under development.

The second section contains the functions that facilitates working with the MLDL-ram. They do not change anything in the ram but will provide a quicker access to the ram (LROM will tell you almost immediately where you can continue with writing in the ram or where you can store a User-code program).

Note : All inputs which has to be placed in the alpha-register are related to hexadecimal

In the description of the functions it is assumed, that you have one **MLDL** ram page available for exercising the examples. To ensure that the examples work out in the way we have described them, is it necessary to clear one block and to place it at the proper page. Place the first block off your **MLDL** ram at page A. This is easily achieved by turning the appropriate (left) hex rotary switch to A. Disable the block by switching the left enable switch down (off). To avoid problems with the second block, it is advisable to switch this block of too.

After these preparations we can clear the whole block. Input for this is A in ALPHA. Now execute the function CLBL. For detailed information of it's operation see page 14.
Switch the **MLDL** ram page on line by switching the left enable switch to the ON state. It is now ready for the examples.

INPUT : All the hexadecimal input in the ALPHA register is checked on valid data. Data is valid only, if it consists of the hexadecimal characters. These characters are the numbers from 0 upto 9 and the letters A through F. Any other character in ALPHA will cause an error. The display will show DATA ERROR.
If the error occurs in a function during a running program, the error will be displayed and the program is halted at the instruction, that caused the error.

OUTPUT : Every function in this **MLDL** rom that gives an hexadecimal output to the ALPHA register, will automaticcally execute an AVIEW after it has put it's data into the ALPHA register. So, if you are using for example the function LOCA in a program, it is not necessary to do a AVIEW after the function. (Otherwise the result will be displayed twice. In conjunction with the printer your results will also be printed twice.

MLDL WRITE FUNCTIONS

AFAT Append FAT entry
XROM 10,01

The function AFAT enables the user to update the FAT, e.g. to append the starting address of a routine that has been written in the **MLDL**-ram. Functions are only accessable to the HP-41 when they have an entry in the FAT. This also holds true for programs that are transferred to the **MLDL**-ram. The function MMTORAM takes care of this automaticaly.

Input for AFAT is in the format UOPAAA. AAA is the start-address of the function within a page, P is the page number where the function is loaded, O is an offset and U tells the HP-41 if the routine is a M-code routine or a User code program.

U=0 M-code function. The address points to the first word that is executable
U=2 User code routine. The address points to a Global Label

Example : AAA=3FF The start of the function or routine is found at X3FF.

In order to understand the interaction of O and P it is necessary to realise that EPROM and **MLDL**-ram can be placed at every wanted page, e.g. at any desired port. It must also be kept in mind that an EPROM or **MLDL**-ram page contains only 4K. The value of P is only pointing to the page where the **MLDL**-ram is positioned at at this moment. The value of P will change when you address the **MLDL**-ram to a different page. Opposite to this is the behavior of the value for O. O is a constant added to the pagenumber. It will not change when you place the **MLDL**-ram at a different page. The constant O allows you the possibility to execute functions and routines from another page other than the one where the FAT entry is lodged. So it is evident that the page which is called must always be O pages further in the memory.

Example : The page that contain the FAT is at page 8, and the page that contain the routine itself is at page C, address is 490. We want to make an entry for a User-code routine with AFAT.

The value of O (the offset) is C - 8 = 4
The value of P (page containing the fat) is 8.
The value of AAA (start-address) is 490.
The value of U (M- or User code) is 2.
We do now need the following input for AFAT

248490

When we move the first ROM to another address we must also move the second ROM the same number of pages in the same direction if the value of O is something else then zero. Leading zero's in the input can be omitted

Example : For our **MLDL** ram we will write the rom name with the help of ASSM.
This name is coded as follows :

Address	Data	Comment
A086	0B1	1 end of the name
A087	030	0
A088	020	space
A089	012	R
A08A	005	E
A08B	013	S
A08C	015	U
A08D	017	W
A08E	005	E
A08F	00E	N start of the name
A090	3E0	start of the function

Enter ASSM mode and set the address to A085. Make sure you are in user off mode and key in the data words as given. The address is automatically increased every time. See also ASSM for more details.
To be able to see the rom name when we are executing a catalog 2, we have to place the xrom name entry into the FAT. This is done with AFAT.
We do have a function name, so the digit representing U will be zero.
The rom name is not located at another page, so the offset is also zero.
We are working at page A, so the value of P will be A.
The starting address of the function is the first executable word of the function and is in our case located at 090.

This results in a total entry for AFAT of 00A090

As leading zero's can be omitted, we can use A090 as the entry address for AFAT. Write the entry into ALPHA. Go out of ALPHA and execute AFAT. If you do now a catalog 2 you will see NEWUSER 01 in the display when the catalog routine has arrived at page A. (if you have no printer or timer module, it will be the first name that appears in the catalog.

CLBL Clear ram Block
XROM 10,02

Clearing a block of MLDL-ram is done with the help of CLBL. Input is in ALPHA in the format BBBBEEEE.
BBBB is the first word of the block that has to be cleared.
EEEE is the last word of the block that must be cleared.
Execution of CLBL puts zero in all the addresses between the given ones, including the start and end addresses.

Example : We discover after some time, that we don't want to use a certain part of the rom. We could leave it in the ram page, but for good housekeeping we want it to be cleared. This is accomplished by getting the right begin and end address into ALPHA and execution of CLBL. Switch to ALPHA and give as input the start and end address of the block of code we want to clear. The starting address of this block is 7DDE and end address is 7DE8
So the total entry for CLBL is 7DDE7DE8. Get out of ALPHA and execute CLBL. With ASSM you can check, that the words at the specified addresses are deleted.

Another option of CLBL is to clear a whole 4K block at once. For this input P in ALPHA. P represents the pagenumber of the page you want to clear. **** ATTENTION **** This last option is dangerous. It operates like MEMORY LOST, but in this case it is a memory loss of the specified MLDL-ram page.

Example : Switch the other page of MLDL ram to page B. Get into ALPHA and give the address of the page to be cleared (B). Get out of ALPHA and execute CLBL. Now you can switch the second MLDL ram page on line by setting the right enable switch to the ON position.

COPYR COPY Rom page
XROM 10,04

The function COPYR enables the user to copy an entire page of ROM or **MLDL**-ram to another page of **MLDL**-ram. This gives you the opportunity to change anything you want in the just copied block of ROM.

Input is in ALPHA and has the format SD.
S is the page from where the copy has to be made (Source).
D is the page to which the copy is destined (Destination).

This function will sound a low tone to indicate the completion of the function.

Example : We want to make a copy of our working **MLDL** ram page. This could be done with move by giving as input A000AFFFB000. But this will take longer and asks for a more complicated input. Therefore we will make use of COPYR. The input for this example is AB in ALPHA. When this is done, the function COPYR can be executed. After the tone has sounded we can check, if the second rom is available by executing a catalog 2. You will now see the romname NEWUSER 01 appearing twice in the catalog.

CRNAME CReate NAME
XROM 10,05

This function requires the input of a name for a function in the Alpha register. The functions name is written in reversed order to MLDL-ram in the correct format for a function name. Also the FAT entry is automatically created with the function AFAT. For more information on the formats used see AFAT. It is not possible to create a function name on another page using the offset digit 0.
The last letter of the function name is written at the current David Assem address. This allows you to enter the assembler mode and start writing the desired function immediately, without searching for the start adddress of the function first.

Example : We will add a second name and FAT entry to our MLDL-ram page. Use ASSM to step to the desired address, in this case we will go to address A0B3. Leave ASSM mode and write the name of the new function in the Alpha register, for example USER 01. Now execute CRNAME. A tone will sound and the message READY is displayed. The Alpha register contents is replaced by a hexadecimal address, but this address is of no use.
Execution of a catalog 2 will show you the new function name USER 01 in the catalog after NEUSER 01.

DFAT Delete FAT entry
XROM 10,06

The function DFAT is used when you want to delete an entry from the FAT. The function or routine which is deleted will be invisible for the HP-41 after execution of DFAT. The XROM numbers of all the routines and functions that came after the deleted function in the FAT, will get one lower. Pay attention to this fact when you use functions or routines from the ram you are working on. The same input format is used as with AFAT. The difference is that you do not need to specify the value of U.
So the input format will be OPAAA (offset),(page),(address).
DFAT will search in the page with number P and delete the specified entry. Leading zeros may be omitted.

Example : In the example of the function AFAT we have added the function name to the FAT, to give the **MLDL** ram page a name. If you execute a catalog 2, you will see NEWUSER 01 and after this USER 01. The last entry has to be removed.
This is easily accomplished by getting the right entry address into ALPHA and execution of DFAT.
Give in ALPHA the entry address of USER 01. This address is A0B3. Get out of ALPHA and execute DFAT. With a catalog 2 you can check, that the entry has been removed. You should only see NEWUSER 01 in the catalog.

GETROM GET ROM image
XROM 10,07

This is the counterpart of the SAVEROM function. Input format is the same, so the name must be in alpha and the page number must be in x. For more information on the format of the files, we refer to the function SAVEROM.
Getrom will read back the contents of the rom file and put it in the desired ram page. There is no checking done to see if the specified page is actually a ram page. This is to allow you to get a rom file to a page that is not switched on.

Example : If you have saved a rom file on tape, we can demonstrate it coming back. First of all clear the page we are working on. This is done with CLBL. You probably know by now how this function works, so it is left up to you to clear the block. Put in ALPHA the name of the file we want to read back, e.g. USER1. In the X register the page address should be entered to which we want the rom read back. In our case this will be page B. Now the function GETROM can be executed. After it has finished, you can check if it is back again in the usual way with a CAT 2.

IPAGE   Initialize rom PAGE
XROM 10,08

This function sets up a ram page to load user programs and/or assembler code functions. The entire specified page is cleared and the specified xrom number and the name in alpha are written at the appropriate places. This we have already done manually when we explained AFAT. With this function it will be much easier.

Input for this function in ALPHA is the name of the rom. This name must be from one to 11 characters. As it is the name of the rom it is advisable to make it at least 8 characters. This has two reasons. First, a function name of more then 7 characters can not be executed. Second and more important is the fact that the CAT function of the HP-41 CX searches for names that are longer then 7 characters. So, if you use a name of less then 8 characters, the rom name will not show up in the header catalog of the HP-41 CX. This is also the case with the CCD module, a module likely to spread out as much as the PPC rom. Second thing to give as input is the **MLDL** ram page number to be initialized. This page number is given in the X register. (in decimal)

When the function is executed, it will prompt you for the xrom number of the page. There is no checking done on the input, because it is possible to use other xrom numbers, but you can not execute a function in a rom with a xrom number higher then 31, so it is advisable to use a xrom number between 1 and 31. See for the already used xrom numbers appendix D.

The name that will be written to **MLDL** ram consists of the last eleven characters in the alpha register.
Output of the function is in alpha the address of the first empty word as it is used for the function MMTORAM.

Example : We will now initialize our page with the help of IPAGE. Give the desired name in ALPHA. We will make use of the same name as we used in the examples before. It will be NEWUSER 01. Give the right page number in the X register (10). Now execute the function IPAGE. At the prompt the desired xrom number can be given. We will make use of xrom number 21. This is the xrom number for user roms. After a short while a tone will sound and the message READY will be in the display. Pressing ALPHA once gives you the first free byte available to load from. This will be address A092.

MMTORAM Main Memory TO RAM
XROM 10,09

The function MMTORAM is used to copy a program from main memory in the calculator to the desired **MLDL**-ram page in a **MLDL**-box. All the necessary translations for a good operation of this program are made automatically. The Function Access Table (FAT) is updated at the same time with the new Global Labels of the program. For good operation of this function it is necessary to initialize the **MLDL**-ram in the proper way.

Preparation of the **MLDL**-ram: You need a block of ram words that is long enough to hold the desired program. The length of the program can be found with the help of CBT (see CBT). Add two to this number of bytes and you have the number of bytes that will be needed for the program when loaded into the **MLDL**-ram. Now you must find a block in the ram space that is large enough. Write down the starting address of this block. **BE CAREFUL** Addresses in ram are given in hexadecimal form, but the length of the program (by CBT) is given in decimal form. Key into ALPHA the starting address of the block (it's advisable to leave about 20 words between the starting address of the block where the program will be written and the first empty word in the ram you have found, for future revisions).

When you are initializing a 4K block of **MLDL** ram automatically with the help of IPAGE, you do not have to do all of this. The loading address will be automatically given by IPAGE. Also the first next empty word will be returned by MMTORAM to the ALPHA register, to make loading easier.
User flags 0 and 1 can be set or cleared to achieve the desired private status

flag 0	flag 1	status
cleared	cleared	program open
cleared	set	program open, after COPY private
set	cleared	program private
set	set	program private

With the help of these two user flags it is possible to make the program completely private in the **MLDL**-ram, e.g. you can not go into PRGM mode to examine the program and it is not possible to copy the program into the main memory with the help of the COPY function.

A partly private status is also possible. In this case it is possible to examine the program, but after copying it into the main memory it will be private. The third option means no security at all. Programs are now free to examine and to copy (compare with e.g. the math module).Please note that changes in the program are only possible when it is stored in main memory (see the manual of the calculator for it's behavior when you are in rom).

With user flag 3 you can have the option to delete the numeric labels in a program. (for more information about this option see CMPDL).
When this flag is cleared, nothing unusual will happen. The program is first compiled and then loaded into **MLDL**-ram with the desired private status according to the settings of flag 0 and 1. If this flag is set to the contrary, the program will be loaded with all numeric labels deleted. (if this is possible)

MMTORAM can be executed after these preperations regarding the user flags. The function will prompt for the name of the program that has to be copied. It is enough to press ALPHA twice when the program counter is already set in the wanted program. Otherwise you must enter the name of the program in the same way as with CLP or COPY.

MMTORAM calls one of the two present compilers, depending on the status of user flag 3 and will compile the program (for messages during compilation see COMPILE). When the program is compiled, the message LOADING PGM will be displayed. When the whole process is finished, a tone will sound and the message READY will be displayed.

When the function has been finished, it will return the address of the next free byte in **MLDL**-ram. Be carefull. If you are loading manually, this is the address of the first byte after the program. It doesn't have to be necessarily empty. Whenever you are loading, with the **MLDL**-page initialized with IPAGE, it will be the next free byte available.

A CAT 2 will show you the updated FAT with the new labels.
Noting down the start and end-address of the used block will allow you to make changes without address mistakes.

For an example of how to load your user code programs in the **MLDL** box, we rever to **How to set up your own EROM page.** There a complete description is given how to set up a **MLDL** ram page for loading user-code programs.

ROMSUM create ROMSUM of page
XROM 10,11

To check if a ROM is still in good shape HEWLETT-PACKARD has put a checksum in each ROM. With the function ROMSUM you are able to compute this checksum and put it at the proper place in the MLDL-ram you are developing. The checksum is calculated by adding all the words on this page, take modulo 255 and put the remainder in xFFF.
The input is P in ALPHA. P is the page number of the MLDL-ram you want to update the checksum.

Example : To be able to detect if our rom is still in good shape, we are going to compute the checksum of the rom. Give the address of the rom in ALPHA. Attention, we are using the second MLDL ram page now, so the input will be B instead of A. Get back to normal operation mode again and execute the function ROMSUM. This will take a few seconds. During this time the display will remain blank.
When the function is completed, you can check if the checksum is calculated in the proper way. This is achieved by keying into the X-register the used xrom number 21. Now execute ROMCHKX. The display will change into 21 @@-@@ TST. After a few seconds it will change to 21 @@-@@ OK.
(Remember 21 is the xrom number we used for our MLDL ram page).

MOVING M-code

Introduction

Moving mcode around within MLDL-ram (EROM) has been made easier for the 'm-coder' by use of several versinons of a prgram called MOVE.
Initially Mcode prgrammers moved block of words in MLDL-ram with the REG>ROM & ROM>REG routines developed by Paul Lind and Lynn Wilkins. The MOVE routine from the ASSEMBLER 3 eprom set automated the procedure. This process required a lot of data registers to be available in main RAM for temporary storage. Eventually, the ERAMCO MLDL operating system contained a MOVE routine which did not use any main memory from the HP-41.

To be more specific the HP-41's operating system uses this feature of absolute addressing for the following items:

1: jumps -forward or backwards- aftfer checking the carry flag,
2: entries in the FAT and
3: (relocaeable) xeq's and goto's.

The fact that programs written in assembly language use absolute addressing to execute subroutines might be confusing, but once understood this feature is not difficult to use.
However the confusing starts all over when a block of mcode words is moved, meaning that the absolute addresses on which these words reside, are changed.
After this has been done all features using absolute addresses must be updated, something quite tedious if the user has to check manually 4096 words of mcode each time some part of a routine is moved.

The MOVE routine as described here, like other move routines, will allow you to move Mcode around in MLDL-ram. Unlike previous move routines, it will update port dependent XEQ and GOTO instructions and FAT labels (if present) when using the DAVID-Assembler eprom set. User flags 0 and 1 (if set) allow you to DISABLE specific parts of the updating process for port dependent XEQ's and GOTO's, while user flag 2 (if set) will ENABLE the routine that CLEARs the source block or any portion thereof that is not overwritten by the MOVE operation.

user flag	CLEAR	SET
0	enable UPDBL	disable UPDBL
1	enable UPD4K	disable UPD4K
2	disable CLEAR	enable CLEAR

UPDFAT is disabled when the second word in the FAT is temporarily set to 000, while the user defined DA-Ass labels are only updated when this eprom set is plugged in and enabled.

Changes which should be made to jumps (for instance JNC +1C or JC -2D) after a block of
mcode words has been moved are very local problems as these jumps can only cover 63 words (hex: +3F) forward or 64 words (hex: -40) backwards. The main thing for the user regarding jumps is not to forget to update the relevant jumps after a block of mcode words has been moved, meaning that if Mcode words have been moved within the span of a jump instruction to, insert or delete Mcode words the jump distance of the relevant JNC or JC instruction also has to be increased or decreases with the same number of steps.

Automatic updating of jumps is not covered by any of the programs in this set of routines.

Each of the tasks done by MOVE is performed by a subroutine that can also be used seperately, thus giving the programmer greater flexibility. These subroutines are:

- a) UPDLBL UPDdate LaBeLs as provided by the DA-Ass eprom set.
- b) UPDFAT UPDate Function Address Table.
- c) UPDBL UPDate port dependent XEQ's and GOTO's within the block that was moved.
- d) UPD4K UPDate port dependent XEQ's and GOTO's in the rest of the 4K page.
- e) CLEAR Clear the source block, or any remaining part thereof after the move has been accomplished.

Updating the DA-Ass labels is done by the program UPDLBL or the program MOVE uses UPDLBL as a subroutine.
Updating the FAT is done by the program UPDFAT or the program MOve which uses UPDFAT as a subroutine.
Updating relocateable xeq's and goto's is dealt with by the program UPDBL and by its supplementary program UPD4K, which are also part of the main program MOVE.

UPDLBL, UPDFAT, UPDBL, UPD4K and CLEAR can be used individually, one at a time. However, manual procedures are time comsuming and they allow errors to slip in. Therefor, the MOVE routine was created for your convenience; it allows you to execute the whole procedure with a single functin, automatically.

For a thorough understanding of all programs each routine will be explained seperately after which MOVE will be discussed giving the user a complete pcture of this set of routines.

Throughout this write-up reference will be made to the required addressing setup in the alpha register for the programs to execute correctly. This reference to the required addressing scheme will be done in the following form:

BBBBEEEEDDDD

where:

BBBB is the begin address (first word) of the block to be moved.

EEEE is the end address (last word) of the block to be moved.

DDDD is the new starting address of the first word of the block to be moved.

In order to understand the descriptions that follow, the reader might feel the need to acquant himself with the following topics: Extended FAT addressing. This is covered in the appendix.

UPDBL UPDate BLock
XROM 10,12

XEQ's and GOTO'S are two byte instructions to absolute addresses. As long as these addresses cannot be changed by the user (like the 12k operating system, the timer module and the HPIL module; the card reader and the diagnostic rom) this king of XEQ's and GOTO's will suffice.
However most plug-in modules are made port independant which means that the standard kind of two byte XEQ's and GOTO's are not useable anymore (e.g. if a module resides in port 1 an xeq may call a subroutine at absolute address 877F, but when plugged in port 3 the same call should nou be made to absolute adress C77F).
To cover this kind of addressing HP has provided the three byte port independable (or relocateable) XEQ's and GOTO's. These are XEQ's and GOTO's to certain routines in the main operating system which use the 10 bit data word that immediately follows the XEQ or GOTO instruction to create the address to which the relocateable XEQ or GOTO should be addressed, depending on the present page in which the routine resides.

Unfortunately a ten bit word is not large enough to cover a complete 4k addressing space and therefore the 4k addressing space has been split up into 4 equal parts each addressing 1k of the total block of 4096 addresses. these 4 kinds of relocateable xeq's and goto's are referred to in this text as "absolute" relocateable XEQ's and GOTO'S.
HP has included a fifth possibility -referred to in this text as "option 5"- to do a relocateable xeq or goto to an address within the same 1k block in which the relocateable xeq or goto is residing.

The main advantage in using option 5 relocateable xeq's and goto's is the fact that the subroutine in the main frame operating system which covers this type of relocateable xeq or goto does not call a subroutine. This means that there is one more return level available on the return stack for the calling program.
This feature of the option 5 relocateable XEQ's may be used to advantage when writing sophisticated Mcode programs that use several subroutine levels: the HP-41 CPU subroutine stack is only 4 levels deep.

Function	Address		Op.codes
rel. XEQ	1st 1k	X000 - X3FF	349,08C,dab
rel. GOTO	1st 1k	X000 - X3FF	341,08C,dab
rel. XEQ	2nd 1k	X400 - X7FF	36D,08C,dab
rel. GOTO	2nd 1k	X400 - X7FF	365,08C,dab
rel. XEQ	3rd 1k	X800 - XBFF	391,08C,dab
rel. GOTO	3rd 1k	X800 - XBFF	389,08C,dab
rel. XEQ	4th 1k	XC00 - XFFF	3B5,08C,dab
rel. GOTO	4th 1k	XC00 - XFFF	3AD,08C,dab
rel. XEQ	same 1k		379,03C,dab
rel. GOTO	same 1k		369,03C,dab

(dab = data byte used by main frame to make up the address of the relocateable XEQ or GOTO).

It should be noted that the address of the third byte (data byte) of a relocateabel XEW or GOTO defines the 1k block in which the relocateable XEQ or GOTO is residing (i.e. a three byte relocateable XEQ or GOTO at addresses 83FE, 83FF and 8400 resides in the second 1k block of mcode words at page 8: from 8400 upto and including 87FF).
When a block of mcode words has been moved, the following items within this block of mcode words need to be checked for a possible update:

1: all relocateable XEQ's and GOTO's within the moved block pointing towards addresses also within the moved block.
2: option 5 relocateable XEQ's and GOTO's withing the moved block pointing towards addresses outside the moved block.

UPDBL is a program which takes caare of this updating of relocateable XEQ's and GOTO's residing within the moved block of mcode words.
All relocateable xeq's and goto's are checked against the performed move and checked for a possible update.
Before a relocateable xeq or goto is rewritten a check is made if it is possible to rewrite this relocateable xeq or goto using option 5. Only if it is NOT possible to rewrite a relocateable xeq or goto by using option 5 will the absolute form be used, thus ensuring minimum useage of the return stack by the relocateable xeq's or goto's.

UPD4K UPDate 4K byte
XROM 10,15

This program supplements the program UPDBL. UPDBL updates relocateable XEQ's and GOTO's within the moved block. You must also update port dependent XEQ's and GOTO's that branch to routines within the block of Mcode that is moved. This job is done by UPD4K.
After a block of mcode words has been moved UPDBL takes care of updating the relocateable XEQ's and GOTO's within the moved block. However outside the moved block there may also exist relocateable XEQ's and GOTO's which refer to subroutine addresses within the moved block and which need updating.

Special attention must be paid to the handling of absolute relocateabel xeq's and goto's by UPD4K (and UPDBL as well):
A check is made against the addresses involved in the move of a block of mcode words and if needed an update will take place. IF an update is needed UPD4K will first try to use the option 5 form and only if this is NOT possible will the absolute form be chosen.

Note: It must be thoroughly understood that here lies a crucial pitfall for the careless or naive user. If a block of mcode is moved it may involve option 5 relocateable XEQ's or GOTO's which can not abe rewritten in their original form after teh move because a 1k boundary has been crosed. If this option 5 relocateable xeq also fills up the return stack then one return address will be lost because there is one return level less available for absolute relocateable XEQ's.
For instance the extended function page in the new HP=41CX draws heavily upon subroutines and when trying to extract some of the new functions the user might run against the above mentioned problem which is inherent in moving Mmcode around within MLDL-ram.

Another special feature of UPD4Kis the fact that not a complete 4k page minus the moved block is chekcked. The following areas are excluded from a check by UPD4K:

1: addresses X000 and X001
2: the FAT (except when the word at address X001 is zero).
3: the original (old) block of mcode words or -in case of overwriting the old block- the remainder part thereoff.
4: the new block of mcode words.
5: addresses XFF4upto and including XFFF

Points 1 and 5 fall in the error catagory and are explained under "ERROR MESSAGES". Point 4 -updating the moved block- is explained in "USING UPDBL", leaving points 2 and 3 for further explanation.

There is no need to update the old block as this will in most cases not be used anymore and which will probably be cleared as well. This seems especially advisable with regards to the remaining part of a block that is parially overwritten. See "USING CLEAR" for further details.
However there is an argument in favor of not updating the old block of mcode words. Not updating and not clearing the old block of mcode words gives the user the opportunity to create a workable copy of a subroutine which can be used for experimenting without disturbing the original block of mcode words, which (e.g. after a unsuccessfull experiment) can be re-instituted foer normal use while the experimental block just have to be cleared. (Re-instituting the old block can be done very easily by setting up the alpha register with the BBBBEEEE of the new block followed by a DDDD which chould now be the begin address of the old block, then execute UPD4K followed by executing CLEAR).

Finally, UPD4K does noet updadte the FAT under normal conditions. The FAt containds address pointers, not program instructions. Thus, if UPD4K did check the FAT space, it could at best scramble pointers that look like a port dependent XEQ or GOTO. Updating the FAT can be defeated by setting the word at address X001 to zero, ie. by implying that there are no address pointers in the FAT. This trick allows yoy to copy a block of code without altering the FAT, since UPDFAT will now not recognize the FAT as FAT anymore. This procedure is sometimes convenient. However, there is a possible pitfall associated with the use of extended FAT addressing: there is a remote possibility that a FAT pointer will read as 08C or 03C (indicating an offset of either 3 or pages). These pointers could be misinterpreted as a port dependent XEQ or GOTO.

Just to be complete a description of this bug is included here. With reference ot "USING UPDFAT" and the article "EXTENDED FAT ADDRESSING" in the appendix there exists the very unlikely possibility that a disquised FAT-entry is interpreted as a relocateable XEQ or GOTO.
The second word in a three byte relocateable XEQ or GOTO is always 08C for option 5 relocateable XEQ's and GOTO's. IF a 08C or 03C exists in a FAT as the first word of a two byte FAT-entry this indicates an offset eiter 8 or 3 pages ahead. HP has upto now only used 1-page offsets in two specific cases: in the 8k NAVIGATION rom and in the 8k REAL ESTATE rom and even in these cases for user-coded programs only.

But just suppose that in a "disquised" FAT there is an entry of which the first word is 03C. If the last word of the two byte FAT-entry immediately ahead is 379 or 369 then UPD4K (only in case of a disquised FAT) will interpret this combination of bytes (379,03C or 369, 03C) as an option 5 relocateable xeq or goto. If this relocateabele xeq or goto points towards an address within the moved block then an unwanted change in the disquised FAT will be made.
However the first digit of the second word of a two byte FAT-entry should always be 0 (zero). Therefore in the example 379 and 369 are illegal words in the FAT.
Nevertheless the HP-41's operating system does not check the first digit of the second word of a FAT-entry but uses the last two digits of the word to make up the starting address of a program, thus making a correct working FAT possible even with this error.

To sum up all the factors which are involved before this bug can bite:

1: an illegal word must exists in the FAT (i.e.379 or 369).
2: offset page addressing to Mcode routines must exist in the FAT.
3: points 1 and 2 must meet each other in the right combination.
4: the FAT must be disquised to enable UPD4K to check this area of a 4k page.

To be 100% sure that noting of the above can ever happen, it seems adviseable to give every 4k page of MLDL-ram its own name, a FAT which only refers to functions or programs within its own page, a revision code of its own and a corretly updated checksum at the end of the 4k page.

UPDFAT UPDating FAT entries
XROM 10,13

When a block of mcode words has been moved within a specified page, entries in the FAT pointing to addresses within the moved block must also be updated.
The program UPDFAT -also used as a subroutine by the program MOVE- takes care of this problem.
Several aspects regarding an update of the FAT must be taken into account:

1: the size of the FAT.
2: entries pointing towards user-coded programs within the same page.
3: entries pointing towards Mcode programs within the same page.
4: entries pointing towards user-coded programs on another page.
5: entries pointing towards mcode programs on another page.

Aspects 4 and 5 signify a very specialized use of the FAT and this kind of useage of the FAT will normally not be made by the mcode-programmer. However HP uses these possibilities, e.g. in the NAVIGATION rom and in the REAL ESTATE rom, so in order to give the user some knowledge of what might be possible with FAT-entries an article has been appended to this write-up explaining this ultimate use of the FAT.

As it is the philosophy of this set of routines to work on just one page at a time aspects 4 and 5 are not relevant. They will however be taken into account by UPDFAT and when encountered will be left unchanged.
Aspects 2 and 3 are those FAT-entries which are checked updated if they point towards addresses which have been changed by the move of mcode words.

The procedure to update a FAT-entry is basically the same for user coded entries as for mcode entries, however there is one significant difference:
UPDFAT is able to distinguish between thes two kind of FAT-entries and if a FAT-entry points towards a user coded program the mcode words at all addresses are compared against the addresses within the FAT. It should be realized that a user coded program starts with two header bytes which should not be forgotten when performing a move.
If however a FAT-entry points towards a mcode program UPDFAT does NOT take into account the first word of the moved block.

The reason behind this is the fact that just before a starting address of a program which resides in the FAT as a Mcode function there must always be a FAT-name of at least one byte. By disregarding the address of the (original) first byte of a moved block of Mcode words it is now possible to create extra addresses for mcode words between the name of this mcode program and the first byte of the program without updating the FAT which in this case should indeed not be done as the name of the program is not moved.
The size of the FAT is determined by two specific items. First the hex-value at address X001 gives the number of valid FAT-entries while a double 000 NOP indicates the end of the FAT.
UPDFAT only uses the hex-value at address X001 to determine the end address of hte FAT therby giving the user the opportunity to skip this automatic feature of the prgram MOVE.
Check the hex-value of the word at address X001, change it into the NOP value 000 and the FAT will not be updated when the program MOVE is used. UPDFAT does not recognize the FAT when it is "disquised" in this way theprogram MOVE will assume that there is not FAT.
After using this option the user is responsible for re-enterring the value of the original word at address X001, thereby re-insitituting the FAT. If this last fact is forgotten the HP-41 will not recognize any FAT-entries anymore at the page concerned.
Normally the user does not have to be concerned about the FAT. If a move of a block of mcode words is done the program UPDFAT, by itself or as part of the main program MOVE, will take care of any necessary updates of the FAT. However, there are occasions when you may not want to automatically update the FAT. See the UPD4K and CLEAR write-up further on.
Only in some very incidental cases might it be necessary to skip the automatic update of the FAT.

CLEAR — CLEAR block
XROM 10,03

Generally, programmers have found that any portion of MLDL-ram that does not hold useful code should be cleared. This allows you to find unused space when you want to add a new routine, and it reduces the risk of spectecular crashes when you make a mistake. Therefor when you relocate code (as opposed to making a copy), you should clear the source block or any portion thereof that you do not overwrite when you use MOVE.

This task, when done manually is time consuming and it is easy to make mistakes. CLEAR will do the job for you. If the block has to be moved only a few places (to clearing the remainder is especially helpful, since the presence of 000 nops will not affect a running program, provided all relevant jumps have been updated.

A special note regarding the automatic use of CLEAR as a subroutine in the program MOVE is given here:
automatic execaution of CLEAR by the program MOVE is indicated by setting user flag 2. This is opposite to the use of user flags 0 and 1 which must be clear to do there assigned functions. This protocol has been set up intentionally as clearing a part of Mcode words can give some quite surprising results if the user is not fully aware of what is going on. Furthernmore forgetting to set user flag 2 does not do any harm and can be easily corrected by executing CLEAR after using MOVE with the same addressing scheme still residing in the alpha register.

Note that there is a function named CLBL in this operating system to, that enables you to do the same thing. As input format and usage of this function is quite different from CLEAR, we decided to include it in the operating system.

UPDLBL UPDate assm LaBeLs
XROM 10,14

This subroutine works in close concert with the DAVID-Assembler eprom set. This is a 4K module which give the user full control over the HP-41 CPU operation. One of the many features contained in this eprom set is the fact that the user may insert at random addresses self defined labels for easy reference during program developoment or when annotating Mcode programs. These labels will reside in a special buffer in the HP-41 main memory and will remain there as long as the DA-Ass eprom set is plugged in and enabled. When the HP-41 is turned on while the DA-Ass epprom set is not present this buffer will be cleared automatically.
When moving Mcode words in MLDL-ram the addresses of the user defined labels have to be updated like the relocateable XEQ's and GOTO's otherwise they will not refer to the correct starting addresses anymore. UPDLBL takes care of this process automatically.

As the DA-Ass eprom set is an external 4K EROM block for this set of programs, special routines have been created within this set of programs which make it possible to refer to routines within the DA-Ass eprom set. These routines are such that they can find the DA-Ass eprom set at page as long as the DA-Ass eprom set is addressed above page 7.
Of course special attention is being paid to the fact that this set of programs still have to work correctly when the DA-Ass eprom set is not available. Therefor, if the DA-Ass eprom set is not available MOVE will simply skip over this subroutine. If the DA-Ass eprom set is available but there is no buffer containing labels again when used as a subroutine from MOVE, UPDLBL will be automatically skipped.
On the other hand when UPDLBLis used as a stand alone program the above mentioned error conditions will stop the program UPDLBL.
Apart from the error/ready messages which are available to the other subroutines, the following messages are used by UPDLBL as stand alone program:

NONEXISTENT	with tone 4 indicates the fact that UPDLBL can not find the DA-Ass eprom set.
NO LABELS	with tone 7 indicates the fact that altought the DA-Ass eprom set available there is no buffer containing user defined labels.
READY	with tone 7 indicates the successfull completion of UPDLBL as stand alone program, the error message NONEXISTENT will be displayed and the program is stoped.

MOVE MOVE mcode
XROM 10,10

MOVE is the main program of this set of routines which ties all the forgoing subprograms together in one user friendly program. Earlier versions of MOVE made use of a prompting HEX-keyboadr which - do to shortage of space in this new ERAMCO MLDL operating eprom set - had to be deleted. All references by the user in the alpha register.

The error checking as described in this manual, related to the entering of addreses in the alpoha register still remain valid. Furthermore int the ERAMCO operating eprom set additional error checking is done to make sure that the digits entered in the alpha register are valid hex-digits. (0-9 and A-F)

For a complete guide on how MOVE performs extensive error checking on the entered addresses BBBBEEEEDDDD refer to the ERROR MESSAGES.

While MOVE has been created to perform all the subroutines fully automatic some user influence upon the actions of MOVE is possible. The user flags 0, 1 and 2 are used to give the user control over MOVE:

1. if user flag 0 is set MOVE will NOT update the moved block.
2. if user flag 1 is set MOVE will NOT update the remainder part of the 4k block.
3. if user flag 2 is clear MOVE will NOT clear the old block or the remainder thereof.

Furthermore when an automatic update of the FAT is unwanted the Mcode word at address X000 (refer to UPDFAT for further information). at Address= X001 should be set to 000

The last feature incorparated in MOVE is an update of user defined labels when using the DAVID-Assembler eprom set. Actually this update is the first thing being done by MOVE after error checking the entered adresses. MOVE automatically searches for the DA-Ass eprom set which may reside anywhere above page 7 and if found checks the presence of the special label buffer in the HP-41 main memory. If found any labels residing in this buffer and referring to addresses contained in the block of Mcode words being moved, will be updated automatically.

For more details on how to use user defined labels refer to the DA-Ass manual.

Users of this set of routines who do not use the DA-ass eprom set do not have to be concerned about any consequenses as MOVE will automatically skip over the subroutine UPDLBL when the DA-Ass eprom set is not present.

DISPLAY MESSAGES.

To keep the user informed about what is going on, MOVE and its subprograms use several kind of messages which will be displayed when relevant.
Use is also made of the tone function of the HP-41 to inform the user when the program needs attention:

1. when a program is successfully completed tone 7 will sound
2. if an error condition develops tone 4 will sound

The display messages also fall in three catagories:

1. program messages
2. ready messages
3. error messages

It should be realized that a routine may be finish so quickly that the accompanyng message disappears befor the user will be able to read it.

PROGRAM MESSAGES.

LABEL UPDATE used by: MOVE UPDLBL

Indicating that the user defined DA-Ass labels are being updated. In most of the cases this update is performed so quickly that this message only appears very briefly in the display.

MOVING BL used by: MOVE

Indicating that the program is moving a specified block of Mcode words. At the end of the move the FAT (Funtion Address Table) will be automatically updated if the word at address X001 is other than 000.

UPDATING BL used by: MOVE UPDBL

Indicating that the mved block of Mcode words is being searched for relocateable XEQ's and GOTO's and if found will be updated.

UPDATING 4K used by: MOVE UPD4K

Indicating that the remainder part (anything exept the FAT, the old block or the remainder part of the old block and the new block) is being searched for relocateable XEQ's and GOTO's which will be updated if needed.

CLEARING BL used by: MOVE CLEAR

Indicating that the old block or the remainder part thereof is now being cleared.

READY MESSAGES (+tone 7)

READY used by: MOVE UPDLBL UPDBL UPD4K CLEAR

This message will be displayed when a routine has been succesfully completed.

FROM PAGE X used by: MOVE UPD4K CLEAR

This message will be displayed when a routine has been succesfully completed. However it also reminds the user to check the setting of the user flags 1 and 2.
If a block of Mcode is being moved from one page to another only the concerned block can be updated. It is assumed that there are no relocateable XEQ's and GOTO's on the new page already relating to the just entered block of Mcode words, neither will this set of routines clear an old block of Mcode words which does not reside on the same page as the newly entered block of Mcode words.
If user flag 1 is clear the user specifies an update of the remainder part of the 4k page and if user flag 2 is set the user specifies a clearing of the old block of Mcode words. As the program is restricted against use outside a single specified page it disregards the setting of these flags thereby protecting the user against errors. However the user is kindly reminded of his sloppiness regarding the use fof flags 1 and 2.

Note : User flags 0, 1 and 2 are used only by the program MOVE. The routines when used as stand alone programs do not relate to these flags. Therefor, if for instance the user forgets to set user flag 2 to specify a clearing of the old block simply execute CLEAR to do so after using MOVE with the addresing scheme BBBBEEEEDDDD still residing in the alpha register.

NO LABELS used by: UPDLBL

This message will be displayed if UPDLBL is used as a stand alone program with the DA-Ass eprom set plugged in and enabled but no special label buffer present in main memory

FAT UPDATED used by: UPDFAT

After succesfully updating the FAT this message will be displayed, but only when the subprogram UPDFAT is used as a stand alone program.
When the FAT is updated by the main program MOVE this message will be suppressed. At the completion of MOVE instead the message READY will be displayed.

ERROR MESSAGES (+tone 4)

XROM NR=00 used by: MOVE

There is no memory connected to the destination page or it is switched off or the MLDL-ram (EROM) is not initialized.

DDDD NOT EROM used by: MOVE

Destination address is NOT MLDL-ram (EROM).
A "write" check is made to address XFFF by first increasing the checksum by one, checking the new value and then decreasing this to its original value.
If double addressing exists (e.g a plug-in rom and MLDL-ram addressed to the same page) the checksum of the MLDL-ram block will be changed to an unexpected value, because the 'read' will be done to the MLDL-ram.

Regarding the ERAMCO ES-MLDL 1 it should be realized that only the 'read' part of the ram blocks can be switched off. The 'write' function will always be active. If both ram blocks of the ES MLDS 1 are addressed to the same page then block II takes priority cancelling out block I of the mldl-ram.
The possibility of a change of the checsum is regarded acceptable in the event on double addressing as by using MOVE you should expect the checksum to change anyway.

NONEXISTENT used by: UPDLBL

When UPDLBL is exucuted as a stand alone program this message will be displayed while program execution is stoped if the DA-Ass eprom set is not plugged in and enabled.

ADDRESS ERR used by: MOVE UPDLBL UPDFAT UPDBL UPD4K CLEAR.

This error message will be displayed when:

1. there are more than 12 characters in the alpha register.
2. there are less than 12 characters in the alpha register.
3. EEEE is not on the same page as BBBB.
4. EEEE is a lower address than BBBB.

DDDD < X002 used by: MOVE UPDLBL UPDFAT UPDBL UPD4K CLEAR.

This set of routines does not allow a move which would cause a chnge of words at addresses X000 and X001. The words at thes addresses (specifying the xrom nr. and the nr. of FAT-entries) are considered too critical to be used in automatic changing of mcode words. Changing these words should be done manually.
If you want to load a block of mcode words starting at address X000 or X001 do as follows to circumvent the restricted use of thes addresses:
Load the block at any place within the 4k page making sure that the first word is loaded at an address higher than X001 and that the last word loaded is at an address lower than XFF4.
Check the words which are to be loaded at addresses X000 and X001 and load them manually. Now move the remainder part of your mcode block starting from the third word to the destination address X002.

END > XFF3 used by: MOVE UPDLBL UPDFAT UPDBL UPD4K CLEAR.

Not only does this set of routines restrict against the use of the addresse X000 and X001, the same applies to the block of addresses at the end of a page starting from address XFF4. XFF4 upto and including XFFA are the so-called interrupt-points and XFFB upto and including XFFE contain the xrom revision code. XFFF contains the checksum of the 4k block.
Moving a block of EROM words into the interrupt jump addresses will usually crash your HP-41. The sole reedy is to disable the MLDL-read. Then you can clear these interrupt points. If you are unlucky, the accident could load garbage into random lodations of the MLDL-ram. If this happens, your best bet is to reset the entire system: Clear the MLDL-ram, MASTER CLEAR the HP-41 then reload your software.

For the above good reasons the programs this set of routines will not load words into the interrupt area. Any changes to the interrupt area must be done manually, with full knowledge of the associated consequenses.

If however you do want to copy a routine that uses these interrupts, first copy the routine anywhere into your MLDL without the interrupt jump instruction(s) in the interrupt area. Now move this block of Mcode words to its desired position and only after this has been accomplished manually enter the required interrupt jump instructions.

DATA ERROR used by: MOVE UPDLBL UPDFAT UPDBL UPD4K CLEAR.

This main frame error message will be displayed if the address scheme in the al;ha register does contain an illegal hex digits. Only 0-9 and A-F are valid hex digits and any other characters residing in the alpha registers upon execution of any program will generate this erroro messag.
note: this error message will not crate tone 7.

PGM ABORTED used by: MOVE UPDBL UPD4K

While any of these routines are running, continuous scanning of the keyboard is done to check if the ON key or the R/S has been hit.

If so the routine is stopped immediately and the HP-41 is turned off when the ON key was hit or PGM ABORTED is displayed when the R/S key was hit. It should be realized that when this features is used the program stops and exits at whatever address it is ant the moment of hitting the ON key or the R/S key. If this feature is used without a solid knowledge of what is going on a whole block of 4k MLDL-ram might be left in a uncertain state regarding the relocateable XEQ's and GOTO's.

note : The program MOVE may take up to 15 seconds before completion, so be patient before stopping the program MOVE.

XROM 10,16

This is not a normal function. It does not do anything when executed but it is used as a spacer from write routines and application routines within the **MLDL**-ram . One possible application is to use it as a NOP. It will also terminate data input without raising the stack.

UTILITY FUNCTIONS

CBT Count BYtes
XROM 10,17

This function counts the number of bytes that is occupied by a program. The END statement is taken in account. At the prompt the name of the desired program must be keyed in or if you are already in the desired program press ALPHA twice (compare with the function CLP).

Output is given in the display only. The stack and the ALPHA-register are left undisturbed.
If you try to get the length of a program that is resident in a rom module the error message ROM is given.

Example : At the explanation of COMPILE we will write a short user code program to demonstrate you the advantages of COMPILE. Execute COMPILE once more on this program to make sure the program is as compact as possible. Now you can find out how long the program actually is. If you execute CBT and press ALPHA twice, the display will change to 68 BYTES. This is the length of your program including the END statement
Remember this length for you will see that the use of CMPDL will significantly decrease the number of used bytes, thus giving you a lot of memory back.

CMPDL CoMPile and Delete Labels
XROM 10,18

This is in fact nearly the same function as the normal COMPILE. Therefore we are refering to COMPILE for the set up of the flags and the input format for COMPILE. They are both equal.

The only difference is that this function will delete the numeric labels in the program while compiling. This shortens the program and speeds it up. This can be done, because the HP-41 remembers where to jump to in the jump and execute functions. So after the first run of a program, the HP-41 knows the distances to all the labels and will always jump this distance. It does not matter if there is a label or not. Therefore the labels can easily be deleted. Only when the program contains indirect jumps or xeq's is it impossible to do so. This is due to the fact, that the HP-41 can not remember all the possible addresses of all labels in the program. For this reason you can not use this function when the program contains a GTO ind or XEQ ind.

The program respects all the local labels. So the labels A through J and the labels a through e are respected and will not be deleted. This is necessary because the HP-41 searches for them when you use them from the keyboard.

When this function is executed, it will make use of the user registers to hold the addresses of the deleted labels. Therefore make sure that the number of allocated registers is more then the number of labels in the programs. If you don't take care of this the calculator might crash.

To protect the compiled status as much as possible we change the terminated by the .END. This protects you from accidently writing at the end of the program if you want to continue at the end of the programmemory with new programs.

During program compilation, you will see the following messages after each other.

```
PACKING
COMPL 2B G
COMPL 3B G/X
PACKING
COMPL 2B G
COMPL 3B G/X
READY
```

The compiler makes use of the normal compiler. First the whole program is compiled to find out where to jump to. Then all the LBL's are deleted and their addresses are remembered in the user registers. This is done during the packing stage. After this the program is compiled again. When the function is through you are at the beginning of the program.

The user registers contain the information where the program resided and where the specified labels in the program were. The structure of a register is as follows 100SSSSLLLL0NN. The first two digits indicate alpha type of data. The SSSS part gives you the start address of the program in program counter format. The LLLL part gives you the address of the label in the packed program without the labels. The NN part gives you the deleted label at this address.

Example : We will compile the program that we use by the example of COMPILE. This time we are going to compile it with CMPDL. This is easily done. First make sure we have enough empty registers by setting the size to 18 or greater. We can now execute CMPDL. At the prompt give the name of the program : TST. After the compiler has finished we can see the results. Just run the program. Again there is no delay in the first beep. Also notify the fact that the flying goose does not move anymore. This is because the goose only moves one place to the right whenever the program encounters a label. But since all labels are deleted, it is not necessary anymore to move the goose. If you stop the program and execute the function CBT, you will get as result 48 BYTES. This implies that we have saved 20 bytes of memory, and in this case it means that the program is shortened by roughly one third of it's original length.

COD CODe alpha in hex string
XROM 10,19

The hexadecimal number in the ALPHA-register is converted to it's -bit-representation and this will be placed in the X-register. The contents of the ALPHA-register is unchanged. The stack will be rolled up and the value in the X-register before COD was executed is placed in the LASTX-register.
The display won't be intelligable after the function COD has been executed. For the synthetic programmer this will sound normal.

Example : Input in ALPHA the hexadecimal address of our romname and the start address of our romname (A086A090). Execute COD after placing the address in ALPHA. If we change the display format to fix 9, the display will look like this 0.0000708 90 Save this coded representation of the address, for we are using it to demonstrate an example with DECOD.
These so called non normalized numbers (NNN's) should not be used to make calculations, for they can hang up the calculator for quite some time. Also they can not be stored and recalled in the same mannner as normal numbers, for they are normalized after being recalled. This is easily demonstrated by pressing STO 01 and RCL 01 after each other. The result is a zero X register.

COMPILE XROM 10,20 — COMPILE program

The function COMPILE places in every numerical GTO and XEQ the distance to that numerical label. Programs prepared with the help of COMPILE will usually run faster than programs that have to calculate these distances while running. Two byte GOTO's that can not make the distance will be transformed to three byte GOTO's. Therefore your program can be made longer by this routine and it is required to have at least three registers left after the program. (.END. REG xxx with xxx not equal to zero).

Compile prompts for the name of the program you want to compile. Input is in the same way as with the mainframe function CLP. So if you are not in the program you want to compile, you must input the complete name. Otherwise it is possible to press ALPHA twice. The function will first pack the program (PACKING), then handle the two byte GOTO's (COMPL 2B G) and if needed (in this case compile has found a 2 byte GTO that can not make it and will replace it with a three byte GTO, thus causing insertion of null bytes that have to be packed as well) repeat this sequence. After this is done it will continue with the three byte's GOTO's and XEQ's (COMPL 3B G/X). After the routine is finished it will put the message READY in the display. Labels not found will give the error condition NO LBL xx, with the number xx as the label not found. When you switch to program mode you will find the program step that caused the error condition.

If the program has the .END. as last statement instead of a normal END, it will change the .END. into a normal one. This is done for MMTORAM, which expects a program to be terminated with a normal END.

To be able to change the .END. into a normal one, the compiler needs at least one empty register after the program. During the initial packing of the program a check is made to see if there is at least one register available. If this is not the case, the program will terminate with the message TRY AGAIN. If so you should decrease the number of allocated memory registers. (change size)

After execution of compile you will be placed at the first step of the program.

Deleting steps or adding steps in a program, will change the status of the program into a decompiled one. Reusing the compiler will speed up the execution after the editing session.

Example : Create the next program in your calculator

```
01 LBL 'TST          18 GTO 16
02 LBL 00            19 LBL 17
03 LBL 01            20 BEEP
04 GTO 02            21 GTO 00
05 LBL 03            22 LBL 02
06 GTO 04            23 GTO 03
07 LBL 05            24 LBL 04
08 GTO 06            25 GTO 05
09 LBL 07            26 LBL 06
10 GTO 08            27 GTO 07
11 LBL 09            28 LBL 08
12 GTO 10            29 GTO 09
13 LBL 11            30 LBL 10
14 GTO 12            31 GTO 11
15 LBL 13            32 LBL 12
16 GTO 14            33 GTO 13
17 LBL 15            34 LBL 14
                     35 GTO 15
                     36 LBL 16
                     37 GTO 17
```

If you execute this program after you have loaded it, you will notice the significant time it takes before you hear the first beep. You will hear the second one much sooner. Stop the program and goto step 1. Delete the superfluous label 01.
Execute the function COMPILE. You will be prompted for the name of the program to be compiled. Press ALPHA twice, since we are in the program already. (It's also possible to give the full name of the program (TST)).
Now the message PACKING is displayed. If you do not have enough room after the program, COMPILE will terminate with the message TRY AGAIN. Then the messages CMPL 2B G and CMPL 3B G/X will be showed shortly after each other. When the compiler is through these messages, a tone will be sounded and the display gives the message READY.
If you press PRGM once, you will find yourself at the start address of the program. Press PRGM once more and press R/S. Notify the fact that there is no delay before the first beep sounds.

Goto step one once more and delete label 00. Execution of COMPILE will give the error message NO LBL 00. If you go into PRGM mode you will be at the step that caused the error, step 19. Please restore the program with LBL 00 at step 01 again,because we are going to use this program again in the example of CMPDL.

DECOD DECODe non normalized number into alpha
XROM 10,21

The function DECOD is the opposite of the function COD. It will translate a -bit-representation in the X-register to the same hexadecimal form as is used by the function COD. The output is given in the ALPHA-register. When DECOD is executed manually DECOD will also give the hexedecimal representation in the display.

Example : We are going to use the same number as we have created with the function COD. First clear the ALPHA register. Now we must get back our just created number. If you do a RDN, it will come back to the X register. Execute the function DECOD. The hexadecimal representation of the number will appear in the display. If you press back-arrow once, it will disappear and the nonnormalized number is viewed again. Go into ALPHA and discover the hexadecimal representation here.

DISASS DISASSemble m-code
XROM 10,22

This function behaves in the same manner as the DISTOA function of the David Assembler rom. However, this function also decodes the FAT, function names, the polling points at the end of a rom and the revision and checksum codes in the proper way.
Output of the function is also send to the Alpha register, but the format is a little different.
A line of machine code always starts with 7 blanks, except in the case there is a label at that address. In this case the first 6 places on the line are used to display the label.
This feature already makes it a lot easier to read dissassembled listings, as the labels are not any more in between the code itself.
Alos the FAT is printed correctly. The start address of the function and its function name are displayed. Also the number of the FAT entry is displayed. Pay attention to the fact, that this entry number usually is not the real XROM number.
Furthermore when the name of a function is encountered the display will show the string FUNCTION FNAME.
At the end of the rom the polling points are given a name. In this case it is easily seen, which entry point is used and wherefor it is used.
For more information on input and output see also the manual of the David Assembler rom.

LOCA — LOCAte word

XROM 10,23

This function allows you to locate a data-word in a 4K block of ROM, EPROM or **MLDL**-ram.
The input format in ALPHA is as follows: BBBBDDD.
BBBB specifies the address from where LOCA starts searching in the 4K block. Actually it will start at BBBB + 1 to allow repeated search in the block. NONE will be displayed when the wanted data (DDD) is not found in this 4K block. Whenever a data-word is found, it will be displayed together with the address at which it is found. The data in ALPHA (adress + word) will be replaced with the data found.This makes it possible to continue searching for the same word.

Example : With a small user code program you can easily print out all the occurrences of an instruction in a rom or **MLDL** ram page. Create the following user code program (make sure you saved the TST program)

```
01 LBL 'LOCATE
02 'ADD + DATA
03 AON
04 PROMPT
05 AOFF
06 LBL 01
07 LOCA
08 GTO 01
```

Input for this program could be a starting address like X000 and the data to search for could be 040. This would give you a complete list of all the **MLDL** WRITE instructions in the **MLDL** rom. Enter for X the page address where the **MLDL** rom is located (usually page F).

Does not display NONE if None found but does set F10.

LROM Last ROM word
XROM 10,24

LROM searches backwards for the last non zero word in a block beginning at a given start-address. Input is AAAA in ALPHA. The display will give the address of the last non zero word and the value at this address. NONE will be returned when the block between the start address and the beginning of this 4K page does not contain any word (other than zero).
This function can be very useful when the end-address of the last program entered has to be found. In this case the easiest way is to put xFF4 into ALPHA and execute LROM. It will give you the address of the last word that is occupied by the program.

Example : If we want to find out where we can load our next user code programs, we could search for empty space with the help of RAMWR, but this would be rather cumbersome. To avoid this, we are going to use the function LROM. In this case we want to search on page A, starting from the end and working backwards. Input for this is AFFF in ALPHA. Execution of LROM will return A0903E0 to the display after a short search time. This tells us, that the next available word in our rom is at address A091.
If we are searching on a completely empty page, LROM will return the message NONE to the display, because it can not find any word unequal to zero on the page. Try this with page 5 for example. Input for this is 5FFF in ALPHA. Execute LROM. After a short while the message NONE will be displayed.

ROMCHKX ROMCHeck by X-reg
XROM 10,25

This function enables you to check if a ROM or **MLDL**-ram is still in good shape. Important though is the fact that a ROM or **MLDL**-ram must contain a good computed checksum (see ROMSUM for the definition of the checksum). HP rom's will always contain a good checksum. During the test the XROM number is displayed along with the short form of the name and the revision number of the ROM. If the ROM or the **MLDL**-ram doesn't contain this short name or the revision number, the display will show @@-@@.

Input in the X-register, the XROM number of the ROM or **MLDL**-ram you want to test (an example is 31 for the cardreader). During the test XX NN-RR TST will be displayed. XX is the XROM number of the ROM that is tested, NN is the shortened name and RR is the revision number.

Output of ROMCHKX is the display XX NN-RR BAD (indicates a bad ROM) or the display XX NN-RR OK (indicates a good ROM) These outputs will be given only when the function is executed from the keyboard.

The behavior of ROMCHKX will be different when it is executed in a program; when a ROM is found to be good it will do the next step in the program. Else it will skip the next step (compare the function FS?: the rule do if true is in force).
When there is no ROM present with the desired XROM number the message NO ROM XX will be displayed. Again it's behavior in PRGM mode is different. It will act as if the ROM is bad and skip the next line.

Example : We can check if the **MLDL** operating system eprom is still good. For this we need an input of 10 in the X register (this is the xrom number of the **MLDL** rom). When we execute the function ROMCHKX, the display will change to 10 OS-7B TST. This indicates that the rom with xrom number 10 is under test. The revision code of this rom is OS-7B. After a short time the display will change to 10 OS-7B OK . When we execute ROMCHKX with a xrom number that is not present it will say NO ROM nn. This can be tried with zero in the X register because a rom never can have xrom nr 00. The display will show NO ROM 00 after ROMCHKX has been executed.

SAVEROM SAVE ROM image to mass storage
XROM 10,26

With this function you can save the contents of an entire rom on cassette tape. The input format for this function is a name in the alpha register and the desired page number in x.
A file will be created on tape of 640 registers, occupying 20 records.
Because there are a lot of users who have been using the Mountain Computer eprom burner set with the functions READROM and WRTROM we also included a user code program to be able to read back rom files in the old 824 format. This is the program 'RROM in appendix H.
The file identifier on tape for the new file created by SAVEROM is $ 07. This means that the files are presented in the DIR as :

NAME ??,S 640

We have chosen for a nonexistant file type to be sure that the data is not accidently destroyed. Therefore the file is also automatically secured after creation. SAVEROM saves 7 records per file compared to WRTROM or 'WROM. Now you will be able to get the maximum number of roms on your tape (e.g. 24 files).

To get the maximum number of files on your tape it is recommended to do a NEWM with 27 file directory entry's. You can write 12 files on each side of the tape then. After having written 12 files you should protect the tape from rewinding from one side to the other by creating a dummyfile "ENDTAPE" of 300 registers.

Example : If you have a cassette drive you can try the following example. We will save the contents of our rom at page A on tape and read it back with GETROM. Give a filename in ALPHA, for example USER1.
We have the name in ALPHA and now we have to give the page address in the X register. In our example this will be 10. Execute SAVEROM. You will hear the cassette drive working for some time. If you watch the drive closely, you will notice that it writes 20 blocks after each other.
When the drive is ready again you could do a DIR and see as entry in the directory of the tape our just created romfile. It will be in the form as described under the function description,
e.g. USER1 ??,S 640.

APPENDIX A

XROM	NAME	INPUT	OUTPUT
10,01	AFAT	UOPAAA in ALPHA	FAT updated
10,02	CLBL	P / BBBBEEEE in ALPHA	block cleared
10,03	CLEAR	BBBBEEEEDDDD	block is cleared
10,04	COPYR	SD in ALPHA	copied block
10,05	CRNAME	function name	name add and FAT updated
10,06	DFAT	OPAAA in ALPHA	FAT updated
10,07	GETROM	name in ALPHA dec. page in X	4K of tape in ram
10,08	IPAGE	name in ALPHA dec. page in X xrom at prompt	desired page cleared name + xrom in page load addr. in ALPHA
10,09	MMTORAM	BBBB in ALPHA flags 0, 1 and 3	stored program
10,10	MOVE	BBBBEEEEDDDD in ALPHA	block is moved and updated
10,11	ROMSUM	P in ALPHA	romsum in xFFF
10,12	UPDBL	BBBBEEEEDDDD	moved block updated
10,13	UPDFAT	BBBBEEEEDDDD	FAT updated
10,14	UPDLBL	BBBBEEEEDDDD	assm labels updated
10,15	UPD4K	BBBBEEEEDDDD	not moved block updated
10,16	---		
10,17	CBT	name at prompt	length of program
10,18	CMPDL	name of program	short comp. program
10,19	COD	hex in ALPHA	binary in X
10,20	COMPILE	name of program	compiled program
10,21	DECOD	binary in X	hex in ALPHA
10,22	DISASS		mnemonic in ALPHA
10,23	LOCA	BBBBDDD in ALPHA	AAAADDD / NONE
10,24	LROM	BBBB in ALPHA	AAAADDD / NONE
10,25	ROMCHKX	XROM in X	bad / ok do if true
10,26	SAVEROM	name in ALPHA dec. page in X	4K in file on tape

A	address digit
B	begin address digit
D	data digit or destination digit
E	end-address digit
O	offset digit
P	page number digit
S	source digit
U	user digit

APPENDIX B

PROGRAMMING AND THE MLDL EPROM SET

Most functions provided by the **ERAMCO MLDL-EPROM** can be entered in program whenever the eprom-set is plugged in an **ERAMCO MLDL-box** connected to the calculator. When the **ERAMCO MLDL-box** containing the eprom set is connected program lines with eprom functions are displayed and printed as standard functions.

If the box is disconnected, these program lines are displayed and printed as XROM functions with two identification numbers. The first number -11- indicates that the functions are provided in the **ERAMCO MLDL-EPROM.** The second number identifies the particular function. The XROM numbers for the **ERAMCO MLDL-EPROM** are listed below.

Function	XROM Number	Function	XROM Number	Function	XROM Number
AFAT	XROM 10,01	DFAT	XROM 10,06	SAVEROM	XROM 10,26
CBT	XROM 10,17	DISASS	XROM 10,22	UPDBL	XROM 10,12
CLBL	XROM 10,02	GETROM	XROM 10,07	UPDFAT	XROM 10,13
CLEAR	XROM 10,03	IPAGE	XROM 10,08	UPDLBL	XROM 10,14
CMPDL	XROM 10,18	LOCA	XROM 10,23	UPD4K	XROM 10,15
COD	XROM 10,19	LROM	XROM 10,24	--	XROM 10,16
COMPILE	XROM 10,20	MMTORAM	XROM 10,09		
COPYR	XROM 10,04	MOVE	XROM 10,10		
CRNAME	XROM 10,05	ROMCHKX	XROM 10,25		
DECOD	XROM 10,21	ROMSUM	XROM 10,11		

Underlined functions are not programmable.

If program lines using the **ERAMCO MLDL** eprom are entered when the eprom set is not connected, the function is recorded and displayed as XEQ followed by the function name. Program execution will be slowed down by lines in this form because the calculator will first search in main memory for a program or program line with the specified label.

APPENDIX C

MESSAGES

This is a list of messages and errors related to the functions in the ERAMCO **MLDL-EPROM set**. When any of these errors are generated the attempted function is not performed, except as noted.

DISPLAY	FUNCTION	MEANING
BAD MLDL	RAMWR	The **MLDL** ram page is malfunctioning.
ENTRY>64	AFAT	There are already 64 entry's in the FAT.
GTO/XEQ IND	CMPDL MMTORAM	The program contains GTO or XEQ ind statements.
NO ENTRY	DFAT	No such entry exists in the FAT.
NO HPIL	SAVEROM GETROM	The HPIL module is not plugged in.
NO LBL xx	COMPILE CMPDL MMTORAM	The GTO or XEQ has no corresponding LBL in this program.
NONE	LROM LOCA	The whole block is empty. There is no such word in the block from start-address up to the end of the page.
NONEXISTENT	-all- ROM>REG	The ERAMCO **MLDL-EPROM set** is not plugged in or is disabled or is malfunctioning. There are not enough registers available to store the specified block.
NO ROM	RAMWR	An attempt has been made to write to an page which does not have a valid XROM number at the first address of this page.
NO ROM xx	ROMCHKX	The ROM with the given XROM number is not plugged in or disabled.

APPENDIX C

DISPLAY	FUNCTION	MEANING
NO WRITE	RAMWR	The data is not written at the desired address. It is impossible to write to an EPROM or ROM page. Also you can not write at a disabled page.
PAGE > 15	GETROM IPAGE SAVEROM	There is an invalid pagenumber in reg X.
ROM	MKPR MMTORAM COMPILE CMPDL CBT	The named program doesn't exist in main memory but is found in ROM
xx NN-RR BAD	ROMCHKX	The ROM with the XROM number xx is bad.
xx NN-RR OK	ROMCHKX	The ROM with the XROM number xx is ok.
COMPL 2B G	COMPILE CMPDL MMTORAM	The 2 byte GTO's are handled.
COMPL 3B G/X	COMPILE CMPDL MMTORAM	The 3 byte GTO's and XEQ's are handled.
LOADING PGM	MMTORAM	The program is loaded to MLDL ram.
PACKING	COMPILE CMPDL MMTORAM	A byte is deleted and the program is packed to reduce the length of the program.
READY	COMPILE CMPDL IPAGE MMTORAM	The function is ready.

APPENDIX D

XROM numbers range from 1 up to 31 inclusive. As quite a few ROM's are available at the moment of this writing it is advisable to choose a XROM number with care to avoid conflicts with other modules.

ROM name	XROM ID	ROM name	XROM ID
MATH	01	SECUR	19 *
STAT	02	CLINLAB	19 *
SURVEY	03	AVIATION	19 *
FINANCE	04	MONITOR	19 * +
STANDARD	05	STRUCT-B	19 *
CIR ANAL	06	C PPC 1981	20
STRUCT-A	07	ASSEMBLER 3	21
STRESS	08	IL-DEVEL	22
HOME MN	09	I/O	23
GAMES	10 *	IL-DEVEL	24
C PPC 1981	10 *	-EXTFCN	25
AUTODUP	10 *	-TIME-	26
REAL EST	11	- WAND	27
MACHINE	12	-MASS ST	28
THRML	13	(- CTL FNS -	
NAVIG	14	HP-IL MODULE)	
PETROL	15	-PRINTER	29
PETROL	16	CARD READER	30
PLOTTER	17	PPC ROM 2 ??	31
PLOTTER	18	ERAMCO-**MLDL**	10

+ Only a small number of this ROM, an early version of IL-DEVEL ROM, were made and are not stocked or sold by HP.

Those marked with an asterisks share their identifying number, and should not be used in the HP-41 at the same time. Of two functions with the same XROM ID the one at the lowest address (i.e. the lowest numbered port) will be accessed first and the other will be ignored. So use discretion when choosing your own XROM number if you want to avoid these kind of problems.

APPENDIX E

X R O M S T R U C T U R E

XROM's are located at whole 4k blocks of addresses. The lowest addresses in an XROM, and a few of the highest have special functions. The remainder may be filled in any way. The locations in the 4k blocks must be filled by ten bit words, giving 2^10 different codes. They may be read as instructions, or as alpha-numeric data. The following summary, adapted from J. Schwartz' January 1983 PPC Conference paper, should be taken into account when studying an application ROM, e.g. the **MLDL**-ROM. A listing can easily be prepared by using the **MLDL**-ROM functions DISASM and MNEM.

Relative address (hex)	Function of code at that address
X000	The XROM ID number in hexadecimal digits.
X001	The number of functions in the XROM (m), including the XROM name.
X002-3	Address of XROM name
X004-5	Address of first routine, program, etc.
X005-7	Address of second routine, etc.
"	" "
"	" "
X002+2n	Address of n'th routine
X003+2n	
"	" "
"	" "
X002+2m	Address of last (m'th) routine
X003+2m	(m < 64)
X004+2m	Compulsory null - 000.
X005+2m	Compulsory null - 000.
"	" "

```
Add. of name     Name of ROM (running backwards)
   "                  "          "
   "                  "          "
Add. of Fn# 1    Start of Fn# 1 code
   "                  "          "
   "                  "          "
Add. of Fn# 2    Start of Fn# 2 code
   "                  "          "
   "                  "          "
   "                  "          "
 XFF4-A          Special interrupt jump locations ( see table ).
 XFFB-E          ROM name abbreviation and revision #.
 XFFF            ROM checksum for diagnostic use
--------------------------------------------------------------
```

Word pairs containing function addresses:

First word of pair:	b	0	O	O	O	O	a11	a10	a9	a8
Second word of pair:	0	0	a7	a6	a5	a4	a3	a2	a1	a0

This results in the following address in this 4k block if OOOO is zero:

p3 p2 p1 p0 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0

Where p0-3 is the bit representation of the 4k page number and a0-11 represent the relative offset from the beginning of the page.When OOOO is not equal to zero it must be added to p0-3. For more information see the function AFAT.

If the two words would read 003, 0FF this would represent a starting address of a function at address X3FF (hex). The bit b in the first word indicates USER code or microcode. If set the address is the start of a USER code program (e.g. 200, 0A1 in the printer module is address 60A1, start of USER code program "PRPLOT")

APPENDIX F

THE SPECIAL INTERRUPT POINTS

xFF4 Interrupts during PSE loop.
xFF5 Interrupts after each program line.
xFF6 Wake-up with no key down.
xFF7 Interrupts when turned off.
xFF8 Interrupts when peripheral flag is set.
xFF9 wake-up with ON key.
xFFA Wake-up after memory lost.

Do not use these points unless you know exactly what you are doing. Careless use of these points may cause CRASHES.

ASSEMBLY LANGUAGE INFORMATION

SHORT REVIEW OF THE HP-41 INSTRUCTIONS

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.

```
| 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 off 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 last 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 instuctions 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	
DSTTOG	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 instuctions 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.

CLASS 0 OPERATIONS

p=		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NOP		000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
CLRF	*p*	384	304	204	004	044	084	144	284	104	244	0C4	184	344	2C4	---	---
SETF	*p*	388	308	208	008	048	088	148	288	108	248	0C8	188	348	2C8	---	---
?FSET	*p*	38C	30C	20C	00C	04C	08C	14C	28C	10C	24C	0CC	18C	34C	2CC	---	---
LD@R	*p*	010	050	090	0D0	110	150	190	1D0	210	250	290	2D0	310	350	390	3D0
?R=	*p*	394	314	214	014	054	094	154	294	114	254	0D4	194	354	2D4	---	---
R=	*p*	39C	31C	21C	01C	05C	09C	15C	29C	11C	25C	0DC	19C	35C	2DC	---	---
SELP	*p*	3A4	324	224	024	064	0A4	164	2A4	124	264	0E4	1A4	364	2E4	1E4	3E4
WRIT	*p*	028	068	0A8	0E8	128	168	1A8	1E8	228	268	2A8	2E8	328	368	3A8	3E8
?FI	*p*	3AC	32C	22C	02C	06C	0AC	16C	2AC	12C	26C	0EC	1AC	36C	2EC	---	---
READ	*p*	038	078	0B8	0F8	138	178	1B8	1F8	238	278	2B8	2F8	338	378	3B8	3F8
RCR	*p*	3BC	33C	23C	03C	07C	0BC	17C	2BC	13C	27C	0FC	1BC	37C	2FC	---	---

MNEMONIC		OPERATION
NOP		No operation
CLRF	*p*	Clears system flag number *p*
SETF	*p*	Sets system flag number *p*
?FSET	*p*	Set the carry flag if system flag *p* is set
LD@R	*p*	Load *p* into "C" at nibble pointed at by pointer and decrement pointer
?R=	*p*	Set the carry flag if the active pointer equals *p*
R=	*p*	Set the active pointer to *p*
SELP	*p*	Transfer control to the desired peripheral *p*
WRIT	*p*	Write "C" to RAM memory or to the selected device in register *p* of the selected block
?FI	*p*	Set the carry flag if peripheral flag *p* is set
READ	*p*	Read "C" from RAM memory or the selected device to register *p* in the selected block
RCR	*p*	Rotate "C" right by *p* digits

CLASS 0 SPECIAL INSTRUCTION HEX CODES

MNEMONIC	HEX	OPERATION
UNUSED	x34	Not in use
UNUSED	x74	" "
UNUSED	xB4	" "
UNUSED	xF4	" "
ST=0	3C4	Clears flag 0 to 7 ("ST" register)
CLRKEY	3C8	Clears the 'key pressed' flag
?KEY	3CC	Set the carry flag when a key has been pressed
R=R-1	3D4	Decrement the current pointer
R=R+1	3DC	Increment the current pointer
UNUSED	018	Not in use
G=C	058	Copy digits *r,r+1* from "C" to "G"
C=G	098	Copy "G" into digits *r,r+1* from "C"
C<>G	0D8	Exchange "G" with digits *r,r+1* from "C"
UNUSED	118	Not in use
M=C	158	Copy "C" into "M"
C=M	198	Copy "M" into "C"
C<>M	1D8	Exchange "C" with "M"
UNUSED	218	Not in use
T=ST	258	Copy "ST" into "T"
ST=T	298	Copy "T" into "ST"
ST<>T	2D8	Exchange "ST" with "T"
UNUSED	318	Not in use
ST=C	358	Copy digits 1, 0 from "C" into "ST"
C=ST	398	Copy "ST" into digits 1, 0 from "C"
C<>ST	3D8	Exchange digits 1, 0 from "C" with "ST"
XQ->GO	020	Drop stack to convert XQ into GO
POWOFF	060	Go to standby mode
SLCT P	0A0	Select "P" as the active pointer
SLCT Q	0E0	Select "Q" as the active pointer

MNEMONIC	HEX	OPERATION
C=KEY	220	Copy key register into digit 4, 3 of "C"
SETHEX	260	Use hexadecimal arithmetic
SETDEC	2A0	Use decimal arithmetic
DSPOFF	2E0	Turn off the display
DSPTOG	320	Toggle the state of the display
C RTN	360	Return from subroutine if the carry is set
NC RTN	3A0	Return from subroutine if carry flag cleared
RTN	3E0	Do a subroutine return always
UNUSED	030	Not in use
N=C	070	Copy "C" into "N"
C=N	0B0	Copy "N" into "C"
C<>N	0F0	Exchange "C" with "N"
LDI	130	Load next rom word into digits 2-0 of "C"
PUSH	170	Push address digits 6-3 in "C" onto stack
POP	1B0	Pop address from stack into digits 6-3 of "C"
UNUSED	1F0	Not in use
GOTO KEY	230	Load key register into lower 8 bits of "PC"
RAM SLCT	270	Set ram address to digits 2-0 of "C"
UNUSED	2B0	Not in use
WRITEDATA	2F0	Write register "C" to the selected register
FETCH	330	Load 2-0 of "C" from rom address 6-3 of "C"
C=C OR A	370	Logical *or* of "C" with "A" bit by bit
C=C AND A	3B0	Logical *and* of "C" with "A" bit by bit
PRPHSLCT	3F0	Set peripheral address to digit 2-0 of "C"
?P=Q	120	Set the carry flag if the pointers are equal
?LOWBAT	160	Set the carry flag if low battery
A=B=C=0	1A0	Clear registers "A" "B" and "C"
GOTO ADR	1E0	Copy digits 6-3 of "C" into the "PC"

CLASS 1 INSTRUCTIONS

Class 1 instructions are absolute GOTOs and EXECUTEs. They consist of two consecutive ROM words of the following format :

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	0	1
A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	*p*	*p*

A_{15}-A_0 is the 16-bit address to branch to. The *pp* field of the second word determines what type of instruction it is. The next table shows values for *pp* :

pp	MNEMONIC	OPERATION
00	NC XQ	execute subroutine if carry is clear
01	C XQ	execute subroutine if carry is set
10	NC GO	goto rom address if carry is clear
01	C GO	goto rom address if carry is set

Example : NC GO 0232 which jumps to the memory lost routine is coded as :

0011 0010 01 = 0C9 as first word
0000 0010 10 = 00A as second word

CLASS 2 FIELDS OF OPERATION

FIELD	AREA OF OPERATION
ALL	All digits.
M	Mantissa digits 12 - 3.
MS	Mantissa sign digit 13.
XS	Exponent sign digit 2.
S&X	At exponent digits 2 - 0.
@R	At digit specified by the current pointer.
R<-	Up to and including pointer from the right.
PQ	from pointer P, left up to Q, including pointers.

CLASS 2 INSTRUCTIONS

MNEMONIC	OPERATION	@R	S&X	R<-	ALL	PQ	XS	M	S
A=0	clear A	002	006	00A	00E	012	016	01A	01E
B=0	clear B	022	026	02A	02E	032	036	03A	03E
C=0	clear C	042	046	04A	04E	052	056	05A	05E
A<>B	exchange A with B	062	066	06A	06E	072	076	07A	07E
B=A	copy A into B	082	086	08A	08E	092	096	09A	09E
A<>C	exchange A with C	0A2	0A6	0AA	0AE	0B2	0B6	0BA	0BE
C=B	copy B into C	0C2	0C6	0CA	0CE	0D2	0D6	0DA	0DE
C<>B	exchange B with C	0E2	0E6	0EA	0EE	0F2	0F6	0FA	0FE
A=C	copy C into A	102	106	10A	10E	112	116	11A	11E
A=A+B	add B into A	122	126	12A	12E	132	136	13A	13E
A=A+C	add C into A	142	146	14A	14E	152	156	15A	15E
A=A+1	increment A	162	166	16A	16E	172	176	17A	17E
A=A-B	subtract B from A	182	186	18A	18E	192	196	19A	19E
A=A-1	decrement A	1A2	1A6	1AA	1AE	1B2	1B6	1BA	1BE
A=A-C	subtract C from A	1C2	1C6	1CA	1CE	1D2	1D6	1DA	1DE
C=C+C	double C	1E2	1E6	1EA	1EE	1F2	1F6	1FA	1FE
C=A+C	add A into C	202	206	20A	20E	212	216	21A	21E
C=C+1	increment C	222	226	22A	22E	232	236	23A	23E
C=A-C	A-C into C	242	246	24A	24E	252	256	25A	25E
C=C-1	decrement C	262	266	26A	26E	272	276	27A	27E
C=0-C	complement C	282	286	28A	28E	292	296	29A	29E
C=-C-1	nines complement C	2A2	2A6	2AA	2AE	2B2	2B6	2BA	2BE
?B≠0	set carry flag if B≠0	2C2	2C6	2CA	2CE	2D2	2D6	2DA	2DE
?C≠0	set carry flag if C≠0	2E2	2E6	2EA	2EE	2F2	2F6	2FA	2FE
?A<C	set carry flag if A<C	302	306	30A	30E	312	316	31A	31E
?A<B	set carry flag if A<B	322	326	32A	32E	332	336	33A	33E
?A≠0	set carry flag if A≠0	342	346	34A	34E	352	356	35A	35E
?A≠C	set carry flag if A≠C	362	366	36A	36E	372	376	37A	37E
RSHFA	shift A right 1 digit	382	386	38A	38E	392	396	39A	39E
RSHFB	shift B right 1 digit	3A2	3A6	3AA	3AE	3B2	3B6	3BA	3BE
RSHFC	shift C right 1 digit	3C2	3C6	3CA	3CE	3D2	3D6	3DA	3DE
LSHFA	shift A left 1 digit	3E2	3E6	3EA	3EE	3F2	3F6	3FA	3FE

CLASS 3 INSTRUCTIONS

DISTANCE	JNC-	JC-	JNC+	JC+	DISTANCE	JNC-	JC-	JNC+	JC+
+/- 01	3FB	3FF	00B	00F	+/- 02	3F3	3F7	013	017
+/- 03	3EB	3EF	01B	01F	+/- 04	3E3	3E7	023	027
+/- 05	3DB	3DF	02B	02F	+/- 06	3D3	3D7	033	037
+/- 07	3CB	3CF	03B	03F	+/- 08	3C3	3C7	043	047
+/- 09	3BB	3BF	04B	04F	+/- 0A	3B3	3B7	053	057
+/- 0B	3AB	3AF	05B	05F	+/- 0C	3A3	3A7	063	067
+/- 0D	39B	39F	06B	06F	+/- 0E	393	397	073	077
+/- 0F	38B	38F	07B	07F	+/- 10	383	387	083	087
+/- 11	37B	37F	08B	08F	+/- 12	373	377	093	097
+/- 13	36B	36F	09B	09F	+/- 14	363	367	0A3	0A7
+/- 15	35B	35F	0AB	0AF	+/- 16	353	357	0B3	0B7
+/- 17	34B	34F	0BB	0BF	+/- 18	343	347	0C3	0C7
+/- 19	33B	33F	0CB	0CF	+/- 1A	333	337	0D3	0D7
+/- 1B	32B	32F	0DB	0DF	+/- 1C	323	327	0E3	0E7
+/- 1D	31B	31F	0EB	0EF	+/- 1E	313	317	0F3	0F7
+/- 1F	30B	30F	0FB	0FF	+/- 20	303	307	103	107
+/- 21	2FB	2FF	10B	10F	+/- 22	2F3	2F7	113	117
+/- 23	2EB	2EF	11B	11F	+/- 24	2E3	2E7	123	127
+/- 25	2DB	2DF	12B	12F	+/- 26	2D3	2D7	133	137
+/- 27	2CB	2CF	13B	13F	+/- 28	2C3	2C7	143	147
+/- 29	2BB	2BF	14B	14F	+/- 2A	2B3	2B7	153	157
+/- 2B	2AB	2AF	15B	15F	+/- 2C	2A3	2A7	163	167
+/- 2D	29B	29F	16B	16F	+/- 2E	293	297	173	177
+/- 2F	28B	28F	17B	17F	+/- 30	283	287	183	187
+/- 31	27B	27F	18B	18F	+/- 32	273	277	193	197
+/- 33	26B	26F	19B	19F	+/- 34	263	267	1A3	1A7
+/- 35	25B	25F	1AB	1AF	+/- 36	253	257	1B3	1B7
+/- 37	24B	24F	1BB	1BF	+/- 38	243	247	1C3	1C7
+/- 39	23B	23F	1CB	1CF	+/- 3A	233	237	1D3	1D7
+/- 3B	22B	22F	1DB	1DF	+/- 3C	223	227	1E3	1E7
+/- 3D	21B	21F	1EB	1EF	+/- 3E	213	217	1F3	1F7
+/- 3F	20B	20F	1FB	1FF	+/- 40	203	207	---	---

Class 3 instructions allow the program to jump up to 63 words forward or backward from its present location. The mnemonics are JNC and JC.

ROM CHARACTER TABLE

<table>
<tr><th></th><th>lower 4</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr>
<tr><td rowspan="5">upper2</td><td>0</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>1</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>\</td><td>]</td><td>↑</td><td>_</td></tr>
<tr><td>2</td><td></td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>3</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>4</td><td>|-</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>≠</td><td></td><td></td></tr>
</table>

Note : The colon (3A) displays as a boxed star. The comma (2C) is also the left facing goose when used in a function name or display and the period (2E) is also the right facing goose.

You get the hexadecimal code of a character by taking the number in the upper2 column and place the number in the lower row behind it. Last step is to place a zero in front of the number.

Example : The hexadecimal code of the letter W is 017.
Of the equal sign it is 03D

FUNCTION NAMES

When a function is executed, the operating system checks the ROM words containing the first two characters of the function name and the two words immediately following. The catalog table entry for a microcode function (both mainframe and XROM functions) points to the first word of executable code. The function name is listed in reverse order immediately preceding the first word of executable code.

Example : This example shows you how a normal function name is coded.

```
10CE  081  A    Hex 080 added to indicate end of name.
10CF  00C  L
10D0  003  C
10D1  xxx       First executable word of CLA.
```

FUNCTION PROMPTING

To tell the operating system that the end of the function name has been reached, add 080 hex to the final character. To provide a prompt set the top two bits in the first two characters of the function name by adding the hex constants in the following table :

1ST	2ND	NULL alpha	alpha	#dig.	IND & ind stack	stack	none	example
000	any						X	CLA,CLST
100	000	X	X					CLP,COPY
100	100			3,4				SIZE
100	200		X					
100	300			1	X			CAT,TONE
200	000			2	X	X		STO,RCL
200	100			2	X	X		STO,RCL
200	200			2	X			FS?,SF
200	300		X	2	X			
300	000		X	2				LBL
300	100		X	2	X			XEQ(alpha)
300	200		X	2				
300	300		X	2	X X(.ddd)			GTO

The operating system examine these ROM bits and executes a prompt (if the appropiate bits are set) before the function is executed. These prompts are only executed when you execute the function from the keyboard. However, when the function is executed in a program there will be no prompt at all. **Take care of this.**
If the prompt accepts an alpha string, the input data is loaded into the Q register, right justified in reverse order in ASCII.

Example : Execution of the function ASN with the alpha argument "COPY" will load 00 00 00 59 50 4F 4C into the Q register before the function is executed.

If the prompt is numeric the input data is loaded into the "A" register in binary. Whenever the prompt also accepts indirect, the value in the "A" register is increased with hex 60.

Example : Execution of the function RCL with a numeric argument of 55 will return 00 00 00 00 00 00 37 in the "A" register.
If the prompt would have been filled in with IND 55, the "A" register contains 00 00 00 00 00 00 B7.

PROGRAMMABILITY

Two other ROM words of a microcode function are examined by the operating system. The first executable word, if a nop (000), indicates that the function is non-programmable. This means that if you execute the function in program mode, it executes rather than being entered as a program line. SIZE, ASN and CLP are non-programmable functions.
If the first two executable words of a XROM function are both zero, then the function is both non-programmable and executes immediately. This means that no function name is displayed and that the function will not NULL. The function is executed when the key is pressed rather than when the key is released. PRGM, SHIFT and back-arrow are non-programmable, immediate executing functions. Note that unless your routine checks for key release, and the key to which your function is assigned is held down, the function will be executed repeatedly until the key is released. These two words affect the function operation only if the calculator is in PRGM mode. In RUN mode, they are ignored.

Example : these are a few examples of function name promptings.

12D2	097	W	1105	099	Y	12CC	085	E
12D3	005	E	1106	010	P	12CD	00E	N
12D4	109	I	1107	00F	O	12CE	30F	O
12D5	216	V	1108	103	C	12CF	114	T

FUNCTION INDEX

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. Otherwise a pin could brake from the eprom and make it worthless. Do not connect any external power supply to the eproms. Protect the eproms against static charges, otherwise irrepairable damage to the eproms can result. **Do not** remove under any circumstances the labels on the eproms for these labels protect the eproms against loosing there data by accident through too much U.V. light on the eprom's.

Limited 180 day's warranty

The 83120A **ERAMCO MLDL-Eprom** set is warranted against defects in materials and workmanship affecting electronic performance, -but not software content- for 180 day's from the date of original purchase. If you sell your unit or give it as a gift the warranty is automatically transferred to the new owner and remains in effect for the original 180 days period. During the warranty period we will repair or at our option replace at no charge a product that proves to be defective, provided you return the product, shipping prepaid, to **ERAMCO SYSTEMS** or their official service representative.

CARE AND WARRANTY

WHAT IS NOT COVERED

This warranty doesn't apply if the product has been damaged by accident, 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 merchantability or fitness is limited to the 180 days period of this written warranty. In no event shall ERAMCO SYSTEMS be liable for consequential damages. This liability shall in no way exceed 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.

HOW TO SET UP YOUR OWN EROM PAGE

This part of the manual wiil tell you exactly how to set up an Erom image in your MLDL-box. This is done with the help of a few user code routines that are loaded into the MLDL Erom pages. If you follow the instructions to the letter, nothing can go wrong. And with the help of these instructions you should be able to set up your own Erom image.

step 1

The first thing that has to be done is to clear the Erom page you want to work at and to set the Erom block to the proper page. Therefore you must set the first block with the left rotary switch at page A. Set the rotary switch of the other block to page E. Disable both the switches to the left of the leftmost rotary switch (pull them down). When you set the switches in this position, you can compare the results of your actions with the results that will be given in this appendix.

step 2

Now we will first clear both Erom pages. Key in alpha mode the single character "A". Go out alpha mode and execute CLBL (for more details see page 14) Repeat this sequence with the single character "E" in alpha. At this moment your Erom pages should both be clear. Now you can enable both the Erom pages by pushing the both switches up. Don't expect anything to happen yet. Both pages are still empty.

step 3

Before doing anything else we have to make sure that both pages are empty. Key in alpha "AFFF". Now execute LROM. The display should read 'none'. If this is not the case you should control the setting of the switches and try step 2 again. This is done in the same way for the second block, except you now have to key in alpha "EFFF". The reading of the display should be again 'none'. If this isn't the case return to step 2.

step 4

To allow the HP-41 to find anything that is plugged into the system it uses the first word on every page starting from page 5. If this word doesn't contain a valid identifier, it can't execute a routine or function located at that page. Therefore we will continue with the setting of these identifiers for both Erom pages. In fact this identifier is the xrom number of a module. To avoid any problems with other modules it is recommended in this stage to unplug all your modules.

Also the name of the rom module has to be added. For this the function IPAGE is used. It is enough to put the rom name into the ALPHA register. After this you give the 4K page address in the X register. Now you can execute the function IPAGE. It will prompt you for a XROM number. To avoid problems we choose as XROM number the number 21.

Note : In this manual we described two ways to set up an Erom image. First time we did this with the function RAMWR (see page 5). For this is quite a cumbersome way to prepare an Erom image we did incorporate the function IPAGE (see page 35). Here we already gave you an example of how to create your own Erom image.

Example : We will create one Erom image with xrom number 21 and as name "TEST ROM 1A". For this we make use of the RAM page that is controlled by the left rotary and enabling switch. The block is already cleared and enabled in step 2. The block is addressed at page "A". Now we have all relevant data for the block, so we can initialize it.
Key into ALPHA the name of the module and into the X register the address of the RAM page that will hold the Erom image. This address is 10.
Execute the function IPAGE. At the prompt you answer with the desired xrom number E.G. 21. After a while a tone will sound and the message READY is displayed.

step 5

From now on the HP-41 can recognize anything that is written into Erom block one. So lets give it a try. First of all we have to create a little program in main memory that is to be stored in the Erom block.

We will use the following program:

```
LBL 'test
LBL 01
BEEP
GTO 01
END
```

step 6

You have now created a program in the memory of your calculator. But we wanted to have this program in the **MLDL**-box, because it is using up the last free bytes we had. That's no problem. We only have to use MMTORAM to get the program in the Erom page we want it. For this we have to initialize a few things.

When we have initialized our Erom page manually (without use of IPAGE), we have to give the starting address for our program. This address will be the first word to be used by MMTORAM. Do not use the reserved words in an Erom image in which you are to load your programs (see appendix E and appendix F).
If you work with IPAGE however, the starting address is already given in the ALPHA register. When you have to use the ALPHA register between two sessions of loading programs, it is advisable to keep the contents of the ALPHA register in a normal data storage register, or to note it down (be carefull saving the address in a storage register, for MMTORAM can clear all the user registers, when it makes use of CMPDL). This is handy for future use. If you lost the address however, you can find it back with the help of LROM. Increase the address given by LROM with one, and you have the new starting address to store at.

Second thing we have to initialize is the setting of flags 0 and 1, to achieve the desired private status of the loaded program. There are four options for these flags. For a full description of these options we rever to the function MMTORAM at page 8.

Third and last initialisation we have to make is the setting of flag 3. MMTORAM decides on this flag wether it shall use CMPDL or the normal COMPILE function when it is loading a program. See the function CMPDL for the difference between the two compilers.

Example : We are going to load the program described at step 5. This program has to be loaded in a nonprivate, complete open status. Furthermore we do not want the numeric labels to be deleted.
We do not have to give the starting address, for this is given in ALPHA by the function IPAGE.
For a complete open, nonprivate status flags 0 and 1 have to be cleared.
Flag 3 has to be set for we do not want the numeric labels to be deleted.

When these settings are made, the function MMTORAM can be executed. You will see the messages of the compiler and then the message "LOADING PGM". When MMTORAM is finished a tone will sound and the message "READY" is displayed. The program is now loaded in the Erom image and is ready for use.

Note : If you switch to ALPHA you will see that the starting address is changed. It now points to the first free byte after the just loaded program. This provides an easy way of loading subsequent programs.

step 7

First thing we will do is deleting the program from main memory. When you have done this, you should still be able to execute test for it has been stored in the Erom page. So give it a try. You will hear the familiar beeping every time the program is looping. Stop execution of the program and switch to PRGM mode. Whenever you try to insert or delete a program step, you will see the message 'ROM'. This proves that the program has realy been loaded into the MLDL-box. The program is also included in catalog 2. If you execute CAT 2 you will see the label test showing up in your display sooner or later, depending on the amount of other roms that are plugged into the system.

When you want to store more and other programs, you can follow the described procedure starting at step 5.

Load also the programs described on page 21 (TST) and 28 (MDIS). Load the TST program with flag 3 cleared. Look at the program after you have deleted it in main memory. As you will see, it does not contains the numeric labels any more. This and the fact that it is in ROM now, will speed up the execution quite a lot. Load the MDIS program with flags 1 and 3 set. The program will be open in the erom page, but as soon as it is copied back to main memory, it will be private.

This is the end of the description of our MLDL ROM operating system. We hope you will enjoy to work with this rom. If you have any complaints or wishes you want to see in a future rom, please let us know. We will take these in account as much as possible.

ERAMCO SYSTEMS
W. van Alcmade str. 54
1785 LS Den Helder
The Netherlands