# HP-41 Advanced Developer Notes

| Advanced Modules Architectural Details | | | | | | |
|---|---|---|---|---|---|---|
| **Module Name** | **XROM#** | **Lib#4** | **Sub-Fncs** | **Banked** | **SVC_IO** | **Buffers** |
| 16C Emulator | 16 | √ | √ | √ | - | #9; #11 |
| 41Z Deluxe | 01, 04 | √ | √ | √ | - | #8; #9 |
| ALPHA_44 | 06 | √ | - | - | - | - |
| AMC_OS/X | 05 | √ | - | √ | √ | #5 |
| CLXMEM | 20 | √ | √ | - | - | #9 |
| Cryptography | 10 | √ | - | - | - | #9 |
| Elliptics | 17 | √ | - | - | - | - |
| GJM ROM | 31 | √ | √ | √ | - | #9 |
| HEPAX 4H | 06 | √ | √ | √ | - | #9 |
| PowerCL | 12 | √ | √ | √ | - | #9 |
| PWCL_EXT | 12 | √ | √ | √ | - | #9 |
| RAMPage | 17 | √ | - | | - | - |
| SandMath | 02, 03 | √ | √ | √ | √ | #5; #9; #14 |
| SandMatrix & Vector Calc | 22, 24 | √ | √ | √ | - | #5; #9 |
| Series & Sums | 18 | √ | - | - | - | - |
| ToolBox | 13 | √ | - | - | - | - |
| TotalRekall / UMS | 21 | √ | - | - | √ | - |
| AECROM | 18 | - | - | - | √ | - |
| ZENROM | 05 | - | - | - | √ | - |
| HEPAX 1D | 06 | - | - | √ | - | - |
| CCD MODULE | 09, 11 | - | - | - | √ | #5 |
| HP41 Advantage | 22, 24 | - | - | √ | √ | #5, #14 |

# LASTF: The "Last Function" – At Long last!

The latest releases of the PowerCL, 41Z, SandMath and SandMatrix modules all include support for the "LASTF" functionality. This is a handy choice for repeat executions of the same function (i.e. to execute again the last-executed function), without having to type its name or navigate the different launchers to access it.

The implementation is not universal – it only covers functions invoked using the dedicated launchers, but not those called using the mainframe XEQ function. It does however support three scenarios:

a. Main Functions in the module main FATs
b. Main Functions from any other module called using the launchers
c. *Sub-functions from the auxiliary FATs*.

Because functions from the latter group cannot be assigned to a key in the user keyboard, the LASTF solution is especially useful in this case.

The following table summarizes all launchers that include this feature:

| Module | Launchers | LASTF Method |
|---|---|---|
| **PWRCL_EXT** revision "O" | **ΣCL**, XCAT, TURBO, BAUD, MMU, IMDB, HEPX, YBUF, XXEQ, PLUG | Captures (sub)fnc id# |
| | IOBUS, IOPG#, STR$, ALP$ | Captures (sub)fnc id# |
| | **XQ1 _ , XQ2 _** | Captures (sub)fnc NAME |
| | **XFAT _ _:_ _** | Captures (sub)fnc id# |
| | **FCAT1/2 (XEQ)** | Captures (sub)fnc id# |
| **SandMath 4x4** revision "M" | **ΣFL**, HYP, FRC, RCL# | Captures (sub)fnc id# |
| | **ΣF$ _** | Captures (sub)fnc NAME |
| | **ΣF# _ _ _** | Captures (sub)fnc id# |
| | **FCAT (XEQ)** | Captures (sub)fnc id# |
| **SandMatrix 4** revision "M" | **ΣML**, (M: / V: / P:), ΣDST/ΣPRT | Captures (sub)fnc id# |
| | **ΣV$ _** | Captures (sub)fnc NAME |
| | **ΣV# _ _ _** | Captures (sub)fnc id# |
| | **FCAT (XEQ)** | Captures (sub)fnc id# |
| **16C Emulator** | **16C ,** ΣLEFT /ΣRGHT, ΣMOD, ΣBIT | Captures (sub)fnc id# |
| | **16$ _** | Captures (sub)fnc NAME |
| | **16# _ _ _** | Captures (sub)fnc id# |
| | **FCAT (XEQ)** | Captures (sub)fnc id# |
| **CL-XMem** | **YMEM**, STKSWP | Captures (sub)fnc id# |
| | **SF$ _** | Captures (sub)fnc NAME |
| | **SF# _ _ _** | Captures (sub)fnc id# |
| | **SCAT (XEQ)** | Captures (sub)fnc id# |
| **41Z "Deluxe"** | **ZKBRD _** | Captures (sub)fnc id# |
| | ZHGF, ZPRT, ZNEXT, ZBSL, ZHYP | Captures fnc id# |
| | **ZF$ _** | Captures (sub)fnc NAME |
| | **ZF# _ _ _** | Captures (sub)fnc id# |
| | **ZCAT (XEQ)** | Captures (sub)fnc id# |

For consistency sake, the main launchers (**ΣCL**, **ΣFL**, **ΣML**, **16C**, **YMEM, ZKBRD**) connect to their corresponding Alphabetical launchers ('$') using the ALPHA key, and with the index launchers ('#') using the PRGM key.

Note that the Alphabetical launchers (**XQ1**, **XQ2**, **ΣF$, ΣV$, 16$, SF$, ZF$**) will switch to ALPHA mode automatically. Spelling the function name is terminated pressing ALPHA, which will either execute the function (in RUN mode) or enter it using two program steps in PRGM mode by means of the associated numeric launcher (**XFAT**, **ΣF#**, **ΣV#, 16#, SF#, ZF#**) plus the corresponding index (using the so-called non-merged approach). These conversions are done automatically.

And rounding it all up, the LASTF operation is also supported by the **FCAT** functions when the execution is triggered during the sub-function enumeration – using the [XEQ] hot key. This was added to the implementation with revision "M" of the Library#4.

Another new enhancement is the displaying of the sub-function names (up to six characters only) when using the index-based launchers **XFAT**, **ΣF#**, **ΣV#**, **16C**, **SF#**, and Z**F#** - which provides visual feedback that the chosen function is the intended one (or not). This feature is active in RUN mode and when entering it into a program.

For ultimate convenience, the display of sub-function names also occurs when *single-stepping a program* – but obviously not so during the program execution. This is the closest it gets to managing sub-functions as close as possible to standard functions in the main FATs.

## LASTF Operating Instructions

The Last Function feature is triggered by pressing the radix key (decimal point - the same key used by LastX) while the launcher prompt is up. This is consistently implemented across all launchers supporting the functionality in the three modules – they all work the same way.

When this feature is invoked, it first shows "LASTF" briefly in the display, quickly followed by the last-function name. Keeping the key depressed for a while shows "NULL" and cancels the action. In RUN mode the function is executed, and in PRGM mode it's added as two program steps if programmable, or directly executed if not programmable.

If no last-function yet exists, the error message "NO LASTF" is shown. If the buffer #9 is not present, the error message is "NO BUF" instead.


## LASTF Implementation details.

The functionality is a two-step process: a first one to capture the function id#, and a second that retrieves it, shows the function name, and finally parses it. All launchers have been enhanced to store the appropriate function information (either index codes or full names) in registers within a dedicated buffer, with id# = 9.

The buffer is maintained automatically by the modules (created if not present when the calculator is switched ON), and its contents are preserved while it is turned off (during "deep sleep"). No user interaction is required.

The buffer reserves one register for each of the five modules that have sub-functions, plus the header also stores the standard-41Z last-function - which is always in its main FAT. Therefore up to five last-functions can be simultaneously available at any given time. A total of five registers are used, as follows:

| Register | Used for: |
|---|---|
| b5 | CL-XMem - GJM ROM |
| b4 | SandMatrix fcn id# |
| b3 | SandMath fcn id# |
| b2 | PowerCL fcn id# |
| b1 | 41Z Deluxe / 16C Emulator |
| b0 | Header – Standard 41Z Module |

For modules other than the Standard 41Z, the function id# format can be either a string of alpha characters (stored in reversed order) representing the sub-function name, or a three-digit hex value in the [S&X] field representing the main/sub-function index. In the latter case the [MS] field is marked with an "F" to tell the cases apart.

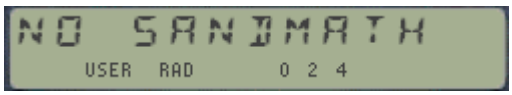
When the LASTF action is triggered pressing the Radix key, the code seeks for the function data in the relevant buffer register, depending on the module carrying the action. When found, it displays the sub-function name (either by mirror-imaging the alpha id# or by looking it up in the corresponding auxiliary FAT. Finally it's parsed to the corresponding section responsible for the execution of the sub-functions.

### Dependencies

These modules do require the 41-CX OS and the Library#4 installed. Note that the Library#4 was also updated to revision "M" – now including subroutines in support of this functionality. As always, make sure that matching revisions are installed in your machine.



Note as well that some of the functions in the SandMatrix also require the SandMath present. This is implemented a non-halting warning as well.



## Routines Documentation.

This section documents the routines in the Library#4 module and in the specific modules that provide the LastF functionality. As it turns out, I realized I had forgotten most of the secret souce ingredientes during the development of the 16C Emulator Module – writing the LASTF for that one was an exercise of re-discovery all over again!

Phase 1.- Capturing the function info into the LastF buffer.

This is typically done just when the function code is issued to the sub-function launcher, even before the tables search. This task is accomplished by routines [LASTF#] and [LASTF$] depending on the case: either using the sub-function index stored in M[S&X] or its name mirror-imaged in regster Q. Another input parameter is the buffer reg#, assumed to be in C[S&X] at the call.

Phase 2.- Executing the Last function from fcn code in LastF Buffer.

Routine [LASTF?] is usually inserted in the partial key sequence as triggered by the radix key. It first selects RAM chip0 and the partial entry sequence with a call to [EXIT4], and then retrieves the function code stored in the corresponding register of the LastF buffer using [LASTF4] and the buffer reg# in C[S&X].

a. If it is a function index the execution is passed to [SUBXEQ] within the numeric subfunction launcher, like it's the case for every other sub-function triggered otherwise by a custom keyboard assignment. *This requires CPU flag 8 cleared*.

   Depending on the [XS] field the parser knows whether it's a main function or a sub-function from a secondary FAT. The convention here may be different for each module, but conceptually it's the same across all of them. In the former case, the function name is displyed using the OS routine as 0x2F52 (unlabeled), a pause is introduced calling [WAIT4L] in the library#4, anf the execution is transferred to [RESET2] = which does the housekeeping chores and finally calls [RAK70] to execute the function.

   If however it is a sub-function code then displaying the function name needs to use routine [FNAME] in the Library#4 instead. Its input parameter is the address within the secondary FAT, which is calculated by another subroutine in the Library#, [GETADR], as as a direct function of its index number: adr = table top + 2x (1 + index#). [GETADR] will fetch the address bytes from the sub-FAT structure much the same as the CATALOG routines do it on the main FATs.

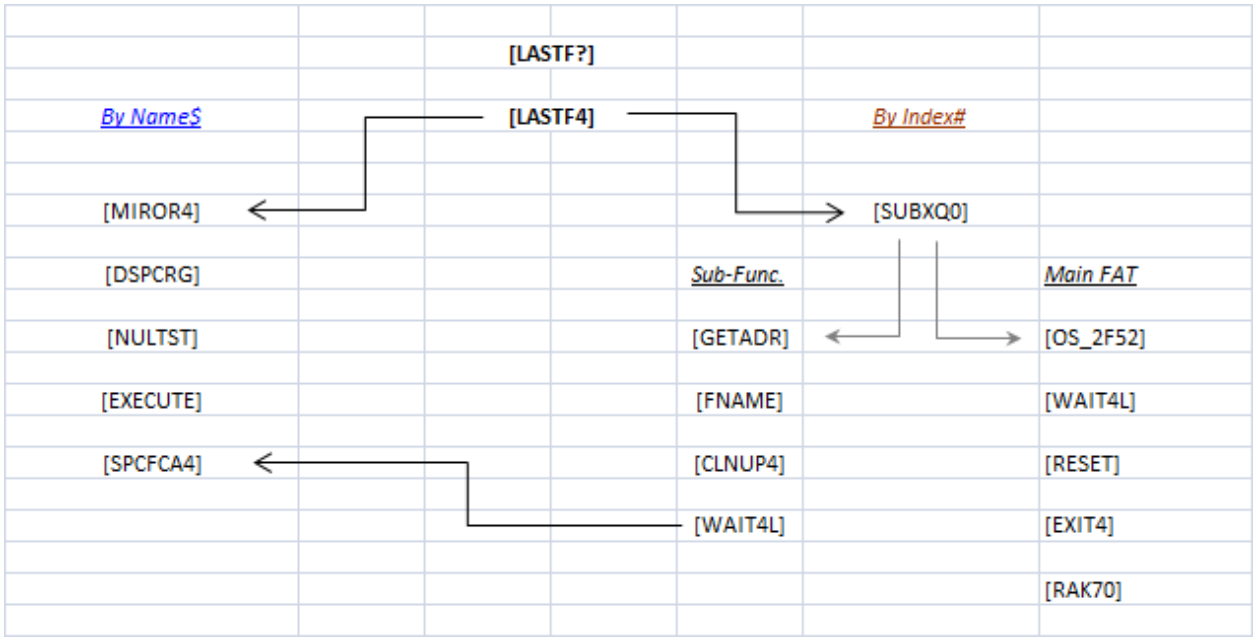
b. If it is a string with the function name, the string is put in the Q register and mirror-imaged in C, in a format understood by [DSPCREG]. The name is shown in the display and after a chance to null it using [NULTST] the execution goes to [EXECUTE] within the Alpha-Subfunction launcher, *which sets CPU flag 8*.

Either way (by index or by name) the execution is sent to the library#4 routines [HEPXA4] or [SPCFCA4] located at 0x4E25 and 0x4E28 respectively in the Library#4. The difference is whether the auxiliary FAT is at the top of the page - for [HEPAXA4] - or starting at address 0X800 - for [SPCFA4]).

These routines will triage the situation depending on whether the machine is in program mode, running a program or in manual mode – so the sub-function will be entered as two program lines or executed directly. (with the sub-function address in hand it's easy to just send the program pointer to the right place). Note that if program mode is on then the subfunction index is used instead of its name in all instances – even when the input parameter was its name.

For this to work across modules from the Library#4 the routine needs to know the sub-function launcher code, which is different for each module. This is entered as input parameter in the leftmost 4 digits of register 8(P).

The figure below sketches the routine flow. Note the fundamental role of the [SPFCA4] routine, final step of all possible execution branches dealing with sub-functions. Note as well that there is a lot of code reuse between LASTF and the sub-function launchers, as basically both perform the same thing.



## Auxiliary FAT structure.

The Auxiliary FAT can be placed either in the main bank or in a secondary bank. Within the page it can be located starting at the top of the page (0xp000) or at the beginning of the third quad (0xp800). The first and last words are always 100, which serves as control for the FCAT enumeration in forwards or backwards mode. The second word on the table must be the number of sub-functions – which unlike the standard main FATs, is not limited to 64. A practical limit of 99 exists, although it is possible to exceed it even if the sub-function names won't display properly.

Sub-functions are listed in the Auxiliary FAT using two words per function, with the start address coded in the same way it is arranged for the main FATs of a module. There is no provision made for prompting functions, nor for non-programmable ones.

The first sub-function should always have a hyphen as first character, denoting a section header. The FCAT function code needs that at least another section header function exists for the section Catalog feature to work (pressing ENTER^ with the catalog enumeration stopped).

## Sub-function Launcher design.

There are two sub-function launchers on each module: one uses the sub-function index as parameter, while the other uses the sub-function name.

The by-index implementation takes the index from the prompt, and uses it to basically read the function start address from the auxiliary FAT and parse the execution to that address if in RUN mode; or to enter the sub-function in a program using two program lines – following the non-merged approach design. When it is used with program mode on or SST'ing a program line it will show the sub-function name briefly in the display to provide visual feedback to the user.

The by-name implementation takes the name string from the Q register and proceeds to do a search by name on the auxiliary FAT. This alphabetical search is done in the library#4 routine [SPCFA4], and thus it requires that the corresponding bank with the sub-function FAT is enabled. When a match is found then the function start address is used as described above. Note that when program mode is active the alphabetical launcher will use the sub-function index and the numeric launcher as well.

If the sub-function is not found in the auxiliary FATs (there are two of those in the Power CL module), the code will attempt a standard function search in the module's main FAT – or in any other module currently plugged in. This is done by the library#4 routine [XEQFNC], which basically uses the OS routines [ARSCH] and [RAK70] for that purpose.

The sub-function launchers are also responsible for adding the sub-function information in the LASTF buffer – so it can be re-executed using the LastF functionality. This is typically done by calling routines [LASTF#] or [LASTF$] for the numerical and alphabetical launchers respectively – using the buffer register as input parameter.

## The Sub-Function Catalogs. { FCAT }

An important feature is the ability to enumerate all sub-functions within the auxiliary FATs. Similar to the Catalog 2 in the OS, the listing can be stopped and resumed using the R/S key. FCAT also allows for a single-step enumeration and for a section listing – showing only functions with a hyphen in the first character of their names.

In the SST mode the function shown can be executed pressing the XEQ key. This is accomplished by back-calculating the sub-function index, getting the sub-function start address from it, and and parsing the execution to that address in the same way as described above. The sub-function index is also added to the LastF buffer during that process as well.

The code will also read the sub-function name and present it to the user as visual feedback while pressing the XEQ control key. The message "XEQ 'FNAME" is shown briefly before the function is nulled if the key is kept depressed.

Note that the sub-function enumeration uses a continuous loop approach, even when stopped in manual mode - detecting key pressings as opposed to the partial entry sequence technique. Even if this is a less preferable option due to the battery usage associated with it, it is however necessary in this case because the code will be accessing auxiliary banks – which is not compatible with the partial entry sequence design.