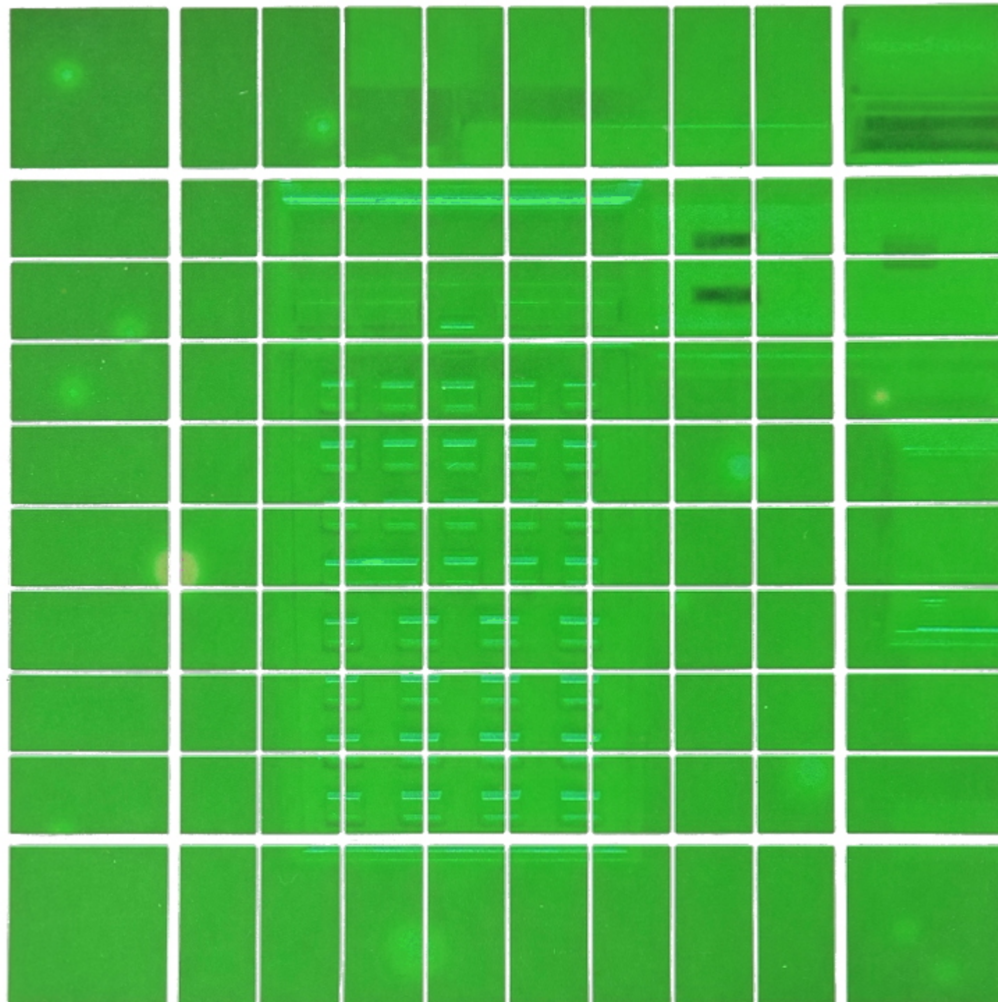
Boost



Boost Documentation

Release 0B

Håkan Thörngren

Nov 20, 2020

Contents

1 Introduction **1**
- 1.1 Plug-in module . . . 1
- 1.2 This release . . . 1
- 1.3 Resource requirements . . . 2
- 1.4 Using this guide . . . 2
- 1.5 Further reading . . . 2
- 1.6 Acknowledgments . . . 2
- 1.7 License . . . 2
- 1.8 The name . . . 3
- 1.9 Feedback . . . 3

2 Fundamentals **5**
- 2.1 OS4 . . . 5
- 2.2 Secondary functions . . . 5
- 2.3 Altered keys . . . 6

3 Catalogs **7**
- 3.1 Overview . . . 7
- 3.2 Buffer catalog . . . 7

4 Execute **9**

5 Assignments **11**
- 5.1 Improved ASN . . . 11
- 5.2 Functions . . . 12

6 Stack and memory **15**
- 6.1 Functions . . . 15

7 Compare functions **17**
- 7.1 Prefix entry . . . 17
- 7.2 The missing compares . . . 18
- 7.3 Compare to constant . . . 18

8 Arithmetic **19**
- 8.1 Functions . . . 19

9 Alpha register **21**

9.1 Functions 21

10 Program related **23**
10.1 Functions 23

11 Buffer stack **25**
11.1 Stack limitations 25
11.2 Buffer keeping 25
11.3 Sanity checking 26
11.4 Return stack extension 26
11.5 Error messages 26
11.6 Functions 26

12 Buffers **31**
12.1 Errors 31
12.2 Functions 31

13 Input functions **33**
13.1 Functions 33
13.2 Timed wait 33

14 Pseudo random numbers **35**
14.1 Functions 35

15 Extended memory **37**
15.1 Random data file access 37
15.2 File operations 38

16 Miscellaneous **41**
16.1 Functions 41

Index **45**

CHAPTER 1

Introduction

Welcome to the Boost module for the HP-41 calculator. Boost is system extensions module, but can also be seen as a companion module to the OS4 module.

1.1 Plug-in module

Boost is a module image that needs to be put in some programmable plug-in module hardware. This can be a Clonix module, an MLDL or some kind of ROM emulator. You need to consult the documentation of ROM emulation hardware for this.

It is also possible to use Boost on HP-41 emulators.

The Boost image is a 2x4K module. Two banks occupies a single 4K page in the normal memory expansion space (page 7–F).

You must also load the separate OS4 module in page 4 for Boost to work.

Note: Clonix and NoV modules will require an update of its firmware to a version that allows independent bank pages, as the original firmware bank switch all pages simultaneously held by the module. A work around is to load OS4 and non-banked modules in one Clonix module while banked application modules are loaded to a second Clonix module.

1.2 This release

This version, 0B is a work in progress module. The existing functions have been tested and are believed to work, however, the module is incomplete in that more functions are planned to be included.

1.3 Resource requirements

Boost will allocate one register from the free memory pool when first powered on. Additional use of Boost may allocate further memory, e.g. using the pseudo random number generator will require one additional register. Using the buffer stack will also allocate memory from the free area.

Apart from this, it does not impose any restrictions on the environment and will run comfortable on any HP-41C, HP-41CV, HP-41CX or HP-41CL.

The XROM number used by this module is 6.

1.4 Using this guide

This guide assumes that you have a working knowledge about:

- The HP-41 calculator, especially its RPN system.

1.5 Further reading

If you feel that you need to brush up your background knowledge, here are some suggested reading:

- The *Owners Manuals* supplied with the HP-41, Hewlett Packard Company.
- *Extend your HP-41*, W Mier-Jedrzejowicz, 1985.
- The *OS4* Documentation, Håkan Thörngren if you want to study the internals of the user OS4 module.

1.6 Acknowledgments

Some of the code sequences used in Boost have been borrowed from, or is based on source code found elsewhere. No permissions for this have been asked for, or been granted by the original authors or copyright owners.

The `CODE` and `DECODE` functions are written by Ken Emery.

Part of the code used for the `R/S` replacement is copyright by Hewlett Packard Company.

1.7 License

The Boost software and its manual is copyright by Håkan Thörngren.

MIT License

Copyright (c) 2020 Håkan Thörngren

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the Software), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED AS IS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR

PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.8 The name

The Boost name was picked as it is meant to provide a good amount of power up for the HP-41, to give it a boost.

1.9 Feedback

Feedback and suggestions are welcome, the author can be contacted at hth313@gmail.com.

The source code and releases can be found at https://github.com/hth313/boost41.

CHAPTER 2

Fundamentals

The chapter gives some basic background of how Boost works and how it differs from many other modules.

2.1 OS4

Boost is a companion module for the OS4 extension module. OS4 provides a lot of ground breaking new features for the HP-41 and Boost is intended to unlock it to the user.

Another module that uses OS4 is Ladybug, which provides an integer mode, much like the HP-16C. In addition to the obvious applications of such module, it can also be used as a system programming module for the HP-41 as it makes it a lot easier to work with non-normalized numbers.

2.2 Secondary functions

Boost make use of secondary functions. Secondary functions can be entered by name, assigned to keys and stored in programs. The only downside is that when stored in a program they occupy a little bit more space, typically 4 bytes instead of 2. Secondary functions with a postfix operand (semi-merged function) costs one byte extra compared to the corresponding XROM function.

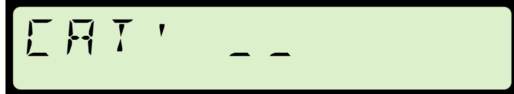
The main advantage of secondary functions is that they allow a 4K module page to have virtually unlimited number of functions.

Seldom used functions and functions that are not expected to be used in programs are good candidates for being secondary functions when you run out of primary XROM numbers.

Note: At the moment it is not possible to display secondary functions in catalog 2. Such feature is planned, but not currently provided. You need to consult the manual as the functions are there are can be reached from the the `XEQ` and `ASN` keys.

2.3 Altered keys

Boost makes use a of a sparse system shell. When the calculator is turned on with Boost plugged in, a system shell is added that replaces the `XEQ`, `ASN` and `CAT` keys with custom versions. You can see that the custom version is active as it displays the name followed by a tick. If you press the catalog key, you will see:



In addition to having the tick, you can also see that it has two underscores. The catalog mechanism is extensible in that it will actually allow other aware plug-in modules to listen to the catalog function and provide catalogs of their own. Even if Boost does not implement a given catalog, it is possible that another plug-in module may provide it and they are all reachable from the same catalog key.

The modified keyboard becomes available when the HP-41 is powered on and is also installed at a master clear (MEMORY LOST).

Some hardware allows for inserting modules by software means while power is on. One example is the `PLUG` functions in the 41CL. Once the module has been inserted by software means, you need to turn the power off and then back on to properly initialize Boost. For the original HP-41, you plug in physical modules while powered off.

Note: When the 41CL does a master clear it disables the MMU which causes any module you may have plugged in to be removed. Thus, after you enable the MMU again, you will need to cycle power to properly initialize Boost.

CHAPTER 3

Catalogs

Boost provides an extension mechanism to the catalog function. Once Boost is inserted it installs a system shell that replaces a couple of keys and `CAT` is one of them.

3.1 Overview

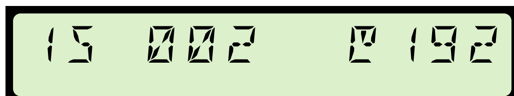
The new `CAT` function prompts for a 2-digit catalog number to show. Once entered, the notification mechanism of OS4 is used to find a module that implements the entered catalog.

If there is no module that overrides the entered catalog it falls back to the normal system catalogs. If there is a module that implements the entered catalog it is handled by that module. Boost itself implements a buffer catalog with number 7 (more are planned in future updates).

Note: The catalogs provided by Boost are built on top of mechanisms in OS4 that gives them very similar behavior compared to the original catalog 1–3. Once stopped the calculator goes to light sleep, saving power. If a key that is not implemented by the catalog is pressed, the catalog is terminated and the key action is performed. This is different from catalogs 4–6 in the HP-41CX which are busy waiting programs that needs to be explicitly exited before normal key pressed can be accepted.

3.2 Buffer catalog

The buffer catalog displays the buffers in normal buffer area. Each buffer is shown together with its address and size. A typical line can look as follows:



The first number is the buffer number (0–15), followed by the size and then the start address. All numbers are decimal.

When the catalog is running you can stop it with the `R/S` key. The `ON` key will turn the HP-41 off and any other key will speed up the display.

When stopped the following keys are active:

SST step to the next entry in the catalog.

BST step to the previous entry in the catalog.

<- terminate the catalog.

R/S continue running through the catalog.

C erase the buffer shown.

When stopped the shift and user keys are active and toggle the state as usual. Any other key press will terminate the catalog shown and the function bound to that key will be executed.

CHAPTER 4

Execute

The XEQ key is very fundamental for the HP-41 as not only does it act as a goto subroutine, it also allows functions that are not on the keyboard to be invoked. The catalogs are searched in order, first catalog 1 which contains user programs, then 2 which are the plug-in modules and finally 3 which are the built-in functions.

The Boost module provides enhanced functionality over the usual XEQ. Once Boost is plugged in the XEQ changes to XEQ' which allows the following alternatives:

By name

Press the ALPHA key and spell out the name of the function you want to execute. This will search for the function in the usual way, but here it also includes searching among the secondary functions.

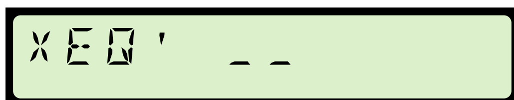
By local label

This is numeric and top row keys single alpha letter labels, as before.

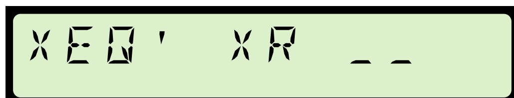
By XROM code

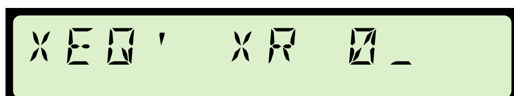
If you press the XEQ key a second time you can key in an XROM function number.

When XEQ' is activated it shows the following:



You can now enter a local label number, press the ALPHA key to spell out the function name, or you can press the XEQ key again and fill in the prompt for an XROM:





XEQ' XR 06,_ _

XEQ' XR 06,1_

XROM 06,10

Note: When the XROM number is complete and you hold the last key down the display changes to display the actual function.

CHAPTER 5

Assignments

In this chapter we look at various enhancements made to key assignment mechanism.

5.1 Improved ASN

The ASN key is replaced by a function ASN' which allows you to make assignments in three ways:

By name

Assignments can be made by name in the same way as before. Simply press the ALPHA key and spell out the name of the function to assign. In addition to the usual functions, this will also search for and handle assignments of secondary functions.

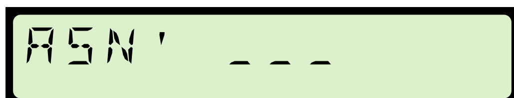
By decimal function code

It is possible to assign any two-bytes function code to a key by filling in the decimal values of the function code.

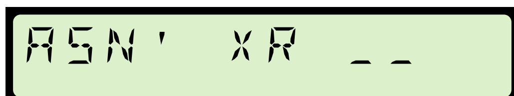
By XROM code

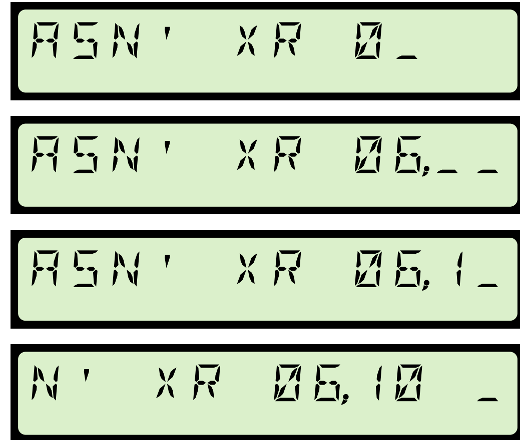
If you first press ASN followed by the XEQ key, that is SHIFT-XEQ-XEQ, the ASN' function will prompt you for a numeric XROM function code and assign it to a key.

When ASN' is activated it shows the following:



You can now press the ALPHA key (as you would normally do) and spell out the function name, or you can press the XEQ key and fill in the prompt for an XROM:





5.2 Functions

5.2.1 CLKYSEC

This will delete all assignments of secondary functions. This is intended to be used when you load a large set of key assignments from some storage media and want them to replace all existing assignments. Devices like the card reader provide two functions, one to load key assignments and one to merge them with the existing ones.

As existing storage media code is unaware of secondary assignments, `CLKYSEC` provides a means of replacing keys as it will wipe the existing secondary key assignments, which is what a load and replace key assignments is intended to do. Not calling `CLKYSEC` means that loading keys acts as merging them with respect to existing secondary assignments.

Note: This is simpler than it sounds. If you want to replace keys, also execute `CLKYSEC`. The reason why this function is needed is that all the old ways of loading key assignments from external storage media uses firmware which is totally unaware of secondary assignments.

5.2.2 LKAOFF

Disable assignments made on the top row keys. This is useful if you have assignments on the top row keys and want to use an RPN program that makes use of the auto-assignment feature of the top row keys.

Assignments on the top row keys remain inactive until you reactivate them again using `LKAON`.

Note: Some application shells like Ladybug will disable top row auto-assignments completely when activated. This is done by a setting in its shell descriptor and in such cases `LKAOFF` has no effect (as long as that shell is active). In such case you need to deactivate the application shell (using `SHIFT-USER`) or activate another application shell on top of it that does not disable the top row assignments.

5.2.3 LKAON

Enable the auto-assignments on the top row keys. This is the default behavior unless you have executed `LKAOFF`.

5.2.4 MAPKEYS

This function rebuilds the key assignment bitmap registers for both primary and secondary assignments. You should not normally need to use this, but it can be useful if you manage to corrupt the bitmap registers or used some function that adjusts key assignments without properly updating the key assignment bitmap bits.

CHAPTER 6

Stack and memory

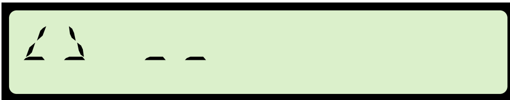
Functions related to stack and memory values are documented here. You may also want to consult the sections about compares and stack as they also operate on stack an memory values. See *Compare functions* and *Buffer stack*.

6.1 Functions

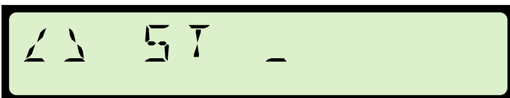
6.1.1 Generic exchange

The <> genereric exchange function takes two arguments and swaps the values described by the arguments. This is a generalization of the built-in `X<>` function. As <> takes two arguments it can exchange values between two arbitrary registers, either or both may be register indirect.

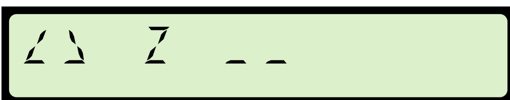
To enter the function you call it by name in the usual way. It will resond by prompting for the first argument:



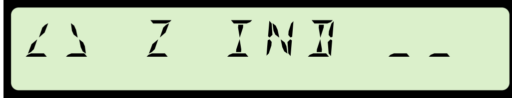
Press the dot key if you want to specify a status register:



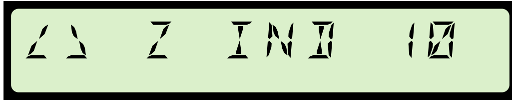
Select the Z register which completes the first argument, you are now prompted for the second argument:



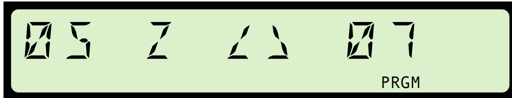
Press the shift key to enter a register indirect argument.



And finally enter the register to complete the function, which is executed when you release the key:



In program memory the <> function is displayed infix:



In this example the `IND` variant was omitted to make the complete function fit on the display together with the line number. You can of course enter indirect arguments in program mode, however, the line becomes so long that it scrolls horizontally.

6.1.2 VMANT

View the mantissa of the value in X. This displays all digits of the X register, stripping off any exponent. The actual value in X is not affected. The bring back to the normal display, press the back arrow key as usual.

6.1.3 F/E

Enables the hybrid `FIX` with `ENG` mode. Normally when using `FIX` mode and the number needs to be shown with an exponent, the HP-41 switches to `SCI` mode. The `F/E` mode changes this so that the HP-41 instead will switch to `ENG` mode in such situation.

CHAPTER 7

Compare functions

The original HP-41 provides a set of compare functions that compares the X and Y register, as well as some to compare X with 0.

The HP-41CX adds additional compare functions such as `X<NN?` which compares X to a register pointed to be the Y register, essentially `X < IND Y?`.

The Boost module provides four prompting generic compare functions =, ≠, < and <=. It should be fairly obvious what they do. They take two arguments and will prompt for them, one at a time. With these you can create any compare you like, e.g. `X < 10?` would test if X is less that the value in register `10`.

`IND Z = IND 01?` would take one register number from stack register Z and one from register `01`. These two values point to two registers that are read and compared if they are equal.

When executed the generic compare functions work the usual way. In a program they will skip the next program line if the test is false. If executed from the keyboard they will display `YES` or `NO`.

7.1 Prefix entry

As any prompting function you start with the function name. To key in `Z = IND L` you need to type `XEQ` ALPHA = ALPHA:

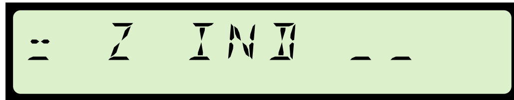
```
=  _ _
```

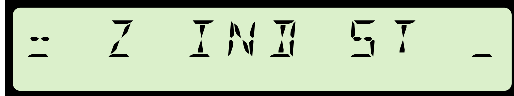
This will show a prompt for the first argument (the one that goes to the left of the equal sign). You can now press the dot key followed by `Z`:

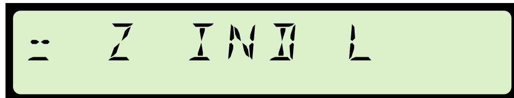
```
=  ST _
```

```
=  Z _ _
```

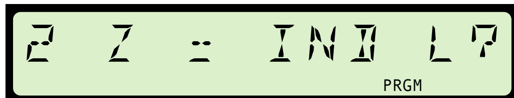
Once the first argument is entered, the calculator will prompt for the second argument. Complete the entry by filling in the second argument:



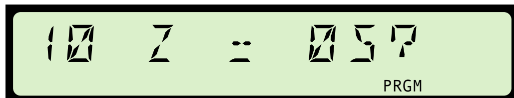




In program memory the OS4 module will display such function with the function name between the two arguments. It will also append the question mark for these functions, which means that once entered you will see `Z = IND L?` when you step to such line. As that program line is a bit too long for the display it will scroll part of it off, leaving you with something like:



A shorter compare function that fits in the display will show together with its line number:



7.2 The missing compares

What if you want to compare if X is greater than register `05`? There is no compare greater-than provided, instead you need to use less-than and swap the arguments, i.e. `05 < X?`.

7.3 Compare to constant

What if you want to compare to zero (or another constant)? In this case you would need to keep zero in a register or load a zero at some point so that it is in a known location in the stack, then compare towards that constant by its register location. One way of seeing is that you use a constant register, simply put the desired constant into a register and keep it there. Doing this way means that you are not limited to zero, you can use any desired constant compare.

CHAPTER 8

Arithmetic

Functions related to arithmetic calculations.

8.1 Functions

8.1.1 DEC _ _

Decrement the specified register, that is, subtract 1 without affecting the last X register.

8.1.2 INC _ _

Increment the specified register, that is, subtract 1 without affecting the last X register.

CHAPTER 9

Alpha register

Functions related to the alpha register and strings.

9.1 Functions

9.1.1 ATOXR

Remove the rightmost character from alpha register and push its character code on the stack. This is similar to `ATOX` in the Extended Functions module, but works on the opposite side of the string in alpha register.

9.1.2 XTOAL

Take the character code from `X` register and append that character to the left side of the string in the alpha register. This is similar to `XTOA` in the Extended Functions module, but works on the opposite side of the string in alpha register.

9.1.3 ARCLINT _ _

Prompting function like `ARCL`, but returns the integer part only of the value.

CHAPTER 10

Program related

Functions related to program control.

10.1 Functions

10.1.1 XEQ>GTO

Drop one level from the return stack, essentially it converts the last `XEQ` to be a `GTO` instead.

10.1.2 RTN?

Test if there is at least one level of return address on the call stack. Executes the next program line if there is, otherwise the next program line is skipped. In keyboard mode it displays `YES` or `NO`.

10.1.3 RTNS

Return the number of pending return levels to `X` register. This will be a number 0–6.

10.1.4 PC<>RTN

Swap the current location counter with the top of the return address stack. Essentially making a return, but setting up for a bounce back to the next program line when that code returns.

10.1.5 GE

Go to the permanent `.END.`. This puts the program location counter at the last address in the last program, which is similar to pressing `GTO ..`, but it will not pack program memory and it will not insert any `END` as is done in some situations.

CHAPTER 11

Buffer stack

The RPN stack is central in the HP-41. However, it is very short and mostly suitable for intermediate values in calculations. A dynamic stack that can be used to tuck away values and contexts and later restore them, to make it appear as we did not actually mess things up inside a subroutine, is sorely missing.

The buffer stack provided here aims to solve this problem. Functions to push and pop values to and from a dynamic stack are provided. The actual storage for the stack is buffer number 3. Space is dynamically allocated and returned from the free memory area as needed.

Hint: To make the buffer stack work well you should keep some free registers around that can be used for the dynamic stack. This means that you should not fill your entire memory to the brim with programs or allocate way more data registers than you actually need.

11.1 Stack limitations

The buffer stack is mainly limited to available free memory, but there is also a hard limit to 253 registers of dynamic stack space due to a buffer being limited to maximum 255 registers (two registers are used for overhead).

The registers needed to push the alpha register varies from zero to four registers. In order to use as few registers as possible to represent the alpha register in the buffer stack, the actual count 0–4 is stored in the buffer trailer register. This limits the push depth of the alpha register to 13.

11.2 Buffer keeping

The stack is created as needed and will remain until you have popped all values from it or explicitly delete it. The buffer will be deleted at power on if the stack is empty. This will reclaim the two remaining registers (header and trailer).

Note: If you abort a calculation so that things are left on the stack, they are not reclaimed and continue to occupy memory. If you have done this and you know that you will never need those pushed items anymore, you need to explicitly remove the stack buffer (`CLSTBUF`). The easiest way to see if you have a stack buffer left behind is to run `CAT 07` and look for buffer number 3 in it. If it exists, you can use the `C` key to erase the buffer when the catalog is stopped on that buffer.

11.3 Sanity checking

There are no actual tagging of elements in the stack like you have on an RPL machine. It is assumed that you write programs that pair stack pushes with pops properly.

However, some elements on the stack do have tags (magic numbers) to detect bad stack use. When you push and pop the flag register, the system flags are not affected and that unused area is used for a magic number to detect mismatched stack operations.

11.4 Return stack extension

Extending the return stack also includes some extra sanity checking where the (unused) PC field of the record in the buffer stack is used to tell if the top element on the stack is a return stack push. This is useful as you may need multiple return stack extensions to hold the recursion depth. In such case you can push something else on the stack before deep recursion and use inspection of the top of stack (`TOPRTN?`) to see if it is a return stack entry to control up-recursion. An alternative way would be to keep track of the number of return stacks pushed in some register.

The return stack extension is not transparent in that you cannot blindly use `XEQ` and `RTN`. You will need to use `PUSHRST` and `POPRST` at appropriate times and provide your own logic for this. You can use the other return stack utilities like " RTN?" and `RTNS` to manage the call stack extension.

11.5 Error messages

The following error message are possible:

`NO ROOM` no more space in the free memory area, or buffer cannot be grown due to size and representation contraints

`STACK ERR` this is given if you try to pop from an empty stack, or it is determined that you tried to pop something that is not the top stack element

11.6 Functions

11.6.1 PUSH _ _

Push a single register to the stack. This function takes a single postfix argument which allows for data registers and RPN stack registers to be pushed.

11.6.2 POP _ _

Pop a single register from the stack. This function takes a single postfix argument which allows for data registers and RPN stack registers to be popped.

Note: A `POP X` will replace the value in `X` register without having any other affect on the stack. Thus, `POP` is more like `STO` to the given location than a `RCL` of a value.

11.6.3 PUSHA

Push the alpha register to the buffer stack. You can have a maximum of 13 alpha registers on the stack at any time, trying to push more will result in a `NO ROOM` error message. The actual register consumption depends on how long the string in the alpha register is. Pushing an empty alpha register costs nothing, apart from using up one of the 13 levels.

11.6.4 POPA

Pop the alpha register from the buffer stack.

11.6.5 PUSHFLG

Push the flag register.

11.6.6 POPFLG

Pop the flag register.

11.6.7 PUSHRST

Push the call stack on the buffer stack. This also clears all current stack levels as the buffer stack can be seen as an extension to the call stack.

11.6.8 POPRST

Pop the call stack from the buffer stack.

11.6.9 PUSHST

Push the entire RPN `XYZTL` stack (five registers) to the buffer stack.

11.6.10 POPST

Pop the entire RPN `XYZTL` stack from the buffer stack.

11.6.11 POPFLXL

POP and fill `X` and `L` registers. This function pops the entire RPN `XYZTL` stack from the buffer stack, but keeps the current value in the `X` register. The popped `X` value is moved to the `L` (last `X`) register.

This is useful when you write a RPN program that takes a single operand from `X`, performs some calculations that disrupts the stack and leaves a result in `X`. Now with `POPFLXL` you can restore the other stack registers and as a bonus have a proper last `X` value, so that your RPN program behaves as a normal single argument function, e.g. like `SIN`.

11.6.12 POPDRXL

POP, drop and fill `X` and `L` registers. This function pops the entire RPN `XYZTL` stack from the buffer stack, but keeps the current value in the `X` register. The popped `X` value is moved to the `L` (last `X`) register. This also drops the RPN stack to simulate that it was dropped, meaning the old `T` register is duplicated to `Z`, and the old `Z` is dropped to `Y` while the old `Y` value is discarded.

This is useful when you write a routine that takes two operands from `X` and `Y`, performs some calculations that disrupts the stack and leaves a result in `X`. Now with `POPDRXL` you can restore the other stack registers and as a bonus have a proper last `X` value, so that your RPN program behaves as a normal two arguments function, e.g. like `+`.

11.6.13 PUSHBYX

Push a range of data registers. Takes a register range `RRR.BBB` in the `X` register. `RRR` is the first register in the range and `BBB` is the last register to push.

11.6.14 POPBYX

Pop a range of data registers. Takes a register range `RRR.BBB` in the `X` registers. `RRR` is the first register in the range and `BBB` is the last register to pop.

11.6.15 STACKSZ

This returns the size of buffer stack to the `X` register. Pushing anything on the stack will increase this number. Popping something from the stack will make this number return to the same value as it was before the push-pop operation. Thus, this number can be used as a gauge to see if we are back to a previous point. It can also be used to see if things have been added to the stack or removed below a given point.

The actual number returned is the sum of the stack registers used by the buffer and the number of alpha register pushes that are on the stack. The two register buffer overhead is not included in this count. The means that an empty stack and a non-existing buffer stack both will return `0`.

11.6.16 TOPRTN?

Test if the top level record on the buffer stack is a return stack record. This can be used to control recursion to see when you have exhausted the return stacks pushed on the buffer stack.

To make this work in a reliable way, you should start by pushing something else on the stack first before you start recursion. If you have nothing you already pushed, you can push the `X` register using `PUSH X` to serve as a marker. When you are done, simply pop it off the stack. If you do not want to clobber `X` doing that, you can for example pop it to the `T` register instead (or the `Q` register if you are into synthetic programming and do not want to even disturb `T`).

Note: There are two ways this function can fail to work as intended. If the next record on the stack is the alpha register, it may be empty in which case this function will actually look at the next thing on the stack. Also, the test for whether the top element is a return stack record checks a magic number (`0x2ac` in the rightmost part). There is a (very) minor risk that what is pushed happens to contain that pattern and being something else. However, no normalized number has a bit pattern like this and `0xac` is not a normal letter.

11.6.17 CLSTBUF

Remove the stack buffer.

CHAPTER 12

Buffers

Buffers are blocks of private memory that modules can allocate for various private purposes. Buffers are allocated from the free memory pool and are located between the key assignments registers and the program memory. A buffer can be 1-255 registers in size. There are 16 buffers possible in the HP-41 design (0-15).

In addition to ordinary buffers, the OS4 module allows for a concept of hosted buffers. A hosted buffer resides inside the OS4 maintained system buffer (buffer number 15). Hosted buffers uses a number range (0-127) that is unrelated to the ordinary buffers.

The Boost module provides functions related to both ordinary and hosted buffers.

Note: Originally buffers were designed with I/O in mind and were often called I/O buffers. Later they were used for Time module alarms and the name was generalized to be just buffers.

12.1 Errors

If the buffer number specified is outside valid numeric range, 0-15 for normal buffers and 0-127 for hosted buffer, a `DATA ERROR` message is returned.

12.2 Functions

12.2.1 CLBUF

Removes a buffer specified by the X register (0-15).

12.2.2 CLHBUF

Removes a hosted buffer specified by the X register (0-127).

12.2.3 BUF?

Does the buffer (0-15) specified in the `X` register exist? If it does not exist, the next program line is skipped. In run-mode `YES` or `NO` is displayed.

12.2.4 HBUF?

Does the hosted buffer (0-127) specified in the `X` register exist? If it does not exist, the next program line is skipped. In run-mode `YES` or `NO` is displayed.

12.2.5 BUFSZ

Gives the size of the buffer (0-15) specified in the `X` register. The buffer number is saved in the `L` register and the size replaces the value in `X` register.

12.2.6 HBUFSZ

Gives the size of the hosted buffer (0-127) specified in the `X` register. The buffer number is saved in the `L` register and the size replaces the value in `X` register.

CHAPTER 13

Input functions

The standard HP-41 OS provides PSE as a way to wait for input and resume if not input is entered for about a second.

The Extended functions module add GETKEY that will busy wait for up to 10 seconds and finally the HP-41CX offers GETKEYX that is similar to GETKEY, but provides more control. Both these functions keeps the calculator at full power consumption while running.

The Boost module offers a couple of alternatives, some of which makes it possible wait or look for key input while the processor is powered down. The interval timer of the Time module is used for this, which means that they require the Time module to be present. The interval timer is normally used to provide the ticking clock display, but here we borrow it and use it as a time out.

13.1 Functions

13.1.1 PAUSE

This function works as the built-in PSE, but it is aware of OS4 application shells and will properly display the correct X register view as defined by the active shell. The built-in PSE does not do this and revert to showing X the old way, which may not be what you want when an application is active as it may alter the default view of the X register.

One example if the Ladybug module which replaces the standard view of X with integer values shown in the selected base. The default way of showing such numbers are as non-normalized numbers, which are essentially garbage. Using PAUSE when Ladybug may be active provides a pause function that works in the correct way.

13.2 Timed wait

DELAY and KEY are prompting functions with a value range of 1 to 10000, which corresponds to 0.1 to 1000 seconds. You will need to use register indirect arguments to access anything past 9.9 seconds (postfix argument 99).

13.2.1 DELAY _ _

The argument is the number of tenths of a second to wait. Example, `DELAY 15` will wait for 1.5 seconds. Pressing a key while waiting will terminate the timer and execution resumes on key release. The key press is otherwise ignored.

This also accept indirect arguments, e.g. `DELAY IND X` will read a value from the X register, divide by `10` and wait for that number of seconds.

13.2.2 KEY _ _

Works similar as `DELAY`, but will also return the key code of the pressed key to X. If no key was pressed, `0` is returned.

13.2.3 Y/N?

Simple test for yes or no input. Beeps and waits for up to 25 seconds for input. Executes the next line if `Y` is pressed, skips the next line in `N` is pressed.

Active keys are:

ON turns the calculator off.

Y resumes program execution at next program line.

N skips one program line and resumes execution.

R/S stops the program.

<- stops the program.

CHAPTER 14

Pseudo random numbers

The base operating system in the HP-41 lacks a pseudo random number generator. The Boost module provides an MCODE version of the one that is available in the Game application module.

The seed is stored in a hosted buffer offered by OS4, not competing with any other I/O buffer. The hosted buffer number used is 0.

Initialization of the seed is based on the current time offered by the Time module, if present.

14.1 Functions

14.1.1 RNDM

Provide a pseudo random number in the range 0 to 0.999999.

14.1.2 SEED

Use the fractional part of the value in X as the seed for the random number generator. This is useful to get a predictable range of pseudo random numbers, e.g. useful for test or demo purposes.

14.1.3 2D6

Simulate rolling two common six sided dices, giving a value 2-12 with a distribution that corresponds to throwing two dices. The value 7 will the most common result, while 2 and 12 are the most uncommon outcomes.

CHAPTER 15

Extended memory

Boost provides some additional functions related to extended memory as implemented on the HP-41CX and HP-41CL. Currently the separate 82180A Extended Functions as used by the HP-41C and HP-41CV is not supported.

Note: Large extended memory as optionally available on the HP-41CL using a custom firmare is fully supported by all these functions, as they use the official entry points.

15.1 Random data file access

These functions make it possible to treat data files in extended memory much like ordinary registers. A set of functions corresponding to the existing ones such as `RCL` and `STO` are provided. The names start with an `X` and they prompt for the register to access. Indirection is also possible, but the indirection register acts on the ordinary data registers, or stack registers.

Possible errors when using these functions are:

NO 41CX OS if there is no 41CX style operating system provided by the calculator.

FL NOT FOUND if not current file is selected.

FL TYPE ERR if current file exists, but is not a data file.

NONEXISTENT if attempt is made to access a register beyond the end of file.

Selecting a file can be done using the `SEEKPTA` function which takes the name from the alpha register and the pointer to set from X. When using these function you are bypassing the pointer mechanism and instead access arbitrary registers just as you would do with ordinary data storage registers.

15.1.1 XRCL _ _

Prompting function that reads directly from a data file. `XRCL 03` will recall the fourth register in the current data file to X. `XRCL IND 04` will read data register 04 (the normal data register, not register 04 in the data file).

The value in that data file register tells which register to recall from the current data file to the X register.

15.1.2 XSTO _ _

Prompting function that writes directly to a data file. `XSTO 03` will store the contents in the X register in the fourth register in the current data file. `STO IND Z` will store the value in the X register into the current data file. The register number that is written to is in stack register Z. If the indirect register number is numeric, e.g. `STO IND 10`, the indirection register is the normal data register (not the data file register).

15.1.3 XVIEW _ _

Prompting function that reads directly from a data file. `XVIEW 03` will read the fourth register in the current data file and show it in the display. `XVIEW IND 04` will read data register 04 (the normal data register, not register 04 in the data file). The value in that register tells which register number to fetch read the current data file and show in the display.

15.1.4 XARCL _ _

Prompting function that reads directly from a data file. `XARCL 03` will recall the fourth register in the current data file and append the value in the alpha register.

15.1.5 <>X _ _

Dual prompting function that takes one normal register argument and a register value for extended memory. Swaps the two values between the two indicated registers.

Register indirect arguments are permitted on both sides. A stack argument is permitted for the left hand side, but not for right hand side as it is extended memory.

Note: The name here is selected so that the X appears after the exchange name. This is for two reasons. First, there is already a built-in function X<> which takes one argument and exchanges between the X register and the argument. Second, the X appears after the <> to indicate that the Extended memory register operand comes second.

15.2 File operations

Function related to files in extended memory.

15.2.1 WORKFL

This function appends the name of the current active file to the alpha register.

Possible errors are:

NO 41CX OS if there is no 41CX style operating system provided by the calculator.

FL NOT FOUND if there is no active file.

15.2.2 RENFL

Rename a file in extended memory. The file to be renamed are in the alpha register followed by a comma and after that the new name.

Possible errors are:

NO 41CX OS if there is no 41CX style operating system provided by the calculator.

FL NOT FOUND if the file does not exist.

DATA ERROR if there is no comma

CHAPTER 16

Miscellaneous

In this chapter various functions that do not belong to any particular category are documented.

16.1 Functions

16.1.1 AVAIL

This function returns the number of free registers to the X register. This is the same number that you would see when standing at end of program memory. In the past this function has sometimes been called FREE?, but it has been renamed here to make it less confusing. The question mark often means that we optionally skip a program step.

16.1.2 COMPILE

Compile all branches in the entire program memory. This non-programmable function will walk through all programs and compile all local GTO and XEQ functions. A short GTO that is out of range will get converted to the corresponding long version so that it can be compiled.

While scanning for GTO and XEQ and their destination LBL the display shows WORKING. If a short GTO is converted to a long version, the insertion will cause nulls to be inserted. To get rid of these the program memory is immediately packed. Thus, during operation you will see the display alternating between WORKING and PACKING.

If COMPILE runs out of memory, the usual TRY AGAIN message is shown.

Execution time varies wildly for this function. For a very large single program that takes up almost all memory, it may take 10 minutes to complete the compilation.

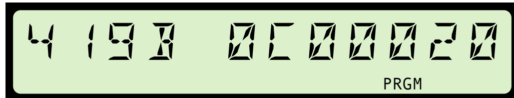
The main purpose of COMPILE is to avoid the initial slow execution of programs that are not compiled. That is, you have a situation where you have time in advance and want to invest that in making your programs run as fast as possible when you later use them. A typical use is you load up your calculator with software prior to an an exam that is time constrained.

16.1.3 RAMED

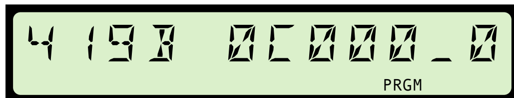
This is a RAM editor which allows you to examine and alter bytes (actually hex digits) in the RAM memory of your calculator.

> **Warning:** This RAM editor allows you to edit any RAM memory as you wish. Be very sure that you understand what you are doing as doing it wrong can compromise the memory structure and lead to a `MEMORY LOST`.

Once started you may see a display like the following:



As there is a blinking cursor it will alternate with the following view:



The leftmost digit is the digit position in the current register, which range from 0 to 13 (shown in hex as `0–D`). The digit position in the example is `4` and is followed by the current address which in the example is `19B` (hex). Part of the register is shown to the right with the current digit position highlighted with a blinking cursor.

Keys of operation are as follows:

R/S terminate `RAMED`.

ON turn the HP-41 off.

+ move to next register.

- move to previous register.

PRGM move cursor right.

USER move cursor left.

0-F alter the digit where the cursor is.

. toggle cursor field between the address and the data.

The HP-41 will give a short beep if you move to cursor so that it wraps around between the last and first digit in the register.

`RAMED` is non-programmable and if invoked in program mode it will use the current program location as the start address.

When started outside program mode the start address is taken from the `X` register. This can either be a decimal address or a right justified binary value (non-normalized number) in `X`.

> **Note:** When used together with the Ladybug module, simply enter the address in `X` and start `RAMED`. An integer value is actually just right justified binary value (non-normalized number).

> **Note:** The reason for `RAMED` to be non-programmable is that is natural to start editing program memory at the current location when inside program mode. If you place `RAMED` inside a program (there are several ways

of how this can be done), then RAMED will start from the address in the X register. When the user presses R/S to leave RAMED, program execution resumes.

16.1.4 APX

This function makes it possible to append to the number in X register. You can see this as a counterpart of the append function in alpha mode.

In the book *Extend your HP-41* there is a discussion of this function and some motivation of why it is useful on page 541, followed by a synthetic program on page 542. The APX function provided here is an MCODE version of this program and works mostly the same.

Somewhat simplified, APX takes the number in X and feeds it into the digit entry mechanism, then tells the system that we are still doing numeric entry.

It can be used quite naturally if assigned to the same place as alpha append (shifted ASN key), making it appear on the corresponding place on the user keyboard. This has the downside that you can only reach the ASN function outside USER mode.

APX also works from inside a program. However, it needs to be followed by STOP or PSE in order to let the user append to the number. When stopped from a program with ALPHA on, it acts as alpha append instead. Thus, APX gives you a programmable alpha append as a bonus.

APX favors editing the mantissa. When given a very large or small number APX will attempt to bring the number into what can be shown without an exponent. Well behaved numbers will have the correct sign and decimal point in the correct location.

16.1.5 LUHN?

Implements the Luhn algorithm as used by credit card numbers. Accepts a two-part BCD number in Y and X. The lower 14 digits are expected in X and any upper digits are in Y. A typical credit card number uses 16 digits.

To enter the number, you can use the usual CODE function, but it is probably easiest to just key it in using the Ladybug module, with a setting of 56-bits word size and hex mode:

```
WSIZE 56
HEX
3432_ H
ENTER
5422395239434_ H
LUHN?
```

LUHN? will skip next line if the Luhn checksum is not correct. In keyboard mode it will display YES for a correct Luhn number and NO otherwise.

Reference: https://en.wikipedia.org/wiki/Luhn_algorithm

16.1.6 CODE

This is the ubiquitous CODE function used to encode a non-normalized number based on a hexadecimal value in the alpha register. The resulting value is put in the X register.

As with DECODE, you may want to look into the Ladybug module. The CODE and DECODE are included for completeness in the case when you do not have Ladybug module inserted in your HP-41.

16.1.7 DECODE

This is the ubiquitous `DECODE` function used to decode the number in `X` and put its hexadecimal value in the alpha register. This was often used in the days of synthetic programming to make sense of the non-normalized numbers that often resulted.

When used from a running program mode the hexadecimal string is appended to the alpha register. When used from the keyboard the alpha register is cleared first.

Note: If you are into fiddling with register values, it can be worth checking out the Ladybug module which makes working with such numbers as easy as working with normal decimal numbers. Just configure it in hex mode with word size 56 for the ultimate way of working with binary (non-normalized) numbers on the HP-41. In addition Ladybug makes a great replacement for an HP-16C.

16.1.8 CTRST

Sets the contrast value for the later Half-nut style displays. Takes a value 0–15 from the `X` register.

16.1.9 CTRST?

Reads the current contrast value 0–15 and puts it in the `X` register. This works for later Half-nut style displays.

Index

Non-alphabetical

2D6 function, 35
<>X, 38

A

alpha functions, 18, 19
append to X, 43
APX, 43
ARCLINT, 21
assignment
    auto, 12
    by name, 11
    of function code, 11
    rebuild bitmaps, 13
    XROM identity, 11
assignments
    loading, 12
ATOXR, 21
auto-assignments, 12
available memory, 41

B

buffer, 1
buffer stack, 23
    depth, 28
buffers, 29

C

checksum
    Luhs, 43
CLKYSEC, 12
Clonix, 1
CODE, 43
compare functions, 16
COMPILE, 41

D

data file access
    <>X, 38
    XARCL, 38
    XRCL, 37
    XSTO, 38
    XVIEW, 38
decode NNN, 43
decrement, 19
DELAY function, 33
display
    contrast, 43

E

edit memory, 42
encode NNN, 43
extended memory, 35

F

filename
    active, 38
fix/end mode, 16
free memory, 41

H

Half-nut display
    contrast, 43

I

I/O buffer, 1
I/O buffers, 29
increment, 19
input functions, 32

K

KEY function, 34

L

LKAOFF, 12
LKAON, 12
Luhn checksum, 43

M

mantissa
    view, 16
MAPKEYS, 13
memory
    available, 41

N

name of active file, 38
NoV modules, 1

P

pop

- alpha register, 27
- data registers, 28
- flags, 27
- register, 27
- return stack, 27
- RPN stack, 27, 28

program control
- PC<>RTN, 23
- RTN?, 23
- RTNS, 23
- XEQ>GTO, 23

program memory
- compile, 41

push
- alpha register, 27
- data registers, 28
- flags, 27
- register, 26
- return stack, 27
- RPN stack, 27

R

RAMED, 42

random mnumber
- seed, 35

random number, 35

rename file, 39

S

secondary functions, 5

stack buffer, 23
- depth, 28

system shell, 5

V

VMANT function, 16

X

XARCL, 38

XRCL, 37

XROM number, 1

XSTO, 38

XTOAL, 21

XVIEW, 38

Y

Yes/no input, 34