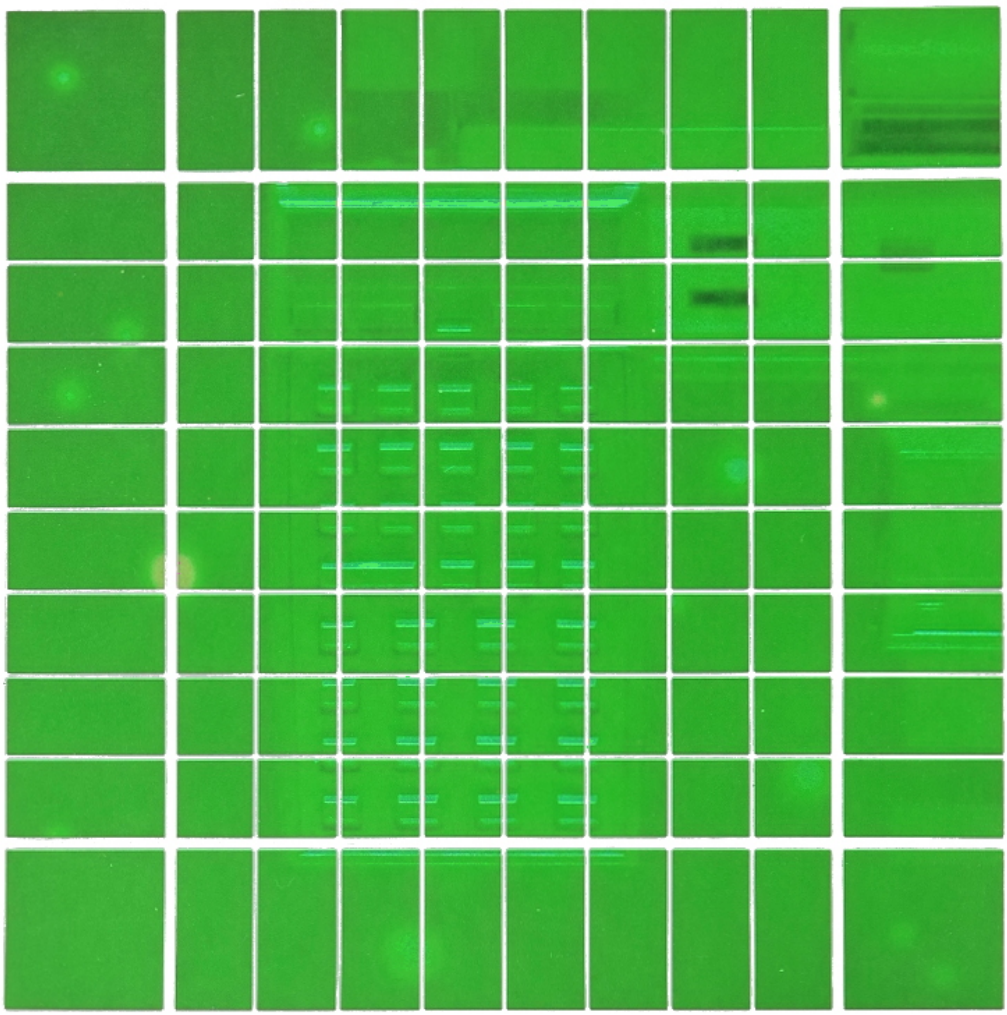
OS4



OS4 Documentation

Release 2A

Håkan Thörngren

Nov 20, 2020

Contents

CHAPTER 1

Introduction

Welcome to the OS4 module for the HP-41 calculator! OS4 is a powerful extension module which removes several limitations that were originally designed into the HP-41 mainframe (the name used by HP for the HP-41 firmware).

OS4 is a support module and requires other extension modules to expose its capabilities to the user. Examples of such modules are Ladybug and Boost. You can also use OS4 in your own module projects.

At a very high level OS4 does the following:

1. Removes the 64 function limitation for a single plug-in module page.
2. Support for programmable prompting XROM functions with one or two postfix arguments.
3. Provides a robust way of defining new alternative keyboard layouts, supporting both full and partial layouts without relying on key assignments.
4. A way to override the default display of the X register with something else.
5. Support for writing temporary modes, such as input modes, updating clock style modes and catalogs. The interval timer can be borrowed (if timer chip present) to allow for periodic updates and timeouts in such modes.

1.1 Plug-in module

OS4 is a module image file that needs to be loaded in some programmable plug-in module hardware. This can be a Clonix module, an MLDL or the HP-41CL. It is also possible to use OS4 on HP-41 emulators.

The OS4 image is a 2x4K module. Two banks occupies a single 4K page and it must be loaded to address page 4.

1.2 Resource requirements

OS4 allocates some space from the free memory pool. How much is taken depends on what you actually do with it, but expect 2-7 registers for modest use.

Apart from this, OS4 does not impose any restrictions on the environment and will run comfortable on any HP-41C, HP-41CV, HP-41CX or HP-41CL.

There is no XROM number used by this module as it is in page 4.

1.3 Using this guide

If you only want to use OS4 because another module requires it, simply load the module image to page 4 and put this guide aside.

This guide is aimed to MCODE developers that want to explore and develop modules using OS4. It can also be of interest to those that want to know how it works internally or want to study how to write HP-41 MCODE.

This guide assumes that you have a working knowledge of:

- The HP-41 calculator.
- Understanding of low level programming.

1.4 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
- *MCODE Programming for Beginners*, Ken Emery, 1985
- *Extend your HP-41*, W Mier-Jedrzejowicz, 1985
- NutStudio Tools Documentation
- A programmers handbook, Poul Kaarup
- The VASM listings (annotated mainframe source code)

1.4.1 Document conventions

Code examples are shown in lower case as the author feels it somewhat more relaxing to the eyes. If MCODE instructions are discussed in the text they are in upper case to make them stand out more.

The development tools used in this guide is NutStudio tools which contains a macro assembler, a linker, a librarian and a debugger. The debugger includes a simulator and a comprehensive scripting abilities. The assembler and linker fully supports generating banked modules and can generate ELF/DWARF images to allow for source level debugging. This has been a very useful tool chain in developing and testing OS4. VisualStudio Code has been used as the a main debugger front-end (user interface).

1.5 Acknowledgments

Some of the code sequences used in OS4 have been borrowed from, or is based on source code found elsewhere. This has been done in good faith as the routines are very much needed for this project. No permissions for this have been asked for, or been granted by the original authors or copyright owners.

Part of the code is based on code that is copyright by Hewlett Packard Company. This includes code used for the HP-41 power on sequence, the search routine in function address tables (`ASRCH`), the code that deals with the assignment bitmap. Also the key dispatch routine `keyDispatch` (originally named `KEY-FC`) comes from the Time module and the buffer search routine borrows ideas from the Time module. The `mapAssignments`

function (originally called RSTKCA) comes from the Extended Functions module with additions presents in the Card Reader ROM. The range function GTINDX and friends codes from the Extended functions module (also part of HP-41CX).

The XBCDBIN routine to convert a floating point number number to binary (similar to BCDBIN found in mainframe, but with more useful range) is written by Ken Emery, reference PPCCJ V11N5P6.

1.6 License

CHAPTER 2

Installation

The OS4 image consists of two banks that occupy a single 4K page and it must be loaded to address page 4.

How this is done depends on your hardware or emulator.

The module emulator you use must support a banked page 4 and also allow for independent bank switching of page groups. The HP-41CL and MLDL-2000 support this. Other MLDLs and module emulation hardware from long ago will probably not be able to support OS4.

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.

Note: The hardware you use to load the OS4 module must be capable of reacting very fast and serve instruction fetches within 4 cycles after wake-up. The MLDL-2000 has shown certain problems in this area when powered by the HP-41. It seems to work when powered over USB, that is, being plugged into a computer in the other end. A hardware modification to the MLDL-2000 may be needed to make it work reliably with OS4 when power by the HP-41 alone.

2.1 Programming

To make it easier to develop modules using OS4 the `OS4.h` header file exists. This is provided in the source distribution along with the fill source code. It is also available in NutStudio tools distribution. This single header file provides definitions of all entry points and useful definitions. Include it in your source file and you are good to go:

```
#include "OS4.h"
```

2.2 Version check

To allow future changes to OS4 there is a mechanism for checking if the plugged in OS4 module is compatible with your own module. The checkApiVersionOS4 routine should be called from the deep wake up poll vector entry:

```
deepWake:   n=c                 ; N= ROMCHK state
            ldi     0x000       ; I need major version 0 at least
            gosub   checkApiVersionOS4
```

The number passed to checkApiVersionOS4 is the required version of OS4. You can find the current version number used in such context early in the OS4.h header file.

This version number is API related and is not the same as the version of the OS4 release.

You can either load the version number defined in OS4.h, or better, the API version your software requires.

Note: If you use the ApiVersionOS4 constant defined in OS4.h, then your software will automatically demand a later version of OS4 if you upgrade OS4, even if you do not actually use any of the additional features provided with that version.

This API version number consists of two parts. The lower 8 bits are an increasing number that is bumped whenever new entry points are added, or additional things are provided by existing entry points. The upper 4 bits are a major version number that must match precisely. Changing the major number means that anything may have been moved or altered (except the version check itself).

Note: If the version is old, OLD OS4 is displayed as an error message. This may have the consequence that poll entries are skipped, but the calculator may still work (to some degree). However, anything may be broken and the user should hopefully understand that the current configuration should not be used.

CHAPTER 3

Overview

OS4 is intended to be a foundation for writing powerful new MCODE modules for the HP-41.

3.1 Page 4 module

OS4 must be loaded to page 4 to work. The reason it requires page 4 is that OS4 needs to have access to the take over vector at address `0x4000`, which was originally intended for diagnostic ROMs.

OS4 uses this vector to wedge itself into the existing operating system. It is unfortunately impossible to do something like OS4 without using this vector, as a poll vector is not powerful enough.

This means that OS4 and modules depending on it are incompatible with anything else that requires page 4. This includes:

1. Diagnostic ROMs. These are take-over ROMs that will not give control back to normal operation.
2. Do not disable the printer. When a printer is disabled by using the switch it relocates itself at page 4. This will cause bus conflicts if OS4 is present.
3. Modules using Library#4. This is a popular library that resides in page 4 is incompatible with OS4. This also includes any modules that rely on Library#4.

Page 4 has only a single (officially) defined entry point and that is its first address `0x4000`. This address is called whenever the HP-41 wakes up (no matter if it is deep or light sleep) and when it is about to go back to light sleep again. The normal case is that page 4 is empty, in which case it executes as NOP which will cause an immediate return if called with a `GOSUB` instruction. If OS4 is loaded there is actual code here that takes control. It is not a complete take over, as it reuses as much as possible of the already existing built-in functionality.

There are no poll vectors at the top of page 4 as in normal plug-in modules.

3.2 Shells

A new concept provided is a *shell* which provides a way of installing new keyboard and display behaviors. Such behaviors can be activated by the user to turn the calculator into something different compared to the

default. This transformation can be anything from mild to very dramatic.

A shell keyboard definition changes the standard keyboard and is also active outside user mode. In user mode it is possible to make assignments as usual and also have the scan for local alpha labels in the current program of the two top key rows. A shell may specify if it wants the scan for local alpha labels in the current RPN program, or not.

Shells make it possible to replace the standard keyboard with something different while preserving all other fundamental behavior, which makes the HP-41 still feel like an HP-41, only different.

3.3 Message system

A new internal message handler system is also provided. In some way it can be seen as somewhat related to poll vectors, but it is more flexible and more dynamic. Message handler descriptors are stored in the same stack as the shell descriptors.

It is possible for a message originator to send (broadcast) messages to anyone that are interested in a particular event. It also allows for collaboration between the message originator and the receivers. Messages can be about anything of interest and is in no way tied to the keyboard or display.

The message system can be used to notify such things as that a new command is keyed in (which should clear the RPN return stack), or that the `CAT` function was entered with a catalog number that is not known. A message can then be sent to ask if there are anyone who wants to act on it, e.g. `CAT 23`. Any plug-in module can register a handler and act on any message.

The main software repository contains a list of known messages and this list can be added to as needed.

3.4 Secondary FAT

A 4K module page is limited to 64 functions. These are stored at the beginning of a page in a function address table (FAT). 64 functions may have been an ample amount in the early days when the HP-41 was designed. However, later we got banked modules allowing up to 4 banks to be kept into a single 4K page. With 16K it is possible to host a lot of functionality in a single addressable page and 64 functions may be somewhat limiting.

OS4 defines an extension to the existing FAT mechanism which greatly expands the number of functions in a 4K address page. You can have up to 4096 additional functions (which would be very hard to fit into 16K words).

These extra functions may reside in banked pages. They can be on redefined keyboard layouts (shell keyboards), they will display in program mode and they can be assigned to keys. If you use an extension to the `XEQ` and `ASN` keys, you can access them by name as any other function. In fact, they will in almost every way act as any ordinary FAT function.

3.5 Semi-merged operands

Functions in plug-in modules can display a prompt like any built in function, but the original mainframe does not allow such functions to be stored into a program.

OS4 provides a concept called semi-merged functions, which makes it possible for plug-in modules to have functions with prompting behavior that can be stored in programs. You enter such functions and fill in the postfix argument prompt the normal way. This work both outside as will as in program mode where the function will be properly recorded as a program step (two actually, see below). They will also display and execute correctly (with some caveats).

Built-in support for ordinary style postfix operands are provided. Full custom prompting behavior is also possible, but you will need to provide additional code on your own for such alternative behavior. Custom behavior needs to provide all expected behavior, such as recording, displaying and proper execution of the function. This is a lot of work, but possible. The ordinary postfix operand semi-merged functions are very easy to define.

In addition, secondary FAT functions can also have semi-merged behavior, including custom behavior. There are essentially no limitations on how you can combine these features.

3.6 Library routines

In addition to the above, OS4 provides many useful routines that makes the life easier for the MCODE developer.

3.7 Banking

The HP-41 banking mechanism is implemented by the memory systems and the CPU is completely unaware of this. This may seem a bit strange, but the Nut CPU actually executes unknown machine instructions as a no-operation and bus peripherals decipher the bus activity and act on the instruction.

The 1LG9 ROM memory chip provided by HP is capable of using two banks. It was used in the HP-41CX and by some plug-in modules (Advantage and IR printer). Such banks are presented at the same page address in the memory map, though of course only one bank at a time. The bank switch instructions are used to switch between the banks and careful layout of the software allows for switching banks in and out in a controlled way, allowing more memory to be used by the system. Most recently introduced memory systems allow for up to four banks, this includes the MLDL-2000, Clonix and HP-41CL.

The 1LG9 only act on bank switch instruction executed from *within* its own memory. The MLDL-2000 and HP-41CL mimics this behavior by pairing, so that page 8 and 9 are bank switched together, then pages following are paired in the same way. The original Clonix module on the other hand switches banks for all pages it serves and how that manifests itself depends on the size of the Clonix module and more specifically which pages it is configured to serve.

As a result of this, a banked module may or may not affect other modules, depending on which memory hardware and in part also how it is configured. While this may sound a bit scary, in normal situations this is not a problem as banked software is typically written so that secondary banks are only active in a temporary fashion and the bank is restored to the primary bank when control is given back to the operating system.

One potential problem is modules that introspect other banked modules. Such introspection is possible by using defined bank switch entry points that banked modules should have. If two such modules are loaded to the same bank switch pair, e.g page 8 and 9, it may not work as expected as when one module switch bank, the other module also switch bank.

This is also very much the case for OS4 which is bank switch and OS4 does introspect banked application modules to access secondary functions that may be in other banks. As page 4 is not bank switch together with application pages, it is kind of safe, but this of course depends on the memory system used.

Implementing very non-standard bank switch, e.g. leaving secondary banks active while no in control is quite fragile due to the different memory systems. While you *may* get such setup to work in a given setup, it may fail when a user loads your module image to another memory system or calculator configuration. Thus, it is probably safest to avoid such practices.

3.8 Catalogs

The original HP-41C mainframe provided three catalogs (1–3) to show user programs, functions in plug-in modules and built-in functions respectively. While you are in a catalog, you can stop, restart and then step it manually. When the catalog is stopped the HP-41 goes to light sleep and consumes less battery power. If you press an undefined key, like starting numeric entry, the catalog exits and the pressed key is obeyed.

The HP-41CX adds three additional catalogs and enhanced the previous catalogs. These new catalogs differ from the original catalogs and are more like separate programs. When the catalog is stopped the HP-41 is still running at full speed in a busy loop, consuming more power. If an undefined key is pressed, like numeric entry, the key is ignored and the catalog remains active.

The reason for this is that the original catalogs were carefully crafted to exist in the operating system in a very specific way. The mechanism used is very specific for this purpose and it was just easier to provide the new ones as add-ons with a somewhat inconsistent behavior. Part of the reason was also that two of the three new catalogs already existed as XROM functions in the Time and Extended Functions modules. Thus the catalog merely calls the already existing EMDIR and ALMCAT functions.

The shell mechanism provides a shell variant that is ideally suited for implementing new catalogs with similar properties as the original catalogs. This includes going to sleep consuming less power while waiting for a key press and ability to terminate the catalog and perform the action of an undefined (by the catalog) key press. New catalogs can even be implemented by different modules and accessed using the same catalog key.

3.9 Reserving identities

The original HP-41 never reserved buffer identities in a central place. This has resulted in that different modules may use the same buffer identity for different purposes, causing incompatibilities between such modules. A similar problem exists for XROM identity allocations, but this was unavoidable as only 31 such are available and over the years hundreds of modules have been made.

As OS4 lists identities for extension points and hosted buffers in a source repository on Github, there is a single central place where they are defined. If you want to reserve such identities, simply edit the OS4 header file and issue a pull request to reserve some identity, avoiding potential clashes.

CHAPTER 4

Addressing

Addressing is the way we refer to different memory locations. The HP-41 separates ROM from RAM by using entirely different access mechanisms. Normally this is typical for Harvard architectures though on the HP-41, RAM works basically the same as I/O peripherals. Addressing RAM is notoriously tricky on the HP-41, while ROM is somewhat easier.

4.1 ROM

ROM normally stores programs. With the `CXISA` instruction that was introduced with the HP-41 Nut CPU (compared to its predecessors), it is possible to read ROM locations for data purposes. This can be used to store strings, constants and tables. The address to read from is placed in `C[6:3]` (the address field of the CPU `C` register) and the fetched word is loaded to `C[2:0]` (the exponent field of the same `C` register). On MLDLs, there is often a `WROM` instruction which does the opposite, it writes the word in `C[2:0]]` to ROM at the address in `C[6:3]` (actually RAM pretending to be ROM to the HP-41).

It can be worth knowing that some instructions are not decoded by the CPU, but by peripherals listening on the bus. The `WROM` is an example of this. The Nut CPU sees it as a no-operation, but the MLDL hardware decodes it and acts on it. The bank selection instructions are similar in that they are handled by the ROM hardware and typically act only if the bank select instruction is fetched from the same memory chip. Peripheral device select, and its read and write instructions are decoded by peripherals in a similar way.

4.2 Stack

The four level internal stack also interacts with the `C[6:3]` field, making it possible to move between top of stack and `C[6:3]`. This field is aligned with the mantissa field, but it is shorter (four nibbles instead of ten), which makes it possible to use instructions acting on the mantissa field to manipulate an address without setting up any field pointers. This is handy as it allows for manipulation of the return address, making it possible to signal different outcomes of a subroutine call by returning to slightly different locations.

```
RTNP2:      c=stk
            c=c+1   m
            gotoc
```

This routine returns one address ahead of what it would normally do. Here we get the address from top of the stack into `C[6:3]`. Then it is incremented by one, but we operate on the mantissa field which is wider. For practical reasons[1] this gives identical result as if we had used the more narrow address field. Finally the `GOTOC` instruction jumps to the address in `C[6:3]` (the stack address plus one in this case).

This style is used in many MCODE programs to handle returns with different outcome, e.g. to signal a failure condition by returning to the normal return address and the success case returns one step ahead. This works as almost every instruction (including short branches) is a single word on the Nut CPU.

As the stack is only four levels deep we need to take care not nesting subroutines too deep. If you study existing code, e.g. mainframe code, you will find that the number of subroutine levels used is often documented. Should you nest too deep, you will end up jumping to address 0 which is not entirely harmful as that is where we end when starting execution, though the code will not work the way it was supposed.

4.3 RAM

RAM registers are as wide as the internal registers, 56 bits or 14 nibbles. This is quite generous, but addressing RAM memory is kind of painful. Typically we calculate and address in the `C[2:0]` field, also called `C.X` field. The good news is that this is one of the more powerful fields of the CPU, as we can load any 10-bit constant using the `LDI` instruction and perform arithmetics there.

To address a RAM address, we use the `DADD=C` instruction which takes the address from `C.X` and selects that data location. After that we probably want to either read or write, so the address in `C.X` is in the way. To make it worse, if we want to perform any kind of address arithmetics, it is natural to have the address in `A.X`. Unfortunately, there is no instruction to copy the contents of `A.X` to `C.X`, instead we need to exchange them, select the address location, then exchange (or copy, as we can copy in the opposite direction) the address back the `A.X` to avoid losing it.

While the RAM chip allows for direct addressing inside the current 16-register window, we can almost never use it as the memory system is quite flexible in how it partitions things, which makes it impossible to know how an arbitrary address translates to a fixed offset inside the current 16-register RAM chip.

The mainframe makes good use of the lower registers (address 0–15), internally called chip 0. By selecting any low address (0–15), we can access any of these registers by its number. For other RAM access, we are essentially bound to addressing a single register at a time, and repeat the selection procedure whenever we access any nearby register.

Note: It can be worth knowing that when the HP-15C code was ported over to the Advantage ROM, it relied a lot on fixed addressing. To avoid having to rewrite it to use dynamically calculated addresses, the HP engineers (ex-HP actually it seems) set up a fixed address area using a buffer so that fixed addressing could be used. This was done in a tricky way by storing a buffer below the key assignment registers (which is the only fixed location outside chip 0). However, it cannot normally be there, so a lot of measures were taken to not leave it there under normal operation.

---

[1] The mantissa field increment may affect all 10 nibbles, not just the four in the address field. This happens when the value in the address field is `0xffff`, which is unlikely in this case as it would mean the return address would be to the last address of the memory space, where there normally is a module checksum. Even if we should affect all nibbles in the mantissa field, it is rarely a problem anyway as there seldom is anything kept there of value in cases where we work on the address part of the field.

CHAPTER 5

Basics

In this chapter we go through some concepts that are used internally in OS4. This is not a primer on MCODE programming or the Nut CPU, you are expected to be familiar with MCODE programming.

5.1 Addressing

ROM addressing is done using 16-bit addresses. This gives a total of 64K of addressable memory space. The HP-41 mainframe divides this up in 4K blocks and treats each such block as a page. There is really nothing magic about having such 4K blocks in the Nut architecture, it is just a way of dividing up the memory to allow for modular extensibility.

Each memory location holds a 10-bit word and most instructions are single word. The only exceptions are the absolute jump and go subroutine instructions which takes two words. However, the good news is that they are able to reach any fixed location in the 16-bit address space. (The `POWOFF` instruction needs to be followed by a `NOP`, so it can be seen as taking two words as well).

As the Nut CPU has notoriously bad addressing capabilities, the only way to read data from the ROM space is using the `CXISA` instruction which expects an address in the `[6:3]` field of the C register.

5.1.1 Packed pointer

As we often work inside a single (often relocatable) 4K page it is convenient to have a compact notation to describe a location inside such page. We will often know which page we are in, as we may be called from it or having some pointer to some ROM structure at hand. Thus, the 4-bit page address is typically known through some context we already have.

Here we have a slight problem as we need 12 bits for a page address and a ROM word is only 10 bits wide. To make it possible to describe a location inside the entire page, OS4 uses a concept called a packed pointer which is a 12-bit page pointer shifted right two bit positions, resulting in a single 10-bit value.

To unpack a packed pointer, the 10 bit value is left shifted two times and combined with the page address. This means that the actual address needs to be aligned at an even of four address, i.e. the address must end with `0`, `4`, `8` or `C` (hex).

In other words, we can describe an address within my own module using a single 10-bit word and it can be placed anywhere in a page relocatable module, provided we follow the alignment constraint.

If you are using NutStudio tools the `.align` directive makes it easy to specify an alignment of four and the `.low12` relocation operator makes it possible to obtain a packed pointer:

```
            .align  4                   ; align table
table:      .con    .low12 FAT1Start    ; single word pointer
```

5.2 Return status

A routine may need to deal with possible error conditions. For flexibility it may be better to return with some error condition rather than displaying an error message. The caller may have another way of dealing with a failure than showing an error message.

Due to the nature of the Nut CPU it turns out that it is often easy to do this by returning to different locations rather than returning some kind of error code. At the call site it looks like this:

```
            gosub   findBuffer
            goto    noBuffer        ; (P+1) not found
foundBuffer: ...
```

We call this `(P+n)` and this works well thanks that almost all instructions are single word (including short jumps) and we most often want to branch to some alternative location to handle the case.

The caller will just `RTN` to get to `(P+1)`, and the success case `(P+2)` means it needs to return to the following location, which can be done as follows:

```
RTNP2:      c=stk
            c=c+1   m
            gotoc
```

The disadvantage here is that we clobber the address field `[6:3]` of the C register, which means that we cannot pass any return value there, as we often use the incremented return when successful.

5.3 Buffer advice

I/O buffers, or just buffers for short, were defined from the beginning in the HP-41 mainframe. However, they were first used by the Time module, about two years after the introduction of the HP-41.

A buffer can have any size from a single register up to 255 registers. The first word is called the buffer header and the leftmost four nibbles of this register have well defined meanings.

The first two are defined to be the buffer number 1-14 duplicated in both nibbles. The Time module which used 10 will therefore put 10 in both nibbles, or `AA` in hex.

The following two nibbles, a byte (eight bits) is the size of the buffer. The buffer header is included in this count. Eight bits limits the size to 255 (as the size 0 has no useful meaning).

Even though the buffer number was defined to be a double word like `AA`, the information carried in the first nibble is only zero or non-zero. Zero means that the buffer is marked for removal and any other value means that it is active.

The last register used in buffer must be non-empty as the operating system will scan from the other direction to find free registers and the first non-zero register found is considered occupied.

5.3.1 Non-null registers

The Time module buffer code take precautions to never store a zero value inside a buffer too. This is due to a 67/97 card reader bug which I have not been able to find out what it means. I suspect that the card reader (at least early versions) may scan for free registers looking at individual registers also inside buffers.

As a result, you should probably avoid storing empty registers inside the buffer to avoid potential memory corruption.

5.4 System buffer

The OS4 module needs to store its own state somewhere. The mainframe code typically uses the 0–15 RAM address status area for such purposes, so that space already occupied. The safest way to find some free memory is to use a buffer and the OS4 module allocates a system buffer with number 15.

The advantages of using a buffer are that it is a safe area and it can grow (and shrink) dynamically as needed, rather than being fixed.

The disadvantages of using a buffer are that it takes a little bit time to locate it and we may run out of space if there are no free registers that can be occupied when the buffer needs to grow.

5.5 Keyboard

HP calculators before the arrival of the HP-41 used fixed keyboard layouts and an increasing number of shift keys culminating with the HP-67 that carried no less than three different shift keys. The HP-41 made away with this and went back to a single shift key and the reassignable keyboard in user mode.

As you are probably familiar with the HP-41, you know about its ability to reassign keys, keys that talk and can be NULLed (to inspect the current behavior). There are actually a lot of different aspects on how the keyboard can be reassigned and different classes of functions that may not be obvious until you look closer at it.

5.5.1 Reassigned keys

Keys can be reassigned and change behavior in user mode. If in doubt, you can press and hold the key to see its current behavior. In addition to explicit assignments, local alpha labels in the current RPN program is scanned when one of the two top row keys is pressed.

5.6 Key codes

There are several ways key codes are represented in the HP-41. The key codes returned from the keyboard as read by a machine instruction does not match the ways we want to present them to the user. The key codes presented to the user are for key assignments and match a logical layout related to rows and columns on the keyboard. Internally though, the 0–79 and 1–80 forms are used. These forms are easily converted between by increment (or decrement) the key code by one. The reason for the two forms is that the internal key tables use an index starting at 0 (0–79 form), but 0 is reserved for an empty assignment slot in the key assignment registers. The solution is to increment the key code by one, giving the 1–80 form, which makes it possible to tell an inactive assignment apart from an active assignment.

Internal key tables are just an array of function codes where we take advantage of the extra two bits in a ROM word to decode a special meanings, like a data entry key or a function that ends digit entry or not. As we

want to allow storing also XROM functions on keyboard, the actual encoding used by OS4 differs somewhat from the ones used in the operating system.

If most of the keys are given a meaning it makes sense to define a keyboard like an array indexed in 0–79 form, just like the built in keyboards. As an alternative, OS4 provides a way of defining a sparse keyboard where a 0–79 key code is stored paired with its function. In this case a linear scan is used, which saves space if few keys are defined, while still being reasonable fast.

As also secondary functions can be bound to keyboard definitions, there are some further schemes and details on how more advanced keyboards are defined. This is further described in *Defining keyboards*.

5.7 Return to mainframe

The normal behavior for an MCODE function is to exit using a `RTN` instruction. As the invocation mechanism push the address of `NFRPU` on the stack before giving control, this is where we will normally return. This exit point enables stack lift (sets the internal push flag, which is CPU flag 11) and falls into `NFRC`.

If you used up all four levels of CPU stack, you must exit back using a `GOLONG` instruction instead. By design, XKD functions (seldom used functions that execute immediately on key down) does not have `NFRPU` pushed on the stack, so they also must always `GOLONG` back. Such functions may want to return back to `NFRKB` instead as it waits for key release and resets the keyboard (useful as it acted immediately on key down).

CHAPTER 6

Functions

The different classes and execution of functions is a rather vast concept on the HP-41, enough to warrant a chapter of its own.

6.1 Execution tokens

Functions are internally represented by an execution token which is a sequence of bytes. This is also how they are stored in programs. There is a very convenient quick reference card that documents them. You can find a copy of that in A programmers handbook on page 37–38.

The execution token of the current function being keyed in is kept in register 10 (chip 0) field `[4:1]`.

6.2 Classes of functions

There are three main classes of functions, ordinary functions, prompting functions and execute direct functions.

In addition there is data entry keys (including backspace), which we will not cover here.

6.2.1 Ordinary functions

An ordinary function is one that goes through the NULL check and does not put up a prompt, e.g. `SIN`. They can either be programmable or non-programmable. This is controlled by whether the first instruction address (internally called XADR) is a `NOP` instruction (opcode `0x000`) or not.

6.2.2 Execute direct functions

Execute direct functions (XKD) are use for functions that act immediately on key down, e.g. `SHIFT`, `SST` and `R/S`. A function is marked as being XKD by starting with two `NOP` instructions (opcode `0x000`).

Normally the default address of return to mainframe (NFRPU) is pushed on the stack before execution, but this is not done for execute direct functions, which must always jump back to some NFR routine when the function is done.

6.2.3 Prompting functions

A prompting function has some upper bits set in its name header. This specifies what class of prompting function it belongs to. A prompting function does not go through any NULL check, instead its name is displayed together with one or more prompt underscores.

The prompt is handled by the partial key sequence parser which goes to light sleep while waiting for key presses. When the prompt is completed, it is either combined with the execution token (in case it is an internal function), or kept separately in a register (in case it is an XROM function). This is because there is no room for the operand to be combined together with the execution token of an XROM. The internal state only have room for two bytes and the execution token of an XROM uses those two bytes without any operand attached to it.

6.3 Finding a function

There are two ways a function can be found, by its name and by its execution token.

The most common way to execute a function by name is to use the XEQ function and then enter the name (separated by pressing the Alpha key).

Execution by token is done from inside a program or when bound to a key on the keyboard.

It can be worth knowing that execution by name actually is a two-step process. The first step locates the function by its name by scanning the catalogs (more on this later) and the second step executes the function by its execution token.

If the function found by name is an MCODE function is, it is first to checked if it is execute direct (XKD). If it is, the function is immediately invoked. The usual return address to mainframe (NFRPU at address 0x00c0) is not pushed onto the stack in this case. If it is not XKD, the function name is displayed, which serves two purposes. If it is a prompting function we want it together with the prompt. If not, we want the name in the display becomes visible if the user keeps the key pressed, which is the talking key feature. If the key is kept down for 0.5 seconds, the function invocation is aborted and NULL is displayed.

If the function passes the NULL check it is executed by its execution token, not by its name. That might sound a bit confusing, but what actually happens is that even though we know the execution address (XADR in mainframe terminology), we only use that for showing the name of the function and deciding what class it belongs to. The actual XADR is only used for execution if the function is XKD. All other functions are executed by its execution token, even when you type it by name.

So what does it mean? We look up the function twice and in two different ways, first by name and then by its execution token. As with every situation of doing seemingly the same thing but in different ways, we risk coming to different results and this is no exception to that rule. What can happen is that you type a function by name, keep the final Alpha key press down, the function name is properly shown in the display (it comes from the XADR) which confirms we found it. When the key is released the function is sent for execution by its execution token, which results in another scan for the function based on its token (not its name). This may find a completely different function that happens to match the given token! Of course, this only happens if you plug in two modules using the same XROM code, something that is probably best avoided.

6.3.1 Search order

Searching a function by name is done in catalog order. User programs in catalog are searched first, followed by plug-in modules (XROMs) in address order[1] and finally the built-in functions in catalog 3.

OS4 extends the search by also searching for secondary functions. This search is done for each XROM page after searching the ordinary FAT in that page.

6.3.2 Lookup

Looking up an XROM by its execution token is done by scanning the plug-in modules in the same page order as is done when searching by name.

6.4 Semi-merged functions

Many operations in the HP-41 consist of a function and a postfix argument, e.g. `FIX 4` or `RCL IND Z`. When pressed, the operand will output one or more underscores to be filled in with the argument. The base operating system allows XROM instructions to be defined as prompting too, but it cannot represent them in program memory. It is mainly a side-effect of the flexibility of how the base operating system was written and the main use of it was to provide an easy way of doing alpha input to functions such as `PRP` in the printer ROM. Later the Extended Functions module provided means of reading such arguments from the alpha register.

The OS4 module provides a way for XROM to prompt for arguments and represent them as program steps. It is not possible to fully merge such program steps, but OS4 allows you to get partly there. In program memory the XROM is followed by an alpha literal that wraps the postfix operand. When shown in program memory, the postfix operand is automatically taken from the alpha literal, allowing you to see the instruction in its full glory. However, if you step ahead you will see the alpha literal as a separate step.

6.5 Secondary functions

The function address table, or FAT for short is the inventory of functions that a plug-in module provides. It is located first in the module. This table provides up to 64 functions, which may have seemed a lot from the beginning, but with the arrival of banked modules you may find that you wish you had more entries.

The OS4 module provides a mechanism for providing secondary functions. Up to 4096 such functions are possible. These extra XROM functions are called XXROM.

You can see such XXROM functions as having a numeric identity in the same way as an XROM, though the function number has a wider range 0–4095, e.g. `XXROM 7,689` would be the 690th secondary function in a module with XROM identifier 7 (the numbers start from 0).

Note: The numeric series are separate, so you have up to 64 primary and 4096 secondary functions in a module.

With the Boost module, you can key the name of the secondary function from its `XEQ'` instruction which is available as a replacement for the ordinary `XEQ` function. This means you can access a secondary function in the same way as any other named function. The normal search order rules are used, following the catalog order. Primary XROM functions are searched before looking at secondary XXROM functions in the same page.

[1] The HP-41CX extended the plug-in module range by adding things in page 3. The search is from page 5 to 15, but on an HP-41CX page 3 is additionally searched after page 15.

A secondary function can also be assigned to a key. If you press such key in user mode it will go through the normal behavior showing its name and NULL if you keep the key pressed. If it is a prompting function it will put up its prompt, just like any primary XROM or built-in function would do.

The actual assignment information is kept inside the system buffer. If you assign a secondary function to a key and remove the module, the key will display as an XXROM, e.g. `XXROM 7,45` indicating which function it is and that it is not present.

Secondary functions can also be entered in programs and they will be correctly displayed in program memory. However, in order to represent them in program memory they are actually stored as an XROM (acting as a prefix) followed by a semi-merged alpha literal.

In summary, secondary functions provide a way of having essentially as many functions as you can fit into the memory constraints rather than being limited by as fixed maximum of 64. Using the Boost companion module, you can access them the same way as ordinary functions and they can also be assigned to keys and stored into program memory. In addition, they are just as powerful when it comes to prompting as any normal function.

Note: You need the Boost module to obtain the `XEQ'` and `ASN'` replacement functions to access secondary functions. The routines that look up secondary functions by name or its numeric identity are in OS4.

6.6 What is up

In the book HP-41 MCODE Programming for Beginners appendix B (page 132) lists what is up on function entry. Secondary functions diverge a bit from what is listed there and the list is probably a bit too detailed anyway. The following is what you can rely on:

1. CPU is set to hex mode.
2. Flags 48 to 55 of the user flag register is in ST.
3. RAM chip 0 is selected.

6.7 Internal representation

Finally we will take a look at the ways that are used to represent the function internally. This is probably not anything you normally need to bother so much about, but in some situations it can be good to know. It also gives an improved understanding for how it works.

6.7.1 By address

We sometimes represent a function by its execution address (XADR), which is the first execution address of a function. Here is an example of how a function starts:

```
            .name   "RAMED"
RAMED:      nop                 ; non-programmable
            ?s3=1               ; program mode?
```

In this case the execution address (XADR) corresponds to the `RAMED` label. From this we can look at previous locations to get the name of the function and we can use the XADR to execute the function. We may also inspect the first locations at the XADR to see whether this function is non-programmable or XKD. The example above is an ordinary non-programmable function.

For secondary functions the address of the XADR is not enough. We also need to keep the bank it is located in, so the XADR for a secondary function is actually two addresses. The bank is represented by a pointer to the bank switcher routine associated with the secondary FAT header which points the secondary FAT the function belongs to. This allows for switching to the correct bank before accessing the first locations.

6.7.2 By execution token

Ordinary XROM functions are represented by a two bytes execution token. Secondary functions have a couple of different representations:

1. The XROM (1–31) identity and the secondary function number (0–4095). This is how key assignments are represented.
2. By the XROM prefix function associated with the secondary FAT header and an adjusted secondary function number (0–255). This is also how they are stored in program memory.

During keyboard execution the needed information, such as XADR, bank switcher and secondary function number are stored in the M register or other temporary places.

CHAPTER 7

Idioms and ideas

Programming idioms are ideas and snippets of code on how to solve typical problems. Knowing and following idioms will help you in writing good code by levering existing experience. Idioms will help you understand how to approach typical problems, and by using them it makes your code easier to read by other people. However, you should not take idioms presented here as the whole truth. Feel free to invent your own and explore MCODE with an open mind. They are just some good ideas on which you can base your own code on. If you feel that you want to do it in different ways, by all means do so.

7.1 Error returns

Often we want to tell whether a routine succeeded or if there was an error. When we want to report an error we can use one of the error exit routines provided by mainframe. Such exits are `ERRDE` for showing `DATA ERROR`, and some other routines for other standard error messages. OS4 also provides some additional error handler routines with other messages.

A routine typically becomes more flexible if it does not exit with an error message, but rather returns in a way to signal an error, delegating the error handling to the caller. This makes the routine more flexible as a caller can take other actions on the error condition than just exiting with an error message.

Normally we would return an error value, like zero which is easy to test for by the caller. On the Nut CPU there is an alternative idiom by returning to different locations depending on whether there was an error or success (or even different classes of results). This is rather unusual, but works exceptionally well on the Nut instruction set for two reasons. First, almost all instructions are of the same size (one word). Second, it is is very easy to handle returns to different locations. A simple return is just the `RTN` instruction, or one of the conditional `RTNC` or `RTNNC` instructions. Returning to an incremented location is also very simple as shown below.

If we want to return back that we failed, we return to something called `(P+1)`. This is fancy for doing a normal return. If we succeed, we return to `(P+2)`, which means that we skip past the normal return address by one.

What it means is that a caller may look as follows

```
locateBuffer: ldi   2         ; buffer number
              gosub findBuffer
```

(continues on next page)

(continued from previous page)

```
            goto  noBuf     ; (P+1) deal with that no such buffer exists
            ...             ; (P+2) success, we found buffer 2 (we probably also
                            ;       got some pointer back and the header selected)
```

In the actual `findBuffer` routine, we test for various error conditions, such as if running out of the buffer area or finding the permanent `.END.`. In the code we can test various conditions, like if we stepped to the register with `.END.` or found an unused register. In such cases we can do a test and conditionally return, which will take us to `(P+1)` of the caller:

```
findBuffer:   ...
              ?a<c    x             ; have we reached chain head (.END.)?
              rtnnc                 ; yes, not found
              ...
              ?c#0                  ; empty register?
              rtnnc                 ; yes, not found
```

When we find what we are looking for, we want to return to `(P+2)`, which is done by incrementing the return address. This is done in the following way:

```
RTNP2:        stk=c                 ; C[6:3]= return address
              c=c+1   m             ; advance to (P+2)
              gotoc                 ; return
```

The `GOTOC` instruction is very useful here as it simply loads the program counter with `C[6:3]`, meaning we take the address field of C and jump to it, which is exactly what we want to do here.

As can be seen, a single conditional `RTN` instruction can handle errors, and a short three word sequence will allow us to return to `(P+2)`. On the caller side, a single word branch (`GOTO`) will take us somewhere to deal with the error condition. There are no error value or flags passed back, nor any tests needed at the call site. It gets very small and simple.

It is not known who came up with this trick first, but it is used extensively in the Time module. You will find code in the mainframe that does not employ this technique, but rather returning 0 or setting a flag (as in the HP-IL module).

The main advantage of the return to different locations idiom is that it saves an instruction on each call site.

A disadvantage is that you need to act on the condition immediately. Another problem is that you cannot use the `C[6:3]` field to carry any return value.

7.2 Call backs

A similar technique can be used for implementing call backs. We can make a call and keep a call back pointer at `(P+1)`. In fact, we can easily have multiple call backs by just adding pointers.

```
            gosub routine
            goto  handler1 ; (P+1) first call back
            goto  handler2 ; (P+2) second call back
            ...
            xxx             ; (P+N) normal return


            ...

handler1:   do-stuff ...
            rtn
```

The called routine can then pop the return address and keep it handy in for example M.

```
routine:    c=stk
            m=c         ; M[6:3]= points to callBack1
```

Later we can call a routine using:

```
            ...
            gosub  callBack2


callBack2:  c=m
            c=c+1  m
            gotoc
```

Here we make use of having the base call back pointer in M[6:3]. Note that it trashes part of the C register. However, making any page relative call (3-word) also destroys most of C register.

The final return from routine is made by jumping back to its callBackN routine.

7.3 Code pointers

As a ROM word is only 10 bits long. We are lacking a few bits to make a full 16-bit code pointer. Instead of using two words, we can get away with only using 10 bits by observing two things.

First, the code that are providing the code pointer is in a page relocatable module. Normally, we do not know which page we will be executing from. This can be found at run-time using the PCTOC in the operating system. In practice, it is often easier to leave it to the called routine to figure it out (as it has the return address to it on the stack).

Second, with the page sort of taken care of, we have 12 bits to represent using 10 bits. We can do this by aligning the code so that the address we want to pass on is aligned to an even 4-word address.

To summarize, we can represent a 4K page local pointer using 10 bits (a single ROM word) and have it page relocatable. The only thing the caller need to do is to ensure it is alignment on an even 4-word address which is easy to do with an assembler directive. For a caller it would look as follows:

```
            ldi  .low12 label
            gosub routine

            ...
            .section code
            .align   4
label:
```

On the receiver side we need to construct the full address from the 10 bit data in C[2:0]. We can get the page from the return address and we need to scale C[2:0] as follows:

```
unpack:     c=c+c   x
            c=c+c   x       ; C.X * 4
            c=stk           ; C[6]= page
            stk=c
            csr     m
            csr     m
            csr     m       ; C[3:0]= full address
            rcr     -3      ; C[6:3]= full address
```

The only minor issue is how to combine the page with the lower 12 bits. This depends a little bit on how we are given the lower part and what we want to do with the result. The above code can serve as an example, but there may be other ways depending on the circumstances.

7.4 Call backs with code pointers

The call backs presented above used ordinary GOTO instructions in a similar fashion as what was done with error returns. As the GOTO instruction only reaches 63–64 words distance, having a couple of routines may result in that some GOTO may be out of range.

There is nothing that says that the (P+N) words need to be GOTO instructions. We could actually use 10-bit code pointers instead and have reachability anywhere in the 4k page (given that we align our call backs).

Our invocation of routine would then become:

```
              gosub routine
              .con  .low12 handler1 ; (P+1) first call back
              .con  .low12 handler2 ; (P+2) second call back
              ...
              xxx                   ; (P+N) normal return

              ...

handler1:     do-stuff ...
              rtn
```

Our routine for a start look as before, as we still want to keep track of the (P+1) pointer, it is just what is stored at those addresses that changed, not the (P+1) itself.

```
routine:      c=stk             ; get (P+1)
              m=c               ; M[6:3]= pointer to (P+1)

              ...
              gosub callBackAdr2
```

What is different is the actual call back helper, here it is named differently to distinguish it from the previous, as we may want to have both variant around.

```
callBackAdr2: c=m
              c=c+1  m
              cxisa
              c=c+c  x
              c=c+c  x
              csr    m
              csr    m
              csr    m
              rcr    -3
              gotoc
```

7.4.1 Optional call backs

We may want to have optional code pointers. That is, the caller may not need to provide a call back at all. This can be done in a couple of ways. We can either read the word and test it for 0. Such value is easy to test for and cannot be legal anyway as it would take us to the first address of the page where there is data (XROM identity and FAT):

```
callBackAdr2: c=m
              c=c+1  m
              cxisa
              ?c#0   x    ; does it exist?
              rtnnc       ; no
              ...         ; yes
```

The alternative would be to store a real pointer that points to a RTN instruction. We can then omit the 2 words to test above, but on the other hand we would need to provide a RTN instruction that is aligned, so it would perhaps not save so much. In this case it is a matter of taste, and having 0 as empty value is easier for the user and is perhaps somewhat more natural.

7.4.2 Combined call backs

While the call back routine is not large, it is not trivial either. We want to avoid code duplication so it may be a good idea to arrange these routines together so that they can share code:

```
callBackAdr1: c=m
              goto   callBackAdr0
callBackAdr2: c=m
              goto   callBackAdr1
callBackAdr3: c=m
              goto   callBackAdr2
callBackAdr4: c=m
              c=c+1  m
callBackAdr2: c=c+1  m
callBackAdr1: c=c+1  m
callBackAdr0: cxisa
              ?c#0   x
              rtnnc
              c=c+c  x
              c=c+c  x
              csr    m
              csr    m
              csr    m
              rcr    -3
              gotoc
```

As can be seen, the cost for an additional (P+N) routine is three words. One word to add one more to the address and two words to create the (P+N-1) entry point.

CHAPTER 8

System buffer

A system buffer is used to keep track of the OS4 state. As all other buffers it resides in the memory area between key assignments and user programs, the so called buffer area.

The buffer number used is 15, which is a buffer number that is probably not used by any other module as of this writing. 15 is normally used for key assignments, so you may wonder how this can work. A buffer as defined by HP is suggested to have the first two nibbles (4-bit values) both set to the buffer number. This allows for 14 buffers, numbered 1–14. The time module uses buffer 10, so it sets the first two nibbles to AA (A hex is 10 decimal) for its buffer. However, FF would not work as it could be mistaken for a key assignment register. The answer is that OS4 stored 1F there instead. The first nibble can actually be any non-zero value except 15 and work properly. Taking advantage of this allows for having buffer 0 and 15, for a total of 16 buffers. Buffer 0 is already used by the Ladybug module which from version 1A takes advantage of the OS4 module, which leaves 15 for the OS4 system buffer.

8.1 Buffer layout

The buffer consists of a header register followed by several areas that appear in a defined order. The size of each area is kept in the buffer header, which means that to get to a certain area, sizes of areas before need to be summed and added to the buffer header address. There are routines in OS4 to help with this, but normally you will use more high level routines that deal with more complete actions on the buffer, finding the area inside the buffer is just a small detail.

The sizes are in the buffer header. Some are two nibbles (a byte) and two are single nibble sizes, as there is a little shortage of room in the buffer header. These single nibble ones have been chosen to be those that are less likely to need a lot of registers.

The header register is defined as follows:

```
ID SZ BF SC:KA SH DF ST
```

The ID is the identifier which normally set to 1F. The SZ field is the size of the buffer. These two fields are defined by the HP-41 operating system and all buffers are like this.

The ST field is a set of flags that keeps track of the internal state. You normally do not need to take these in account as they are handled internally by OS4.

The DF field is used by the semi-merged mechanism to store a postfix arguments during function entry. When entering a single postfix argument, the default argument is stored here. For dual postfix functions, this field is used to hold the first entered argument.

The remaining for fields BF, SC, KA and SH are the sizes of the different areas that follows the header. From the sizes we know the size of each area and by knowing the order we can also calculate the start address of each area by knowing the buffer header address.

Note: You do not normally need to worry about the fields in the buffer header as they are internal to OS4. It should be sufficient to use the provided routines in the API. It is important to use the provided routines when making changes to area sizes, as there are certain invariants that need to be maintained to keep the state consistent.

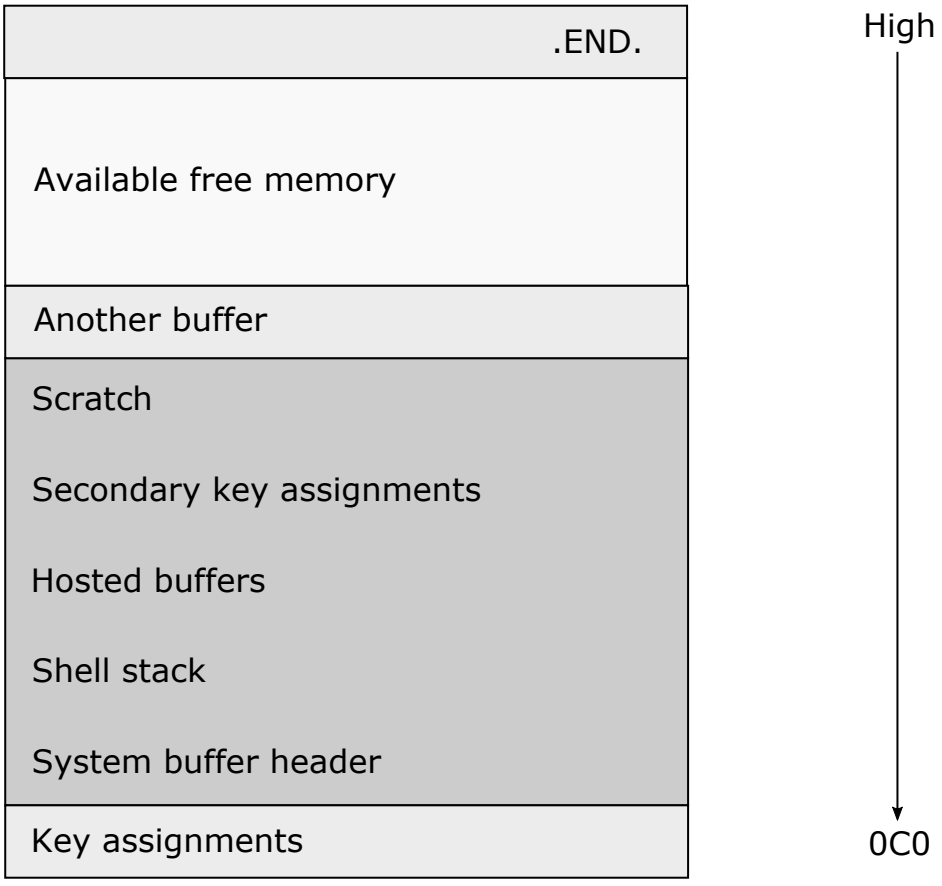


Fig. 1: Key assignments and buffer area.

8.2 Area sizes

Most of the areas are defined as being simply the size, where 0 means that the area is empty. For secondary key assignments we add two additional registers for assignment bitmaps when it is non-zero, meaning it will occupy 0, 3, 4 and up to 17 registers. The temporary scratch area could in theory avoid having a nibble for its size, but making it explicit simplifies the code. (It could in theory be calculated from the rest of the sizes).

8.3 Shell registers

Shells will be described in more detail later. For now it is enough to know that a shell occupies 7 nibbles and two are stored into each buffer shell register. The shell area is a stack, so the order of which they appear describes how they affect the current behavior. The top of the stack is in first register (the one with lowest address) and starts in the lower half. There is no need to have any special marker (`F0`) as with normal key assignments, as the shell registers are kept inside the buffer structure.

8.4 Hosted buffer area

This is an area that keeps track of application buffers. These have much of the same properties as ordinary buffers, but are actually stored inside the system buffer. They are somewhat easier to use than making your own buffer code and are suitable for smaller buffers as the overall buffer size (of the system buffer) is limited to 255 registers. They also have the advantage of having buffer identities that are unrelated to normal buffers, making clashes far less likely.

8.5 Secondary assignments

Functions defined in secondary function address tables can be bound to keys and are stored in the secondary assignment area. Two assignments can be stored in one register. This gives seven nibbles to describe one assignment. The function code has a two nibbles XROM number and a three nibbles secondary function number. The two remaining nibbles are the key code.

Due to using a single nibble being used for the size of this area, there is a limit of 30 secondary functions being assigned to keys.

The secondary assignments have bitmap registers for fast lookup, much like what is used by the ordinary assignments. The ordinary bitmaps shadow the secondaries in case they both say they are assigned. This should normally not happen, unless you use existing functions to load key assignments from secondary storage, like magnetic card. In this case there are typically two variants, one to replace all existing assignments and one to merge key assignments. With respect to secondary assignments, they both behave as merging as they are put on top of existing secondary assignments. This is because they are unaware of the concept of secondary assignments. In this case the user may need to run a routine to clear secondary assignment separately.

This also means that if you load key assignments over a secondary assignment, the primary (ordinary) assignment takes precedence, but if you clear the assignment through means outside OS4, the shadowed secondary assignment may reappear.

CHAPTER 9

Shells

A new concept provided is a shell which provides a way of installing new keyboard and display behaviors.

The shell keyboard replaces the standard keyboard layout and allows for complete redefinition of keys. You can use standard functions, XROMs and even XXROMs. The display routine can be used to change the default view of the X register to something that suits the current shell better.

Both the keyboard replacement and display routines are optional and allows for overriding one of them, while using the default behavior of the other.

By providing several MCODE functions, bind them on a custom keyboard and perhaps a new default display, you have an application. This can be a new mode to make the HP-41 behave like an HP-16C in integer mode, the HP-12C by providing suitable financial functionality, or even a mode for working with complex numbers.

It is worth pointing out that the new keyboard definition is done in a similar way as the built-in keyboard, but with additional capabilities to accept both XROM and XXROM functions on it. No user key assignments are used. In fact, the user can redefine keys on top of it in the usual way, to customize the calculator further.

Note: Even if shells allow for making radical different behaviors, it is often a good idea to adhere to the basic behaviors, allowing the calculator to feel like it normally does. Thus, many keys should probably behave more or less like they usually do. The top mode changing keys should allow for entering program mode or work with the alpha register. Keys to control program execution should probably still do that.

Note: If desired, you can experiment with making the calculator completely customized for a particular purpose, like a military combat computer. In such cases you may want to disable certain things that may distract the operator. Back in the days, OS4 would probably have been a great asset for such purposes. However, today we are probably more inclined to broaden rather than restrict the capabilities of the HP-41.

9.1 Shell stack

The shells are stored in a shell stack. If you activate an application shell it pushed on top of the stack, shadowing all applications below it. When you exit an application, the one immediately below becomes

active again, all the way down to the standard behavior.

There are four kind of entities stored in the shell stack. Even though they are pushed on the stack, they are kept together based on its kind. Pushing a shell means it is stored on top of other shells of the same kind.

You can think of a shell as defining a keyboard layout. Such keyboard layout does not need to have a meaning for every key. If a key that is not handled is pressed, the next shell on the stack is inspected. However, among the applications, only the topmost one is given the chance to handle a key press, shadowed applications are ignored.

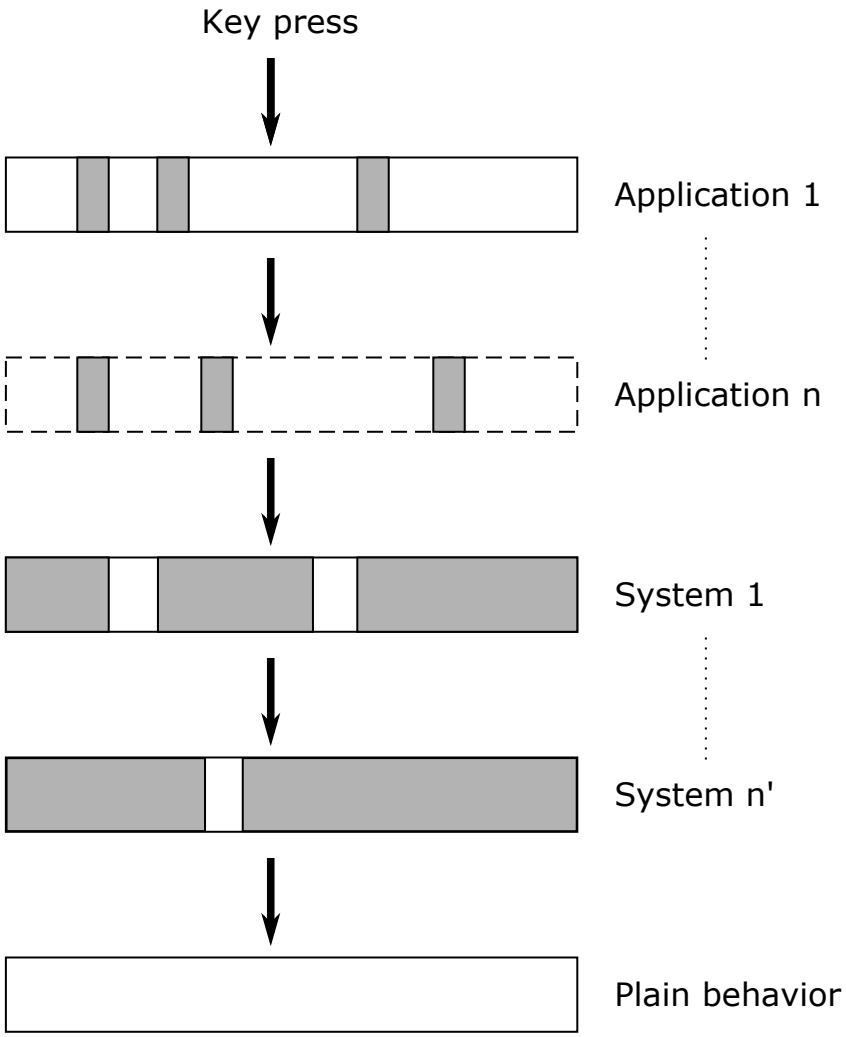


Fig. 1: The shell stack

9.2 Shell kinds

There are four kinds of shells and they have different purposes and follow somewhat different rules.

9.2.1 Application shells

The most fundamental shell type is the *application shell*. They live near the top of the shell stack. Only the topmost application shell on the stack is consulted for interpreting the keyboard and the display behavior. Any application shell below the top one are ignored, but are kept to preserve the ordering to allow the user go back to a previous application when leaving the currently active one.

9.2.2 Transient applications shells

A variant of an application is a *transient application shell*. This is a specialized version of an application that always is at the top of the stack and there can be only one transient application shell in the stack. A transient application is meant to be used for some temporary mode that is normally exited when a key not handled by it is pressed. This is similar to how the original catalog 1–3 works. If you press a key that does not have a defined by the catalog when it is stopped, the catalog exits and the key performs its underlying behavior.

This means that if you press a key that is not handled by the transient application, it is removed from the shell stack and the topmost ordinary application will get a chance to handle the key.

9.2.3 System shells

The third shell variant is a *system shell*. System shells are located below all application shells in the shell stack. All system shells are active and each one is always consulted in the stacking order until a handler is found. They are typically used for replacing single (or a few) keys, providing alternative or additional functionality. One example is a replacement for the assign (`ASN`) function that could be implemented using a system shell.

9.2.4 Extension handlers

The final entity that lives in the shell stack is *extension handlers*. They are very different from the shells as they implement a generic message system. There are no keyboard or display behavior associated with them. Events are routed to message handlers which act on a given message.

9.3 Shell structure

A shell is defined by a structure that consists of several elements. It is defined as follows:

```
              .align 4
myShell:      .con    kind
              .con    .low12 displayHandler
              .con    .low12 standardKeys
              .con    .low12 userKeys
              .con    .low12 alphaKeys
              .con    .low12 appendName
              .con    .low12 timeoutHandler
```

The structure must start on an address aligned by 4. It contains several pointers that also must be aligned by 4 (which can be seen by the use of the `.low12` relocation operator in the example above).

9.3.1 Kind field

The kind field tells what kind of shell this entry represents. The values are defined in `OS4.h` and are either `SysShell`, `AppShell` and `TransAppShell`. The `GenericExtension` also exists, but the structure following it is

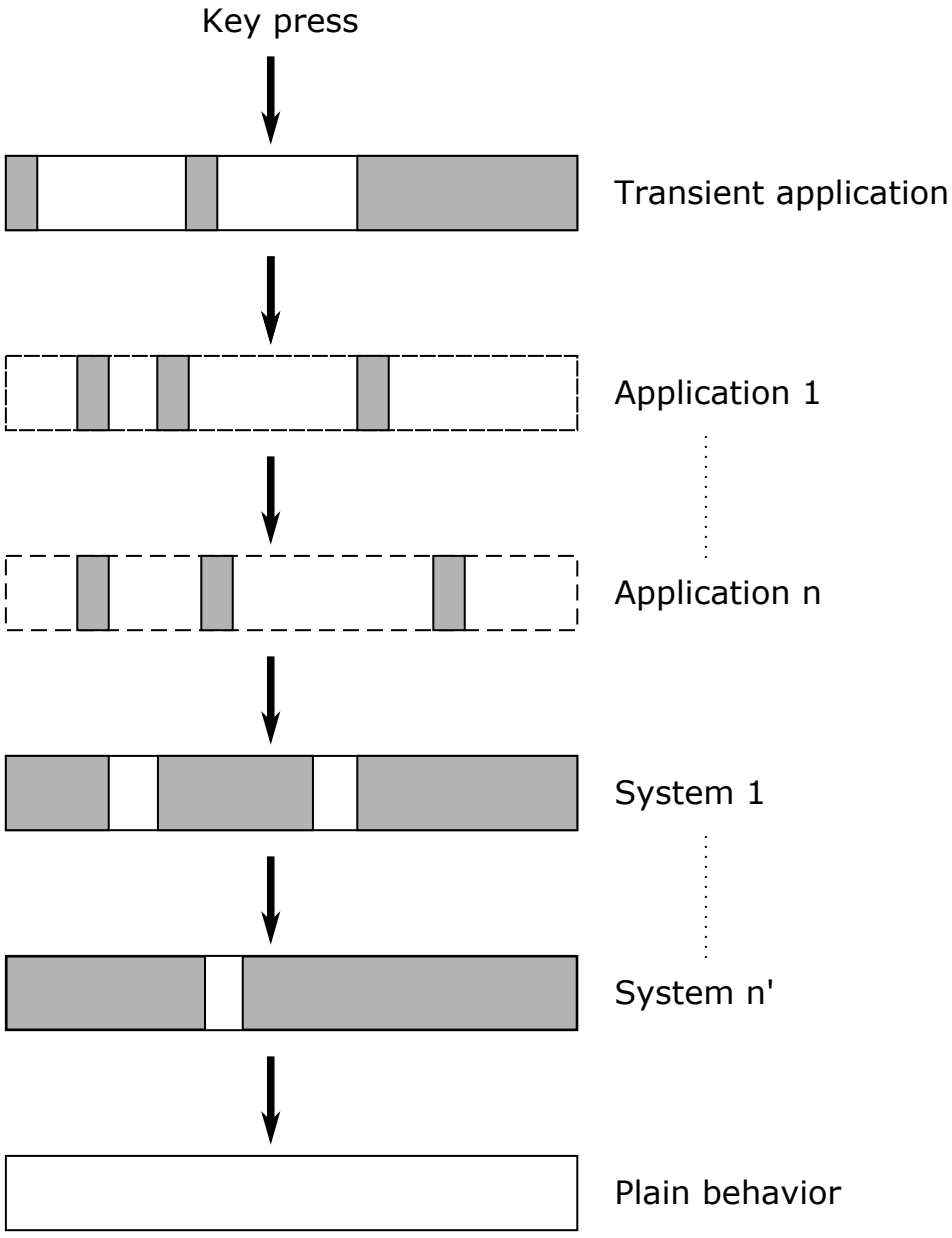


Fig. 2: The shell stack with transient application

very differs compared to application and system shells.

9.3.2 Display handler

This points to the custom display handler that overrides the default display of the stack X register. This is called to replace the built-in provided display of X when appropriate. To get a steadier display it is recommended that functions you implement in your application also ends by updating the display on their own. This is done by calling the `shellDisplay` routine, which this takes care of all possible situations. For example, if a user program is running we do not want to alter the display. Furthermore, the application which your function belongs to may not be the active one, as the user are free to execute any function by name regardless of the state of the shell application stack.

Calling `shellDisplay` at the end of your functions reduces the flicker that results by first having the standard default display of X being replaced by the desired view.

A custom display routine can be used to visualize the floating point value in a different way, e.g. attaching some unit, display as ratio, change the number of display digits in certain situations, or just anything in your imagination. It can also be used for visualizing numbers that are stored on a custom stack, e.g. integers (Ladybug module) or complex numbers. You could even make an application that shows Roman numerals.

In fact, the display routine is rather free to take whatever actions it desires. It is expected to put a value in the display that corresponds to the application it belongs to. However, if the application is something completely non-standard, it may show whatever is appropriate as the default view depending on the state of the application.

Set this field to 0 if a display handler is not defined. In this case nothing happens with the display and you will see the normal display of the X register.

9.3.3 Standard keys

This field points to structure that defines the keyboard layout. This keyboard definition is the replaced standard keyboard.

9.3.4 User keys

This field points to structure that defines the keyboard layout. This keyboard definition is the replaced user keyboard. Normally you will set this to the same value as standard keys.

9.3.5 Alpha keys

This field points to another structure that defines the alpha keyboard layout. If using the default alpha keyboard, set this field to 0.

9.3.6 Name

This fields points to a name of the shell stored in the same way as would do with a text literal with the `MESSL` routine. This should be a short name, typically 3–7 characters.

The intended use is to have a user friendly text representation of the shell. A shell catalog that visualizes the shell stack could make use of it.

9.3.7 Timeout handler

This fields points to a handler which is called when there is a timeout event. This field is only valid for application shells.

Set this field to 0 if no timeout handler is provided.

9.3.8 An example

A Time-Value-Money style shell provides a keyboard with some keys replaced. Its shell definition could look as follows:

```
              .align  4
tvmShell:     .con    AppShell
              .con    0                 ; no display handler defined
              .con    .low12 keyHandler ; standard keys
              .con    .low12 keyHandler ; user keys
              .con    0                 ; alpha keys, use default
              .con    .low12 myName
              .con    0                 ; no timeout handler

              .align  4
myName:       .messl  "TVM"
```

This is an application shell and it provides an alternative keyboard that is used in both standard and user mode. There is no display override as it relies on the standard display of X.

9.4 Key handlers

A shell descriptor has pointers to keyboard handlers which is another structure. It is defined as follows:

```
              .align  4
keyHandler:   gosub   keyKeyboard   ; does not return
              .con    (1 << KeyFlagSparseTable) | (1 << KeyFlagTransientApp) ; flags
              .con    .low12 doDataEntry
              .con    .low12 clearDataEntry ; end data entry
              .con    .low12 keyTable
              .con    .low12 transientTermination
```

This record normally starts with a call to the `keyKeyboard` routine that expects the fields that follows.

The flag field describes certain properties of the keyboard, such as if it allows top rows (A–J) auto assignment, if the keyboard table is sparse and whether this (transient application) should auto terminate on a key that is not handled by it. See `OS4.h` for more details.

The field with `doDataEntry` is the routine that handles data entry. If the keyboard table does not define any data entry keys, you can set this field to zero.

The field with `clearDataEntry` is called whenever data entry is ended. Certain keys end data entry and the routine pointed to by this field is called when that happens. This field should be defined if your application needs to be informed when this happens.

The field with `keyTable` is the actual keyboard table. Refer to *Defining keyboards* for more information about how this is done.

The field `transientTermination` is used when the `KeyFlagTransientApp` bit is set. This field shall be set to either 0 or a valid packed pointer to a routine that does additional things needed on auto termination. The

default behavior removes the transient application and the scratch area, which should suffice in most cases. This routine is called before the transient application and scratch area are removed.

9.4.1 Custom key handler

While keyKeyboard is very convenient when handling keyboard layouts and reassignments, you are not bound to use it. A simple key input routine could use a custom key handler instead:

```
              .align  4
keyHandler:   gosub   clearTimeout
              gosub   exitTransientApp
              c=keys
              rcr     3
              c=0     xs            ; C.X= key code
              gosub   assignKeycode
              bcex                  ; B= floating point key code
              gosub   RSTKB         ; reset key board
              s13=1                 ; continue executing
              golong  RCL           ; push keycode on stack
```

This example is from the KEY function in the Boost module. It has already set up a timeout and a transient application, so these are first removed. Then the key code is fetched and the assignKeycode routine to convert it to a user friendly key code, the same as used in assignments. Finally, it resets the keyboard (wait until key is released) and push the key code on stack and continue execution.

Note: As can be seen the key handler is really a routine. The structure used with keyKeyboard is picked up from the return address left on the stack after the call to it.

9.5 Internal representation

To better understand shells it can be worth looking at how they are represented. A shell consists of seven digits which means that two shells are stored in one register. The seven digit sequence can be broken up in three parts.

9.5.1 Address

The first 4 digits is the address of the shell structure. This means that a shell in theory can be located at any address in the 64K memory space.

Not every address is actually possible. First of all it must be aligned to an even 4-bit word address. This limitation is imposed by the API, not the shell descriptor itself as it can actually handle unaligned addresses. Second, modules can be removed or moved to a different page while the calculator is off. To handle this the page numbers 0 and 1 (which actually points to the mainframe OS pages) have special meanings in the reconfiguration process. No shell can point to these pages. The reconfiguration is executed when the calculator is turned on, see further below.

9.5.2 Kind field

A single digit kind is stored in the descriptor. This is to make it quicker to categorize shells in the stack without having to look it up in the descriptor structure.

9.5.3 XROM number

The last two digits are the XROM number of the owning module. They exist to make the descriptor number unique and for identification of the owning module. As modules can be moved, the page may change and only the 12-bit page offset is fixed. Including the XROM in the descriptor ensures that we both can identify the owning module in case two modules happen to use the same page address for different shells.

An example descriptor is `AC00410` (hex number). The `AC00` is the actual address of the shell descriptor. `4` says it is an application. Finally `10` is 16 decimal, which means it belong to a module with XROM 16, which is currently plugged into page address `A000`.

9.6 Activation

Once you have created a shell structure, activating the shell is done by calling `activateShell`. This routine takes a packed pointer to the shell structure (which is why it needs to be aligned on an even address by 4).

Activation means that a shell descriptor is stored on the shell stack at the topmost location among existing shells of the same kind. It essentially means it becomes the first shell to be consulted of its kind.

You can activate a shell multiple times. Doing so means that it will get moved to become the topmost shell of its kind. In other words, if you activate an application A and then activate other applications they will shadow application A. Activating application A again at this point means it is moved up ahead of the applications that shadows it, making A the active application.

9.7 Deactivation

You can exit a shell using the `exitShell` routine. This will deactivate the shell, bringing any previously shadowed shell in focus again.

9.8 Reclaim at power on

Shells go through a process similar to buffers in the HP-41. At power on they are all marked for removal and it is expected that any plug-in module that wants its shell to survive a power cycle will reclaim it. This is done by calling the `reclaimShell` routine from the power on poll vector.

CHAPTER 10

Applications

An application is a replacement for behavior, which typically includes modifications to the keyboard and the display behavior. You can see it as an alternative mode or working environment.

The HP-41 already contains alternative modes, such as the catalogs and the clock display. Applications as defined here are more formalized compared to the application like behaviors that already exist in the HP-41, which rely on flag settings and various ad-hoc tests.

Technically an application is described by a half register (28 bits) descriptor that is stored on the shell stack kept inside the system buffer. This descriptor identifies and points to a structure (a sequence of words) in the application module. Activating an application is done by pushing its descriptor onto the shell stack, making it the active application. This descriptor is used for finding keyboard and display handlers related to the application, allowing it to define the behavior.

A properly defined application allows for overriding all relevant functionality, while retaining existing behavior that is deemed useful to complement the mode. Thus, you can replace the keyboard to allow for working with complex numbers directly, while still being able to edit and execute programs as usual.

In other words, applications allow you to alter fundamental behavior while preserving the overall HP-41 experience.

10.1 Exiting

It is recommended that exiting an application shell is done with an `EXITAPP` function implemented as follows:

```
            .name   "EXITAPP"
EXITAPP:    golong  exitApp
```

It should be bound to the shifted USER key on an application keyboard. The idea is that the USER key is related to keyboard behavior and it not previously bit used for anything.

10.2 Default display

Your application may have the notion of a default display, which replaces the normal display of the X register.

The application has a pointer to its display routine that is used to display its corresponding default X value. You can decorate or render the display in any way that is suitable.

All functions that are executed returns to mainframe to allow the calculator do the next action. If you are not running a program, displaying a message or entering a multi-key sequence, the default display of X is done. As we cannot alter the built-in behavior, the standard X value is displayed. OS4 will kick in shortly after it and call the display handler of the current application, which replaces the standard display. As this takes a short moment, the standard X value is shown briefly. The effect is display flicker.

10.2.1 Reducing flicker

It is possible to reduce the flicker by having your function exit via OS4. This will cause the default application display to be shown and the message flag is automatically set, blocking the default display of X, the result is no flicker. However, there are many existing functions in the HP-41 that are not aware of OS4, so flicker will sometimes occur, but the overall experience is much better as it happens occasionally rather than all the time.

Normally an MCODE function returns to mainframe with the `RTN` instruction. There are exceptions to this:

1. You used up all four stack levels so the return address to mainframe is no longer on the stack.
2. You want to return using an alternative way, typically `NFRC` (when you do not want to set the push flag) or `NRFKB` (when the function is XKD or during data entry).

To reduce flicker you want to call the `shellDisplay` routine before exiting back to mainframe. There are also a couple of useful entry points `XNFRPU` or `XNFRC` which calls `shellDisplay` before going to the corresponding return point in mainframe.

Note: It is not a good idea to update the display of your own without the control of OS4. The `shellDisplay` routine takes the current active application in account, which may not be the same as the one the function just executed belongs to. This is possible as the user may just have executed your function from the keyboard while having another shell active.

10.2.2 More about the message flag

The message flag is actually given a somewhat new meaning when used this way to reduce flicker. It is actually set when showing an alternative default display for the application and not a message. This is in most situations not a problem, but it matters with the backarrow key. Pressing the backarrow key have different meanings depending on the state of the calculator. If a message is shown backarrow removes the message and reverts back to the default display. If a message is not shown, it acts as clear the X register and disable stack lift.

We can get this behavior in the application, but it requires that we actually know if a message is being shown or the message flag is borrowed for altering the default display of X. Looking at the message flag alone is not enough to tell this. OS4 provides a routine `displayingMessage` for this purpose which answers the question.

In your own `CLX` style routine (bound to the backarrow key) you can use it as follows:

```
              .name   "CLX'"
CLX':         gosub   displayingMessage
              goto    showX         ; (P+1) clear shown message
```

(continues on next page)

(continued from previous page)

```
            s11=0               ; disable stack lift
            ....                ; clear X
```

10.3 Stack lift

You may want to mimic the behavior of the push flag, or stack lift disable (ENTER and CLX functions) for your own environment.

The normal way this is implemented is to have functions to return to NFRPU which always enables stack lift. The few functions that does not do this (ENTER and CLX replacements) need to clear this flag and exit to NFRC instead. Functions that want to leave this flag untouched should also exit to NFRC and leave the push flag untouched.

Note: NFRPU is pushed on the stack before a function is given control, so you can often end your function with a RTN instruction.

This way we set the push flag late and get a sensible default, which is to enable stack lft. However, it is easy to forget about it and just do a RTN when leaving the push flag in the same state would have been more appropriate.

While this is the recommended way, it is possible to revert the logic and set the push flag early and always exit by not touching it. This may make sense if you have support routines to bring up your internal environment as well as an elaborate exit. Still, it can be a good idea to consider doing the normal way as it makes the overall code base more uniform with everything else.

In any case, it can be a good idea to actually test the stack lift behavior of your functions. This is after all a much forgotten detail. The Ladybug module contains test code that inspects the behavior of the push flag for its functions. The HP-41 manuals also specifies in great detail how functions affect (or not) the stack lift flag.

10.4 Data entry

If you application handles numeric data entry in a non-standard way, you need a flag for telling if such data entry is active. The system defines flag 45 for this. You need to share this flag with the system as the Time module may reset this flag due to an alarm.

It is not entirely unlikely that your own environment has its own set of flag and accessing system flag 45 may be awkward. In such case it can be a good idea to copy this particular flag to the CPU flag register together with your own mode flags. The Ladybug does it this way by copying the system data entry to a local flag when entering its data entry code. The internal flag is then written to the system flag before giving control back.

CHAPTER 11

Transient applications

A transient application is a variant of an application that is meant to be somewhat short lived. If you think about the original catalogs (1–3) and the clock display, then you get an idea of what a transient application is.

When a catalog or the clock is active, pressing a key that is not handled by it terminates the mode and interprets the key press outside the mode.

If the shell mechanism would have been available at the time when the catalog or clock was implemented, a transient application would most likely have been used. As the shell mechanism did not exist, various other tricks were used instead.

A transient application may thanks to its single and short-lived existence use various additional resources. There is a scratch area that can be easily obtained from the system buffer for temporary data storage. There is also support for borrowing the interval timer from the Time module (if present). This timer is normally used for the clock display, but thanks to OS4 being in page 4 you can borrow it from the Time module and use it for periodic updates or for implementing a timeout.

11.1 Properties

Transient applications have certain properties that set them apart from other applications:

1. They are not sticky. Instead they are easy to to get out of, typically by pressing the back-arrow key or just any key that is not defined by it.
2. They do not stay active if the power is cycled.
3. The ability to use special resources. As there can only be one that takes precedence over everything else, it is possible to use additional resources like the scratch storage area (as anonymous buffer) or the interval timer.

11.2 Creation

A transient application is defined mostly in the same way as an application. Use the `TransAppShell` kind (instead of `AppShell`).

11.3 Termination

Transient applications normally auto terminate when a key not handled by it is pressed. For this to work with a defined keyboard you need to set the `KeyFlagTransientApp` bit and also define the termination entry, see *Key handlers*.

If you are using your own custom keyboard handler, you need to remove the shell yourself by calling `exitTransientApp`.

11.4 Ideas

To give a better idea about what you can do with the transient application mechanism, we will look at some examples without going too much into details.

11.4.1 Catalogs

The catalog key can be replaced using a system shell. The catalog itself can be implemented as a transient application when the catalog is stopped and waiting for key input. This allows the calculator to go to light sleep when the catalog is stopped while staying active. It also allows for terminating the catalog on a key press not defined by it, very much like the built in original catalogs 1–3.

11.4.2 Periodic display

The clock is a good example of this. We can show an updating time display which is terminated whenever a key is pressed. Another way would be to periodically show a changing value obtained from some external hardware, like a GPS or a multi-meter.

11.4.3 Custom input

Waiting for key input in response to a menu and perhaps dispatching on it in an RPN program could be another way to use for a transient application. Think of it as a variant of the top row auto-assignments, but with better control.

You could also use it for timed input, like `GETKEY` or `GETKEYX`, without having to busy wait. The timeout can be controlled by the interval timer.

11.5 Scratch area

Application shells may need to keep some kind of state. The normal way of doing this is to allocate a buffer. The typical case is an application which may need to store settings or keep some extension to the RPN stack. As there can be multiple applications active in the shell stack, it makes sense to use a buffer for this purpose.

A temporary application shell is typically a temporary mode and as there can be only one such temporary application active at any time using a buffer may feel a bit overkill. For this situation the OS4 module provides a temporary scratch area in the system buffer.

The scratch area can be up to 15 registers large. If a catalog is implemented using a temporary application shell, it starts by running normally to display the catalog entries. When stopped, the catalog can push a transient shell and let the calculator go to sleep, thereby saving power. However, all CPU registers may get clobbered and some storage area is needed to preserve the state. This can be solved using the scratch area and saving the catalog state in it. An alternative is to use the status area in the lower RAM address area, but it is pretty much used up by the operating system and it may be hard to tell what may be safe to use. It is entirely possible that some device may request the calculator to be waked up and serviced, invoking poll vectors and it may be hard to tell what area is really safe. The scratch area provides a solution to this problem.

The downside of the scratch area is that it needs to allocate registers from the free area. This has a potential problem, as there has to be enough registers free, otherwise it will fail.

If there are not enough registers available, you will need to take some actions, which in the simplest case will be to bail out, releasing any allocated resources (exiting the transient application) and exiting via `noRoom` to display an error message.

CHAPTER 12

System shells

System shells are intended for tuning the default behavior of the standard keyboard. Advanced modules from the past like the Zenrom and CCD used various tricks with the partial key sequence mechanism to wedge in alternative behavior. Such tricks may result in certain incompatibilities. Using a system shell provides a cleaner way of making such extensions.

One example would be to replace the assign (ASN) key to provide ways to not only assign by name, but also by XROM numbers or two byte synthetic assignments.

System shells are stored in the shell stack and are consulted in their stacking order. This way the one higher up in the stack has priority over one lower down. In contrast to application shells, where the topmost shell is consulted and the rest are ignored, system shells are consulted in order until a shell that handles the key press are found. Thus, system shells merge their functionality while an application shell shadows the all other applications.

12.1 Display handler

A system shell may also have a display routine, but only the top-most one is ever consulted and that is only if there is no active application.

Note: The display mechanism with system shells is very much unchartered territory. It may be wise to regard it as somewhat preliminary.

CHAPTER 13

Extension handlers

Extension handlers implement a generic message system and can be seen as a more flexible variant of poll vectors. They are descriptors like shells and are stored in the shell stack. Their structure is very different from the shells and they do not participate in key handling or the default display.

Some event is propagated to defined handlers that act on it. The extension handler mechanism is flexible in that it can pass a message specific state in the N register. This can be used for parameters, or some kind of accumulator that is updated by the message handlers where the final state may be passed back as a result. A message handler can choose to act on the message on its own and prevent further propagation of the message to other message handlers, or it may allow other handlers to also see the message.

This mechanism is a lot more flexible than poll vectors, but it comes with a somewhat higher overhead as it needs to scan the stack and then scan lists for appropriate handlers to call. Compare this to poll vectors that simply scan a fixed offset on each of the ROM pages.

The extensible catalog mechanism in the Boost module (CAT' function) sends a message to extension handlers that a given catalog number is desired. Each extension handler checks if this catalog number is something it implements and in that cases it prevents further propagation of the message and the requested catalog is shown. If no message handler implements the catalog, the message will return to the originator (CAT' function) which then performs some suitable default action.

Other events can inform that the shell stack is altered or that XEQ is starting RPN execution in a new location so that any pending returns addresses on the stack should be cleared. This is done today internally by the firmware, but there is no way for a module to get informed about it, as HP never defined a poll vector for such seemingly specialized event. However, today the extension handler makes it possible for an alternative XEQ' function to send out such notification by using the extension handler mechanism.

13.1 Extension structure

The data structure used by extension points is very different from the various shells. Only the first identifier word is shared with them. The rest is just a list of the message numbers it will handle coupled with a pointer to the handler itself.

The extension structure is fairly simple:

```
              .align  4
extensionHandlers:
              .con    GenericExtension
              .con    ExtensionCAT
              .con    .low12 catHandler
              ...
              .con    ExtensionListEnd
```

As usual it needs to be aligned. The first word must be `GenericExtension` to separate it from being some kind of shell. After this follows pairs of the message identity (number) and the corresponding handler. The table must end with `ExtensionListEnd`.

Using a list means that a module only needs to define one extension structure and one shell descriptor, which saves precious RAM space.

Note: If you want to use more than one record in order to provide optional functionality in groups that are independently activated, it can make sense to do that using multiple records.

Activation of the extension handlers can be done from the deep wake up poll vector:

```
#include "mainframe.h"
#include "OS4.h"

              ...

              .section pollVectorArea
deepWake:     n=c
              ldi     .low12 extensionHandlers
              gosub   activateShell
              goto    pollReturn    ; (P+1) failed, not enough memory
                                    ; (P+2) success
pollReturn:   gosub   LDSST0
              c=n
              golong  RMCK10

;;; ***************************************************************************
;;;
;;; Poll vectors, module identifier and checksum
;;;
;;; ***************************************************************************

              .con    0             ; Pause
              .con    0             ; Running
              .con    0             ; Wake w/o key
              .con    0             ; Powoff
              .con    0             ; I/O
              goto    deepWake      ; Deep wake-up
              goto    deepWake      ; Memory lost
              .con    ...           ; module identifier
              .con    ...
              .con    ...
              .con    ...
              .con    0             ; checksum position
```

The routine that sends the message does it using the `sendMessage` routine, which takes the message number and optionally some message specific data in the `N` register.

Any called routine can inspect, update or return the value in N. Each message type defines on its own how N is used. A message handler can prevent further propagation of the message by popping the return stack. If further message propagation is desired, then it should not pop the return stack and also preserve the M register as it contains the shell stack traversal state which is needed to properly pass it the next handler. In both cases return using a RTN instruction when done, or exit in some message specific way.

How many subroutine levels you can use depends on the context in which the message was sent. It is recommended to use as few as possible and to test it. Basically, if you do not want further message processing, you know that you gained one level on the stack when the return address was dropped.

Here is how a catHandler could look:

```
              .public catHandler
              .align  4
catHandler:   c=n
              a=c     x
              ldi     16
              ?a#c    x             ; cat 16?
              gonc    doCat16       ; yes
              c=c+1   x
              ?a#c                  ; cat 17?
              rtnc                  ; no, not one of mine
doCat17:      ...

doCat16:      ...
```

This takes the catalog number from N.X which is where the CAT' function places it. If the passed number is not one of mine, return to the caller which is the dispatch loop. It will continue scanning for other catalog handlers. As the scan state is kept in M, we must not touch it.

The actual catalog implementation should use SPOPND, but it may not be strictly needed if we never return from the catalog handler code. A catalog exits via QUTCAT (quit catalog) which jumps to NFRKB which is one of the entry points for function return. The return address will never be used and is going to pushed off the top of the 4-level return stack at some point in the future.

CHAPTER 14

Defining keyboards

In this chapter we will look at how to define keyboards. A keyboard is essentially a mapping of a key code to a function. The key code is in 0–79 form, which means that we transform a key press to an index starting from 0 and ending with 79. There are no gaps in this sequence, so it can become a simple array lookup.

OS4 allows for two ways to define a keyboard. By using a full definition of all 80 keys or a linear search table.

14.1 Defining a key

The function that is bound to a key is described by a single ROM word.

The built-in keyboards only need to access functions that are built-in, there are no XROM or XXROM functions on these keyboards. For our purposes we want to be able to also use such functions on our keyboards.

As with the built-in keyboards, a single ROM word is used to describe the function bound to a key. In order to cover XXROM function an extension record is also used.

The function value 0 means that the key is not defined by this keyboard. Scanning to the end of a linear table without find the key definition also means it is not defined. An undefined function means that we should keep searching through remaining system shells in the shell stack, eventually reaching the internal base definition.

As with the built-in keyboards, we use the upper two bits in the 10-bit word to classify the key as follows:

0. XROM function (1–63), XXROM if 64–255
1. Digit entry
2. Built-in function, ending data entry
3. Built-in function, not ending data entry

As the value 0 means the key is not defined, the XROM function with identity 0 is not possible to bind to a key. This is normally the XROM header, which you may use for special purposes but it cannot be bound to a custom keyboard.

For XXROM functions the stored value (64–255) describes the XXROM function. As this range is way too short (there can be 4096 XXROM identities), it points to an extension word:

```
keyTable:      ...
               .con    64 + keySecondary - .
               ...

keySecondary:  .con    0            ; secondary (range 0-1023)
               .con    offsetFAT1 + (myEntryFAT1 - FAT1Start) >> 1
```

The value stored is 64 plus the forward offset to the extension record which consists of two words.

The first word is normally 0 for a secondary function. The second word is the function number of the secondary function (0–1023).

The calculation above uses various labels where the `offsetFAT1` value is the first function number in a specific secondary FAT table while `myEntryFAT1` and `FAT1Start` are labels inside that FAT that instructs the assembler to calculate the desired function number.

Note: If you wonder about the offset calculation, the dot is just the current location counter. This makes it easy to calculate the distance to the extension record and by adding 64 we tell OS4 it is an extension record.

If you actually need to use secondary functions above 1023 you need to store the upper two bits in bit 7 and 6 of the first word:

```
keySecondary:  .con    ((offsetFAT1 + ((myEntryFAT1 - FAT1Start) >> 1)) & 0xc00) >> 4
               .con    (offsetFAT1 + ((myEntryFAT1 - FAT1Start) >> 1)) & 0x3ff
```

The extension record can also describe an arbitrary two byte function, which makes it possible to bind an XROM function of another module or a synthetic function such as `RCL M` to a key. In this case the upper two bits in the first extension word must be non-zero to mark that it is a two byte function:

```
rcl_M:         .con    0x300 + 0x90  ; RCL
               .con    0x75          ; postfix M
XROM_20_01:    .con    0x300 + 0xa5  ; XROM 20-23
               .con    0x01          ; function 1 in XROM 20
```

14.2 Sparse keyboard tables

Sparse keyboard tables are useful when only a few keys are defined. They are just a simple linear search table where each entry is a key code (0–79 form) followed by its function definition.

As usual you need to align the table as it will be pointed to from another record using a packed pointer. The table also needs an end marker where the upper bits in the word is set:

```
               .section table, rodata
               .align  4
sysKeyTable:   .con    11            ; CAT key
               KeyEntry myCAT
               .con    18            ; XEQ key
               .con    64 + xeqSecondary - .
               .con    26            ; ASN key
               KeyEntry myASN
               .con    0x100         ; end of table
```

14.3 Full keyboard tables

A full keyboard defines all 80 keys using an array. This is done very similar to how the built-in keyboard are defined, but we use OS4 style function definitions:

```
                .section KeyTable, rodata
                .align  4
keyTable:
                ;; Logical column 0
                .con    0x10a           ; SIGMA+  (A digit)
                .con    0x10f           ; X<>Y    (F digit here)
                .con    BuiltinKeyKeepDigitEntry(0x0e) ; SHIFT
                KeyEntry ENTERI         ; ENTER^
                KeyEntry SUB            ; -
                KeyEntry ADD            ; +
                KeyEntry MUL            ; *
                KeyEntry DIV            ; /

                ;; Logical column 0, shifted
                KeyEntry SL             ; SIGMA+
                KeyEntry SWAPI          ; X<>Y
                .con    BuiltinKeyKeepDigitEntry(0x0e) ; SHIFT
                .con    0               ; CATALOG
                KeyEntry CMP            ; -
                KeyEntry TST            ; +
                KeyEntry DMUL           ; *
                KeyEntry DDIV           ; /

                ;; Logical column 1
                .con    0x10b           ; 1/X  (B digit)
                KeyEntry Hex            ; RDN
                .con    0               ; XEQ
                .con    0               ; right half of enter key
                .con    0x107           ; 7
                .con    0x104           ; 4
                .con    0x101           ; 1
                .con    0x100           ; 0
                ...
```

14.4 Anonymous keys

For catalogs and other transient applications you may want to have special functions only available in that mode. Typical examples are single step, start running the catalog and perhaps some special functions available only inside that transient application.

Naming that function and allocating an XROM or XXROM for it may seem like a lot of overhead. OS4 provides a way of creating anonymous execute direct functions that are only present inside that mode.

From the user point of view, it works like any execute direct function, or special key press, e.g. pressing the C key to clear the current entry in a busy waiting catalog. In both cases, there is no preview of the function and it is not programmable.

Note: In the built in catalogs 1–3 this is handled by execute direct functions like SST. For busy waiting catalogs 4-6 it is a simple key dispatch loop without any real function. The user experience of them are essentially

identical even though they are implemented in very different ways.

OS4 provides a way to generate a special execute direct function form that are well suited for this purpose. They only work with sparse keyboards, which is not a huge limitation as such transient applications typically only binds perhaps 5-10 functions. Here is an example of how a catalog keyboard can look like:

```
              .section table, rodata
              .align  4
              .public keyTableCAT7
keyTableCAT7: .con    40            ; SQRT
              .con    KeyXKD
              .con    66            ; SST
              .con    KeyXKD
              .con    74            ; BST
              .con    KeyXKD
              .con    67            ; <-
              .con    KeyXKD
              .con    55            ; R/S
              .con    KeyXKD
              .con    2             ; Shift
              .con    0x30e
              .con    10            ; Shifted shift
              .con    0x30e
              .con    70            ; User
              .con    0x30c
              .con    78            ; Shifted user
              .con    0x30c
              .con    0x100         ; end of table

              ;; The XKD pointers
              .extern CAT7_Clear, CAT7_SST, CAT7_BST, CAT7_BACKARROW, CAT7_RUN
              .con    .low12 CAT7_Clear
              .con    .low12 CAT7_SST
              .con    .low12 CAT7_BST
              .con    .low12 CAT7_BACKARROW
              .con    .low12 CAT7_RUN
```

All such functions have the special value `KeyXKD` and the key table is immediately followed by a table of packed pointers to the key handler routines. The OS4 key table scanner simply counts the number of `KeyXKD` values seen while scanning the table. If the key pressed is `KeyXKD`, the accumulated count is added to the start of the execute direct pointer table to determine the correct handler. Thus, there are no padding or gaps in the execute direct table in case there are real functions intermixed in the sparse key table.

Note: The reason why this only works for sparse key tables are twofold. First, the `KeyXKD` value is 0, which is already taken for meaning an empty key in a full keyboard. Second, the following table relies on that we have visited all entries before it. Doing something similar on a full keyboard would either means that we would need to scan the up to 80 entries long table, or have a second table of the same size, which would be rather wasteful. It is also typical that transient applications where this is useful only defines a small number of keys.

CHAPTER 15

Secondary functions

Secondary functions are additional XROM functions allows for up to 4096 additional functions that can be used if the provided 64 primary functions are not enough.

The XEQ' replacement function in the Boost module can be used to access secondary functions by name, just as any other function. The replacement ASN' function also allows them to be assigned to keys in essentially the same way as ordinary XROM functions.

Storing secondary functions in programs is also possible using the semi-merged functionality, they can also have postfix arguments in programs.

It can be worth pointing out that once you have set up the secondary FAT tables, everything is taken care of automatically by the OS4 and Boost extension modules. There is no need to write any additional specific code.

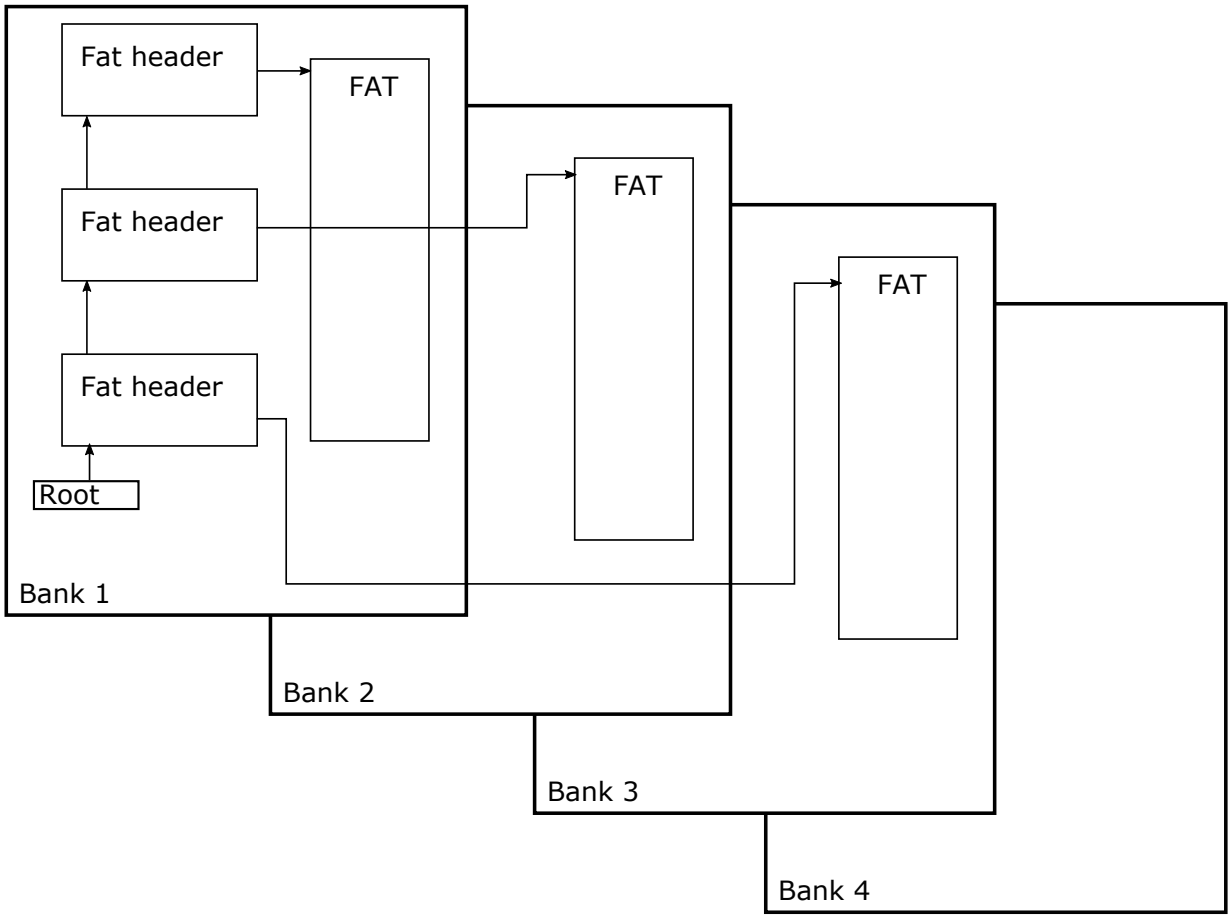
15.1 Secondary FAT

The secondary function address table (FAT) consists of two parts. The secondary FAT header is a linked list of structures in the primary bank. This structure is rather small, typically 6–7 words. The actual FAT table is pointed to by the header structure and is similar to the ordinary FAT. This table requires more space as it uses two words for each function entry. It can be located in a secondary bank, but all functions in it must be (or at least start) in the same bank.

Each secondary FAT header table has an ordinary prefix XROM function associated with it. This function is used to represent secondary functions in RPN programs. As a result each secondary FAT table can hold up to 256 secondary functions.

15.2 Execution by name

To find a secondary function by name, a similar routine to ASRCH in mainframe is needed. As it is not possible to alter ASRCH which is in mainframe ROM, OS4 provides a similar function that is aware of secondary FATs. Execution by name is done by XEQ and the easiest way is to use a replacement of XEQ that uses the XASRCH


Fat header
FAT
Fat header
FAT
Fat header
FAT
Root
Bank 1
Bank 2
Bank 3
Bank 4

routine in OS4. The XEQ' function in the Boost module is an example of this. As it is implemented using a system shell it automatically replaces the ordinary XEQ key when Boost module is inserted.

15.3 Assignments

Assignments of secondary functions are stored in the OS4 system buffer. The XROM identity is stored together with the secondary function index. An assignment consumes a half register (seven nibbles), using two nibbles for the XROM identity, three nibbles for the secondary function identity and finally 2 nibbles for the key code. There is no leading F0 marker in this register as all nibbles are needed for the assignments. This works as they are stored inside the system buffer.

As with the ordinary assignments there are bitmaps secondary assignments for fast lookup if a key is assigned. These are also stored in the OS4 system buffer, consuming two additional registers when the first assignment is created.

Assignments are easily created using the ASN' function in the Boost module, which is OS4 aware. This works very similar to the already discussed XEQ' function. ASN' can handle assignments of both primary and secondary functions.

As the OS4 module controls the keyboard it will also look for assignments when appropriate. This will look for both primary and secondary assignments and if both are present, the primary assignment takes precedence. Normally they are no collisions, but in case you load keys from storage medium you may get duplicate assignments. This is because such mechanisms predates OS4 and only knows about primary assignments.

This has a couple of caveats. If you manage to remove such assignments outside the control of OS4 (by using another load key assignments that replaces the current ones), the secondary assignments that were shadowed will appear again. Loading keys from storage this way will not reclaim the memory used for (shadowed) secondary assignments. Replacing keys this way only works on primary assignments. The secondary assignments are always retained in a merge keys fashion in such cases.

As a workaround, the function CLKYSEC in the Boost module can be used to remove all secondary assignments. It simply calls the clearSecondaryAssignments routine in the OS4 API.

A secondary assignment that belongs to a module which is removed shows up as XXROM ii,kkk if the key is pressed and held.

Note: Secondary function on assigned keys are only searched for when there is at least one shell in the shell stack. If you have secondary assignments and remove every application and system shell from the stack, secondary assignments become invisible. This is unlikely to happen in reality as you need to have the ASN' function from the Boost module (or similar) to create them and it will always put a system shell on the shell stack.

15.4 In RPN programs

In program mode a normal XROM works as a prefix for representing secondary functions. This is followed by a text literal that holds the adjusted secondary function index. This index is based on the secondary FAT it belongs to (0–255). Each secondary FAT header table has such paired XROM that acts as the prefix for it.

You need to set aside one XROM function for each secondary FAT, which is defined in the following way:

```
              .name   "(BPFX2)"      ; short name for prefix function
Prefix2:      gosub   runSecondary   ; Must be first!
              .con    1              ; I am secondary prefix XROM 6,1
              ;; pops return address and never comes back
```

The name is not so important as it is normally suppressed by the decorated view of the secondary function. It can be seen briefly and also when stepping an RPN program by keeping the SST key down. The function is just a call to runSecondary followed its own XROM function number. This number is matched with the value stored in secondary FAT header tables when scanning for a matching table.

As with all semi-merged functions, the fully decorated function is shown in program mode followed by a text literal that is automatically skipped when executed.

If a secondary function in program belongs to a module that is not plugged in, it is shown as an XROM (the prefix XROM) followed by the text literal. This is because in program memory the tables in the module must be present to decode the real function number. When assigned to a key, the actual full secondary index number is stored in the assignment. This allows it to be displayed as XXROM when the key is kept pressed, also when the module is removed.

Note: A secondary function bound to a key that belongs to a module that is not plugged in cannot be entered in a program. This is also because the use of a XROM prefix function requires the secondary FAT to determine which XROM acts as prefix and also the base index for that table.

15.5 Defining

A secondary function is defined as any normal XROM function with a name and an entry point. The name can have upper bits set to tell that it is a prompting function. The first words at the entry point can be NOP instructions (000) to indicate a non-programmable and optionally execute direct (XKD).

Secondary functions can start in any bank. They do not have to be in the primary bank as is the case with normal XROM functions. You should however exit with the primary bank enabled.

15.6 Secondary FAT structure

The secondary FAT structure is different compared to the ordinary FAT and consists of several parts:

1. A root pointer to the secondary FAT start is a packed pointer located at address 0xFC2 in the module page. As this location may contain other data in modules that are not OS4 aware, the module page image must also mark in the module ID field that this location has a valid root pointer, this is described next.
2. The module identity area consists of 4 words located at 0xFFB–0xFFE in the module page. It forms a four letter module identity. The upper two bits have special meanings as follows. 0xFFD location tells whether the module is banked (this is defined and recommended by HP). The upper two bits in the 0xFFE word tells whether there is a secondary FAT structure or not. If any of these two bits are set, the word at 0xFC2 is assumed to be a packed pointer to the start of the secondary FAT header structure.
3. The secondary FAT headers are small records that must be located in the primary bank. This forms a linked list of records. Each record has a packed pointer to the next record and some additional information described below.
4. The actual secondary FAT is pointed to from the secondary FAT header. This FAT is defined in the same way as the ordinary XROM FAT. It can be located in any bank, but all functions in it must be (or at least start) in the same bank. This bank is enabled by the bank switch routine in the secondary FAT header that points to it.

The bank switch routine should either be RTN for a primary bank, or one of the ENROM instructions followed by a RTN. That RTN instruction must be located at the following address in the bank it enables. No registers should be affected by this code snippet.

15.6.1 Root pointer

The root pointer is just a packed pointer stored at location `0xFC2`:

```
              .section PlaceMeAtFC2
fatRoot:      .con    .low12 secondary1 ; Root pointer for secondary FAT headers
```

You also need to set one of the upper bits in the module identity area in the word immediately before the checksum:

```
;;; ************************************************************************
;;;
;;; Poll vectors, module identifier and checksum for primary bank
;;;
;;; ************************************************************************

              .section pollVectors
              nop                   ; Pause
              nop                   ; Running
              nop                   ; Wake w/o key
              nop                   ; Powoff
              nop                   ; I/O
              goto    deepWake      ; Deep wake-up
              goto    deepWake      ; Memory lost
              .con    1             ; A
              .con    '1'           ; 1
              .con    0x20f         ; O (tagged for having banks)
              .con    0x202         ; B (tagged as having secondaries)
              .con    0             ; checksum position
```

15.6.2 Secondary FAT header

The secondary FAT header are small records that must be in the primary bank. They form a linked list starting from root pointer. The first word points to the next secondary FAT header record and the last one has this word set to 0.

```
;;; * First secondary FAT header, serving bank 1
              .section Secondary1, reorder
              .align  4
secondary1:   .con    .low12 secondary2 ; pointer to next table
              .con    (FAT1End - FAT1Start) / 2
              .con    0             ; prefix XROM (XROM 6,0 - ROM header)
              .con    0             ; start index
              .con    .low12 FAT1Start
              rtn                   ; this one is in bank 1,
                                    ; no need to switch bank

;;; * Second secondary FAT header, serving bank 2

              .section Secondary1, reorder
              .align  4
secondary2:   .con    0             ; no next table
              .con    (FAT2End - FAT2Start) / 2
              .con    1             ; prefix XROM (XROM 6,1 - (BPFX2))
              .con    256           ; start index
              .con    .low12 FAT2Start
```

(continues on next page)

(continued from previous page)

```
                switchBank 2            ; this one is in bank 2
                rtn
```

The second field is the number of entries in the secondary FAT we describe. This is used for range checking.

The prefix XROM field is the function number in the main XROM of this module that serves as the prefix XROM used in programs.

The start index is the function number of the first secondary function stored in this table. Each prefix XROM can serve up to 256 functions and we have a full range of 4096 secondary functions. Thus, we may just step this by 256 for each secondary FAT header, which reserves space for adding more functions later without affecting any index in other tables. We essentially leave gaps for future secondary function to be appended to the overall secondary function table.

A packed pointer to the actual FAT follows. The actual FAT pointed to can be located in any bank. The next address holds a code snippet that enables the bank it is located in. If it is located in the primary bank, no change is needed so it can just return. To switch bank you need to use the appropriate `ENROM` instruction followed by a `RTN` instruction that must be in the bank it switches to! This can be accomplished using some clever code arrangement. The easiest way is to use the `switchBank` macro which is defined as follows:

```
switchBank:     .macro  n
                enrom\n
10$:
                .section Code\n
                .shadow 10$
                .endm
```

15.6.3 Secondary FAT table

The actual secondary FAT looks exactly the same as the ordinary FAT that starts at address `0x002` in a module page. The secondary FAT can be located anywhere, but it must be aligned as it is pointed out from the secondary FAT header using a packed pointer:

```
                .section Secondary2
                .align  4
FAT2Start:      .fat    COMPILE
                .fat    RAMED
FAT2End:        .con    0,0
```

Here we define two functions and terminate the table using two zero values.

15.7 Design constraints

The linked list of FAT secondary FAT headers allows for binding XROM prefixes to a range of secondary functions. These prefix XROM functions are needed when secondary functions are stored in programs. To save space in RPN program memory, a single byte is used as the identity, which means that you should not have more than 256 secondary functions in each FAT. Allowing more functions to be handled by a single XROM prefix would cost an extra byte of program memory for each secondary function. It was judged better to use an couple of such XROM prefixes and save program space.

15.8 Bank switching

Enabling the appropriate bank for secondary functions is done automatically once you have set up the secondary FAT structure. Switching back to the primary bank is done by calling the ENBNK1 routine as defined by HP. It shall be at page offset address FC7. HP only defined two bank switchers and this was later expanded to four, the full layout is as follows:

```
ENBNK3:     enrom3
            rtn
ENBNK4:     enrom4
            rtn
ENBNK1:     enrom1
            rtn
ENBNK2:     enrom2
            rtn
```

This block of code should at page address 0xFC3 to 0xFCA in every bank. If you are not using all banks, replace the unused switchers with two RTN instructions (or NOP and RTN).

You should also set at least one of the two upper bits in page address 0xFFD to mark that the page is bank switched. Other ROMs that want to enable different pages in your module shall inspect these bits to determine if the page has multiple banks and may then use the bank switch routines above to switch bank.

OS4 uses this technique to inspect secondary FATs which may be located in other banks than the primary. However, OS4 only uses the ENBNK1 routine as it uses the bank switch routine in the secondary FAT header to enable other banks.

15.9 Error exits with banks

Many functions need to perform some sanity checking on input and exit using appropriate error message routines to notify the user if something is wrong with the input. One such example is ERRDE which displays DATA ERROR.

Typically you perform such sanity checking of input early in your function, as there is no point of doing any work if some input is bad. However, jumping to an error exit like his leaves the current bank of your module page selected. This is a problem if your function is in some other bank than the first one, as you should always leave the first bank enabled when you are not in control.

This can of course be solved by switching bank and then jumping to the error handler, but that is in reality somewhat tricky. To make it simple, OS4 provides a set of routines that display normal error messages and restores the bank of your module for you. In your function you can do something like this:

```
                .name "EXAMPLE"
EXAMPLE:        ...
                ?a#c  x              ; some test
                gsubc   ERRDE_resetMyBank
```

Note that instead of jumping to the error handler, it uses a GOSUB (with carry set condition in this example) to the error exit routine. The called routine pops the return stack to get the page number of your function. It then inspects the module page and calls the defined bank switch entry point ENBNK1 to enable bank 1 in your page, then it jumps to the normal ERRDE routine.

See *Error handling* for existing error exit routines.

15.9.1 Function exit

A related routine `resetMyBank` exists. It resets your bank and returns. This can be used as a normal exit from your secondary bank instead of using a `RTN` instruction. As with the error handlers above you need to use a `GOSUB` instruction:

```
done:           gosub   resetMyBank   ; reset my bank and return
```

The `resetMyBank` first pops the return address to get your module page, enables bank 1 for you by calling your `ENBNK1` and then it returns using the next return address on the stack.

CHAPTER 16

Semi-merged functions

Semi-merged function are prompting XROM functions that take postfix arguments. This is much like many built-in functions, e.g. RCL.

Instructions such as RCL are made up from two parts, the instruction and its postfix argument. On the HP-41 they are displayed in programs as fully merged steps.

The problem with XROM functions is that they cannot be entered in a program together with a postfix argument. The semi-merged feature makes this possible.

16.1 Postfix operands

As we cannot alter the mainframe there is no way we can store a fully merged XROM function in a program step. What we can do is storing it in two parts, first the XROM function and then its argument. As the argument can be any byte, it is not possible to store it raw by itself, as it may grab following bytes making the program impossible to view and edit, and it can even corrupt program memory.

With semi-merged functions we wrap the postfix byte using a text literal. This means that there are actually two program steps:

```
10 ...
11 SL 36
12 "$"
13 ...
```

Note that the SL 36 instruction is fully shown as a merged instruction, but it is followed by its wrapped postfix byte (36 corresponds to ASCII $).

This also works when using indirect addressing:

```
10 ...
11 SL IND Z
12 "*"
13 ...
```

As the SL instruction is a two byte XROM instruction followed by a single letter ASCII constant, the whole instruction requires four bytes of program memory.

When executed, the text literal is simply skipped and has no effect on the alpha register.

Note: It is intentional that the postfix byte is shown. While it can be possible to hide it somewhat, it is judged to be better to actually show what is going on. This provides better control over program memory editing, as the postfix part actually does take a program step and will not be considered merged when following an instruction that skips the next line. You may still be able to use it after such skip instruction, but it will execute the text literal in this case, altering the alpha register.

16.1.1 Default operand

If the postfix operand is missing, the instruction reverts to a default behavior. For a shift instruction, it means shift one step:

```
10 ...
11 SL 01
12 ...        ; not a single letter text literal
```

Such instruction costs two bytes (the XROM itself without any postfix operand). As it is a single instruction, it also works well following a test instruction.

If you enter the SL 01 instruction, it takes advantage of the default and does not store a postfix byte in program memory.

If you delete the postfix operand from program memory, the instruction that used it will change to its default behavior, which can be seen when the instruction is shown.

Note: Some care is needed when using default behavior with prompting instructions. It will still look for its argument and if you have a single character alpha constant that you intended to be an alpha constant, it will become part of the previous instruction. This should seldom happen, but if it does, the easiest way to deal with it is probably to rearrange instructions.

16.2 Single stepping

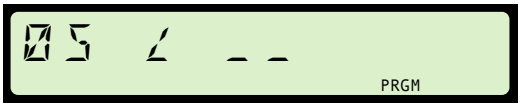
When you single step a semi-merged instruction in run mode (to execute the program step by step), it works properly, but visual feedback of the instruction when the SST key is pressed and held is just the bare instruction without any postfix operand.

16.3 Dual operand functions

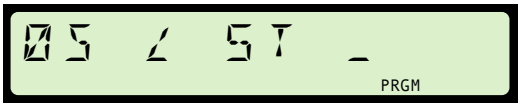
With OS4 you are not limited to a single postfix operand, a function can have two. This is useful things like comparisons or exchange between two registers.

In the following example the < function is used to compare two register operands. All dual operand functions are entered with the function first followed by the two postfix operands. In program memory a dual semi-merged program step is shown infix with the function name between its two operands. A bit in the control word can be used to specify that it should be followed by a question mark, meant to indicate that it will optionally skip the next program line.

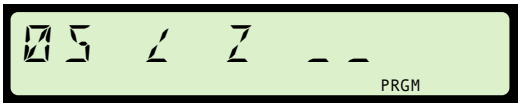
When entered the function is always displayed first followed by the prompt underscores:



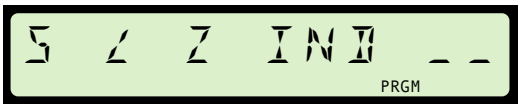
Then if we start entering a stack operand:



When complete the first operand is followed by the second prompt:



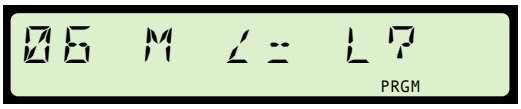
Here the second operand is partially entered:



When the function is complete it will show the text literal to accept the following program step. Here shown with some weird characters due to limitations in the font used. On a real calculator they will various characters, often with all segments on.

If we now back stop to the previous line we can see the decorated function. In this case it is somewhat too long for the display making the line number scroll off the display:

The prompt mechanism is the same as the built-in one. Synthetic status register operands cannot be keyed in. Using synthetic techniques or hex editing the program makes it possible to have them in a program:



Note: If you are observant you may have noticed that the program line number is the same for the semi-merged step being entered as the text literal being shown when done. This is because two program steps are inserted up front in program memory which advances the program line counter twice. The display shows the decorated semi-merged function being entered with the current line number, which corresponds to the program step of the text literal.

16.4 Secondary functions as semi-merged

Secondary functions can also have semi-merged arguments. This works for both single and dual operands. It also works to have the function located in a secondary bank. Thus, the most complicated thing you can put

together is a dual argument secondary function in a secondary bank.

As a secondary function uses a text literal to indicate which function it is, a dual secondary function requires a text literal with three bytes. The first byte is the secondary function number, the remaining two are the arguments. OS4 will merge all wrapped text literals to a single three character text literal in this case.

16.5 Defining a function

A semi-merged function must start with a specific prelude:

```
                .name   "XRCL"
XRCL:           nop
                nop
                gosub   argument
                .con    00 + SEMI_MERGED_NO_STACK
                ...
```

The first thing to observe is that there are no bits set in the name header. This function is not marked as a prompting function.

The first two `NOP` instructions signal that this is a non-programmable execute direct function (XKD).

Even though this is marked as a non-programmable function it can be entered in a program. What happens is that when `argument` detects that it is executed in program mode, it inserts the appropriate program steps and alters the display to make it look as if it was entered in the program and is now prompting for its argument. The calculator is put in a state where it can accept input with the look of the display that you might expect from such action.

The execute direct feature is there to ensure that the function executes immediately on key down. If you press and hold the normal `RCL` key, it will put up its name and prompt immediately, you do not need to release the key. A function such as `SIN` will go through a timeout and cause a `NULL` message if held for long enough.

Using execute direct means that we can mimic the behavior of `RCL` as it acts immediately on key press and it will not go through the `NULL` test.

Note: The execute direct feature is partially broken with XROM functions in the HP-41 mainframe and only works properly in program mode. Outside program mode it will actually go through the NULL test, but there is in practice no real harm from this.

Note: If you have the 41CL, there is an updated mainframe firmware available that corrects this bug.

The `argument` routine is what makes this function become semi-merged, or at least half of it. As mentioned, the purpose of this routine is to put the calculator in the proper state to prompt for an argument for the semi-merged function. It is followed by a control word which is the default postfix argument byte for this function and the upper bits are used to signal if we accept direct stack arguments or not.

In program mode this function does not return. In run-mode it will appear as this function returns with the argument filled in by the user in the `ST`, `G` and `C[1:0]` registers.

Note: Technically, the whole function actually re-executes in run-mode and the state is set up so that the second time it picks up the entered argument. In a running program it picks the argument from the following text literal in program memory and advances the program pointer to skip the text literal.

The second half of the semi-merged feature is not seen at all in the function prelude. It consists of a hook that is called in program mode for each program line. This hook does two things. First, it detects when we are entering a semi-merged argument and will ensure the display looks right and the program memory is written to in the correct way, forming the text literal and also prune it if the default argument is entered. Second, when not entering a program step it will display semi-merged program steps in the decorated fashion.

16.5.1 Dual arguments

Defining a function with dual arguments is similar to the single argument variant. A prelude looks something like this:

```
                .name   "<="
LE:             nop
                nop
                gosub   dualArgument
                .con    SEMI_MERGED_QMARK
                ...
```

The routine changes to be `dualArgument` and there is no longer a default postfix argument. This word now only holds flags as defined in the `OS4.h`. The function above is marked to have a trailing question mark in the name to indicate that this function optionally skips a step. There are also flag bits that allow for telling if stack arguments are accepted or not for each of the two arguments.

The argument bytes are returned in `A[3:2]` (first argument) and `A[1:0]` (second argument).

16.6 Decoding postfix operands

Mainframe contains code to decode postfix operands, but it falls short in several ways and is not always suitable for your needs.

The closest you will come to have a function that parses a postfix operand is `TONSTF`, but it is meant to be used by single digit functions like `TONE` and `FIX`. As a result, it will throw an error if you have a two-digit operand. The other built-in code that deals with operands are not available as subroutines.

The two support routines related to this are `ADRFCH` and `BCDBIN`. `ADRFCH` handles reading register operands which is useful for indirect operands. If you use `ADRFCH` you will need to convert the read register value, which is a floating point number, to binary and `BCDBIN` is useful for this. However, `BCDBIN` cannot handle numbers larger than 999.

OS4 provides two routines that help with decoding postfix operands. `postfix4095` is similar to `TONSTF` but offers as much range as possible. For a direct operand it means 0–127 and for indirect it will handle 0–4095, see *postfix4095*.

The second routine is `XBCDBIN` which is like `BCDBIN`, but can handle a 12-bits range 0–4095.

16.7 Rolling your own

The above postfix operands are simple to use, but what if you really need something very different? One example is the Ladybug module which stores integer literals as program steps.

In the Ladybug module this is implemented by special handling numeric input which is stored gradually into a program step as it is keyed. The actual display is done using the `xargument` form:

```
              .section Code, reorder
              .name   "#LIT"
Literal:      gosub   xargument     ; mark as special form
              goto    20$           ; display it
              ?s13=1                ; running?
```

The GOSUB to the xargument entry marks that this is a special form. The address following the GOSUB is called when it should be displayed in program memory. You need to implement the code to actually display the program step on your own.

CHAPTER 17

Partial key sequences

Some functions prompt for arguments. Internally these are called partial key sequences. The OS4 module provides various support for prompting functions built on top of this mechanism.

One of several nice features of the prompting system on the HP-41 is that the calculator goes to sleep between key presses to reduce power consumption.

17.1 State

Two bytes called `ptemp1` and `ptemp2` are used to keep track of the state during partial key sequences. While the partial key sequence is active they are kept in either the `ST` (flag) and `G` registers.

When not processing key presses, `ptemp2` is preserved in a status register (in chip 0) and `ptemp1` is rebuilt each time as it contains information about what kind of key was just pressed. The address of the code that will handle the next key press is kept on the top of the processor stack. Combined, these can be seen as a continuation, or put another way, the information context needed to process the next key press is saved for future use.

17.2 Display prompt

Functions that prompt for input have bits set in the function name header to tell it is a prompting function. This also tells the code that is decoding which function to run that it first needs to prompt for an argument before it can actually run. At this point, the function code is saved in a status register and the upper bits are stored in the `ptemp2` state byte. The value of these bits informs what class of prompting function it belongs to. Among other things it controls the number of initial underscores.

The function name is written to the LCD followed by a blank and prompt underscores are added by a call to a `NEXT` routine where its numeric suffix tells now many underscores to add, e.g. `NEXT2` adds two underscores. The `NEXT` routine will not return until a key is pressed and the calculator goes to sleep. A partial key sequence flag is also set to indicate that the calculator is processing a prompt. The return address to the caller of `NEXT` is kept on the processor stack while the calculator is in light sleep waiting for the next key press.

Before giving control back, all underscores are removed from the display then it returns to the address following the call to NEXT if back-arrow key was pressed, or to the following address if some other key was pressed. This makes it easy to decode back-arrow key and ptemp1 is set up in the ST flag register, making it easy to dispatch on different classes of keys.

If you accept the input you append what was keyed to the LCD, which can be one or more characters, e.g. the top rows keys often act as 01–10 when pressed. If further input is needed you call the appropriate NEXT routine to output the number of underscores desired.

17.3 Rolling your own

For certain functions it is desirable to use the prompt mechanism, but there are several problems as with prompting XROM functions which requires some care. The operating system is quite permissive in allowing prompting XROMs, but there are limitations on what actually works in practice.

17.3.1 Known bugs and limitations

There are two main classes of prompting functions, those that take alpha arguments and those that take numeric arguments. HP utilized alpha input for XROM functions but they are not programmable. Numeric input with XROMs was not used by HP, but is also possible. Unfortunately, there are some bugs in the code related to this, which will be discussed below.

PRP (print program) in the printer is an example of an XROM that prompts for alpha input. As such function cannot be represented in a program it is non-programmable.

Note: Later this way of doing alpha input to XROM functions used the alpha register. This was used for file names in the HP-IL mass storage and Extended Functions/Memory modules. As they take input from the alpha register they can be made programmable. They are also more flexible as they allowed for dual input, e.g. specifying a program but storing it in a file under a different name.

Prompting XROM functions work up to a point. There is no built-in support for representing them in a program so they need to be non-programmable. In the previous chapter we looked at semi-merged functions which is provided by OS4 and it is a way to work around this limitation. There are also some bugs and unexpected behavior related to numeric prompting XROM functions:

1. If the function is allowed to prompt for a stack register, the XROM function being built becomes corrupted.
2. The printer will also get confused and print the wrong function postfix.

Note: The semi-merged postfix mechanism contains code that works around the problem with entered stack registers corrupting the instruction.

17.4 Custom prompting

Custom prompting can be used if you want full control of the prompting behavior. This only works for non-programmable functions. Typical use are functions such as XEQ and ASN, where replacements can provide additional capabilities using custom prompting.

Marking the name header of a function makes it possible to choose among the built-in standard prompt behaviors. If you want to design the prompt behavior on your own you need to override it. OS4 makes this

possible using a special marker that needs to appear as the first instruction in the function (after any leading NOP instructions that mark it as non-programmable and possibly also XKD).

```
              .con    '\'' + 0x80   ; '
              .con    0x0e          ; N
              .con    0x100 + 19    ; S
              .con    0x101         ; A
myASN:        nop                   ; non-programmable
              gosub   partialKey    ; marker partial key takeover
              goto    assign        ; when executed, argument is done and we will
                                    ;   perform the actual assignment
              goto    abortASN      ; <-
              ...                   ; normal processing

assign:                             ; actual run behavior after prompt done

abortASN:     golong XABTSEQ
```

The call to partialKey marks the function as a partial key sequence takeover function. The number of underscores in the initial prompt is determined by the bits in the name header. An ordinary partial key sequence function is started and the return address for the next key processing is initially set to inside mainframe (which depends on the prompt bits in the name header). OS4 detects that there is a call to partialKey as first real instruction in the function that is prompting and alters the return address that points somewhere in mainframe so that it instead is the return address of the call to partialKey, plus one.

When the first key is pressed in response to the prompt, it is handled by the code following the call to partialKey plus one. The normal back-arrow handler is first, then processing for other keys, which is how the NEXT routines work in mainframe.

The purpose of the name field bits is only to put up the initial prompt. The purpose of the GOSUB partialKey marker is to tell OS4 that the function wants to do its own processing. OS4 alters the return address kept on the stack for the next key processing to point to your own key handler code.

When the prompt has been fully filled in you should jump to one of the null test entry points, e.g. NULT_, NULT_3 or NULT_4 to do null testing and if key is released in time execute the function.

Execution is done the normal way by actually running the function. As the first instruction is a call to partialKey it will get executed. Previously it has only acted as a marker for redirecting (overriding) the prompt handler. Executing it will do nothing as it immediately returns to the next line (the one the prompt handler skipped over before). It should be a short jump to the actual code that performs the function.

What about the collected prompt input data? Normally, alpha input is in the Q register and a numeric operand is in A.X. If you want something else you need to store it somewhere before you called the null test handler code. As the Q registers is available for prompt arguments, it can be a good choice.

Note: Make a jump to XABTSEQ to abort partial key processing. This works almost identical to ABTSEQ in mainframe which has the same purpose, except that XABTSEQ performs some additional clean-ups for the purpose of OS4.

Note: The Boost module uses this to provide replacements for XEQ and ASN. However, you are not limited to improving existing functionality, you can provide something completely new.

17.4.1 Design considerations

Some extension modules (like CCD) show prompt underscores immediately for more than one field, e.g. the two arguments of an XROM. This may be seen as user friendly, but existing base functionality like ASN do not present up front that it will also prompt for a key once you entered the function name. In addition, the key prompt is a single underscore, even though the actual key pressed will be presented as a two digit number.

Thus, you are rather free to do whatever you want and there is nothing unusual with taking a field at a time and gradually prompt for the next thing, even if you know that you eventually will prompt for additional things following a known pattern. On the other hand, making it more elaborate may make it easier for the user to understand it. The take-away is that both ways have been in used for long and are accepted, there is nothing right or wrong.

17.4.2 Limitations

If you decide to use the prompt mechanism offered by mainframe with your XROM function it is possible to do so, but normal limitations apply. The bug with stack operands exists (unless you use a corrected mainframe ROM). The function must be non-programmable and your function cannot be a secondary functions. The CAT' catalog replacement function in the Boost module is currently implemented this way.

If you take over prompt handling but in the end make use of existing functionality in mainframe it will not work properly with secondary functions. One example of this is the ASN' assign replacement function in the Boost module. It uses the mainframe code to ask for the key code as its final step. After this it tries to execute the function from mainframe and it cannot execute secondary functions. This forces it to be an ordinary XROM function.

17.4.3 Support functions

Some prompt support functionality can be found in the Boost module. The parseNumber routine can be used for requesting decimal numbers. This can prompt for a given number of digits and has an accept predicate, making it possible to check the input to be in a specific range, e.g. 0–511 or 1–31. Impossible input is detected as soon as possible, resulting in refused input and a display blink.

```
gosub   parseNumber
.con    .low12 accept_1_31
.con    2              ; request 2 digits
```

To allow the EEX key to be used to extend the range you need to include its mask value:

```
gosub   parseNumber
.con    .low12 accept_1_31
.con    2 | ParseNumber_AllowEEX ; request 2 digits, allow EEX
```

CHAPTER 18

Interval timer

The interval timer in the Time module can be borrowed by applications to provide timeouts or recurring timed events, e.g. a clock.

It is primarily intended for transient applications, e.g. waiting for a fixed amount of time for a key press. Another use is for animating the display, e.g. to show a blinking cursor. It can also be used to display periodic updates of some external hardware readouts, like GPS coordinates or other external measurements.

18.1 Starting

Use `setTimeout` to set the timeout which is a 5 digit BCD number. The interval timer starts and you will get periodic notifications with the specified time interval.

18.2 Stopping

Once you have called `setTimeout` it is your responsibility to also call `clearTimeout` to stop the timer.

If you only want a single timeout, you need to call `clearTimeout` in your timeout handler which is pointed to from the shell descriptor.

If you have a transient application you also also need to stop the interval timer when the shell exits. The normal keyboard definition has a field for this, see see *Key handlers*. If you are doing your own custom key handling, simply call it from your key handler.

18.3 Timeout

For an application this is an entry in the shell descriptor. As mentioned, you need to stop the timer when you decide you do not want any further timeouts with `clearTimeout`.

18.3.1 Partial key sequence

It is also possible to use a timeout when doing partial key sequence parsing. A timeout is notified using the backarrow vector. You can distinguish between a backarrow key press and a timeout by looking at the sign field of the A register. This field is set to zero if the entry was called due to a backarrow key press and is non-zero if there was a timeout:

```
entry:        gosub   NEXT2
              goto    cancel
              gosub   clearTimeout  ; normal key press, give timer back
              ...                   ; handle key

cancel:       ?a#0    s             ; timeout?
              gonc    abort         ; no
              ...                   ; timeout specific handling
abort:        gosub   clearTimeout  ; give timer back
              golong  XABTSEQ
```

If you only want a timeout that cancels the key sequence parser you do not need look for whether it was a timeout. However, you should call `clearTimeout` in both cases.

CHAPTER 19

Entry points reference

19.1 API version check

19.1.1 checkApiVersionOS4

Entry point: 4F80

```
;;; ****************************************************************************
;;;
;;; checkApiVersionOS4 - check the version expected by OS4
;;;
;;; In: C.X = API version number, where the first nibble is the main
;;;           version number that must match. The lower 8 bits are the
;;;           minor number that be at least the same
;;; Out: Only returns if the current version is at least the expected
;;;      one. Exits showing error "OLD OS4" if expected version is
;;;      higher than the current one.
;;; Uses: A.X, C[6:3]
;;;
;;; ****************************************************************************
```

19.2 Fundamentals

Basic building blocks for dealing with packed pointers and other small routines.

19.2.1 jump via packed pointer

This is a set of routines that fetch a packed pointer by offset displacement from a given base address and pass control to it. By using the GOSUB instruction to one of these routines you perform a table indirect call to a packed pointer address. You can also use the GOLONG instruction to just transfer control to it.

```
;;; ***************************************************************************
;;;
;;; jumpC0 - jump via a packed pointer
;;; jumpC1 - jump via a packed pointer, increment by 1
;;;   ...
;;; jumpC7 - jump via a packed pointer, increment by 7
;;; jumpPacked - C[6] has page and C.X holds packed pointer
;;;
;;; Assume that C[6:3] points somewhere in a table of code pointers, well
;;; there should at least be one. Provide different entry points to advance
;;; the pointer to allow for invoking different routines by offsetting a
;;; base pointer.
;;; The location to jump to is assumed to be in the same 4K page as the
;;; table of pointers. The actual pointer is stored packed, that is,
;;; a 10-bit pointer aligned on 4. The value 000 is used to indicate that
;;; the pointer is not defined.
;;; As usual, you typically use gosub or golong to invoke the jumpCN
;;; routine, based on whether you are calling or jumping to the active
;;; pointer. Some care is of course needed in case you use golong and
;;; the stored routine does not exist (has value 000), in that case it
;;; will still return and that will be to what is on the stack. This may
;;; of course be intentional, if it is a tail-jump to exit a subroutine.
;;;
;;; In C[6:3] - base pointer for jump/call table
;;; Uses: C[12:0] and more depending on the routine that is invoked
;;;
;;; Note: This routine is intentionally first in the fixedEntries section.
;;;       Should these be need to add more entries, it can be done easily
;;;       by just doing it and adjusting back the start address of
;;;       fixedEntries accordingly.
;;;
;;; ***************************************************************************
```

19.2.2 call aligned subroutine

These are alternatives to the three word page relative jump and call routines provided by mainframe. They require 4 alignment on the destination address, but avoids the problem of temporary using a subroutine level and can access the full 4K page.

```
;;; ***************************************************************************
;;;
;;; gosubAlign4 - call a 4 aligned subroutine in a plug-in ROM
;;; golAlign4   - golong a 4 aligned subroutine in a plug-in ROM
;;;
;;; These routines are similar to the various GOSUB0-GOSUB3, and GOL0-GOL3
;;; except that they run faster and require that the called routine is
;;; aligned on an even 4 address (the lowest two address bits of the
;;; destination must be 0).
;;;
;;; The main benefit of these routines is that they do not use an additional
;;; subroutine level while allowing calls to go anywhere within a 4K page.
;;; In other words, they allow arbitrary calls within a 4K page utilizing
;;; the full 4 level return stack.
;;;
;;; Also consider using GOLONG/GOSUB in mainframe that is basically as
;;; fast but limits the range to be within same 1K page inside the 4K
```

(continues on next page)

(continued from previous page)

```
;;; page.
;;; Another alternative is using GSB000/GSB256/GSB512/GSB768 in mainframe
;;; which aligns on 256 and all calls must take place from within the
;;; following 256 words. While being far less flexible, it is even faster.
;;; Though it only provides gosub, not golong.
;;;
;;; Uses: C
;;; Assumes: hex enabled
;;;
;;; ************************************************************************
```

19.2.3 return skipping ahead

These routines allow for returning to (P+2) and (P+3), that is, skipping one or two instructions ahead in the return location.

```
;;; ************************************************************************
;;;
;;; RTNP2 - return to P+2
;;; dropRTNP2 - drop stack and return to P+2
;;;
;;; These routines are useful for returning skipping past the instruction
;;; just after the gosub.
;;;
;;; dropRTNP2 is meant to be used with generic extensions that wants to
;;;  return back to the original caller (instead of exiting back to
;;;  mainframe). It simply drops the return address (which points back to
;;;  extensionHandler) and returns to (P+2) of the original generic
;;;  extension caller.
;;;
;;; Uses: C[6:3]
;;;
;;; ************************************************************************
```

```
;;; ************************************************************************
;;;
;;; RTNP3 - return to P+3
;;; dropRTNP3 - drop stack and return to P+3
;;;
;;; These routines are useful for returning skipping past the instruction
;;; just after the gosub.
;;;
;;; dropRTNP3 is meant to be used with generic extensions that wants to
;;;  return back to the original caller (instead of exiting back to
;;;  mainframe). It simply drops the return address (which points back to
;;;  extensionHandler) and returns to (P+3) of the original generic
;;;  extension caller.
;;;
;;; Uses: C[6:3]
;;;
;;; ************************************************************************
```

19.2.4 unpack a packed pointer

These routines read a packed pointer by offset displacement from a given base address and returns the resulting address.

```
;;; ************************************************************************
;;;
;;; unpack - unpack a packed pointer in C[6:3]
;;; unpack1 - unpack a packed pointer in C[6:3] + 1
;;;  ...
;;; unpack7 - unpack a packed pointer in C[6:3] + 7
;;;
;;; Change a base pointer to what a packed field pointer points to.
;;; Basically base.member in C.
;;;
;;; In: C[6:3] - pointer to a structure
;;; Out: C[6:3] - the member at given offset, unpacked
;;;
;;; ************************************************************************
```

19.3 Return to mainframe

19.3.1 XFNRC

Entry point: 4D30

```
;;; ************************************************************************
;;;
;;; XNFRC - do alternative shell display and return to mainframe
;;;
;;; Use this routine to exit back to mainframe, showing the default X
;;; for the active application.
;;; This exits to NFRC, which means the stack lift flag is not affected.
;;;
;;; ************************************************************************
```

19.3.2 XFNRPU

Entry point: 4D34

```
;;; ************************************************************************
;;;
;;; XNFRPU - do alternative shell display and return to mainframe
;;;
;;; Use this routine to exit back to mainframe, showing the default X
;;; for the active application.
;;; This exits to NFRPU, which enables stack lift.
;;;
;;; ************************************************************************
```

19.3.3 resetMyBank

Entry point: 4F90

```
;;; ************************************************************************
;;;
;;; resetMyBank - reset to primary bank (routine in bank 1)
;;;
;;; Call this routine to reset to primary bank. This takes the address
;;; from the return stack and reset using a bank switch routine in that
;;; page.
;;;
;;; In: page address on stack
;;; Out: Primary bank selected, PT=2
;;; Uses: C[6:3], active PT
;;;
;;; ************************************************************************
```

19.3.4 SKP_resetMyBank

Entry point: 4D59

```
;;; ************************************************************************
;;;
;;; SKP_resetMyBank - reset callers bank and then exit to SKP
;;;
;;; This is the conditional exit routine that skips next RPN program
;;; line, or in run-mode prints NO depending on S7.
;;;
;;; The alternative entry point SKP_YESNO_resetMyBank resets S7 to make
;;; it print NO in run-mode.
;;;
;;; ************************************************************************
```

19.3.5 NOSKP_resetMyBank

Entry point: 4D5F

```
;;; ************************************************************************
;;;
;;; NOSKP_resetMyBank - reset callers bank and then exit to NOSKP
;;;
;;; This is the conditional exit routine that exectes next RPN program
;;; line, or in run-mode prints YES depending on S7.
;;;
;;; The alternative entry point NOSKP_YESNO_resetMyBank resets S7 to make
;;; it print YES in run-mode.
;;;
;;; ************************************************************************
```

19.4 General utilities

19.4.1 XBCDBIN

Entry point: 4F94

```
;;; ***************************************************************************
;;;
;;; XBCDBIN - convert small BCD number to binary
;;;
;;; The built-in BCDBIN cannot handle numbers larger than 999, this
;;; routine can handle a range of 0-4095.
;;; Originally by Ken Emery / Skwid, reference PPCCJ V11N5P6
;;;
;;; In: C= floating point number
;;; Out: C.X= binary number
;;; Uses: A, C, +1 sub level
;;; Note: may exit to ERRAD or ERRNE
;;;
;;; ***************************************************************************
```

19.4.2 CXtoX

Entry point: 4F98

```
;;; ***************************************************************************
;;;
;;; CXtoX - convert small binary number to floating point in X
;;;
;;; The final value is recalled to X, hiwch
;;;
;;; In: C[2:0] - binary number
;;; Out: X - floating point number
;;;
;;; ***************************************************************************
```

19.4.3 CtoXRcl

Entry point: 4F9a

```
;;; ***************************************************************************
;;;
;;; CtoXRcl - binary integer to floating point number, use RCL
;;;
;;; The converted binary number is saved in X. The 3 different main
;;; entry points correspond to push value (CtoX), update X after unary
;;; operation (CtoXFill) and update X after binary operation (CtoXDrop).
;;; The two latter also update L.
;;;
;;; In: C - binary integer (all bits)
;;; Out: X - floating point number
;;;
;;; ***************************************************************************
```

19.4.4 CtoXDrop

Entry point: 4F9c

```
;;; ***************************************************************************
;;;
;;; CtoXDrop - binary integer to floating point number, use DROPST
;;;
;;; The converted binary number is saved in X. The 3 different main
;;; entry points correspond to push value (CtoX), update X after unary
;;; operation (CtoXFill) and update X after binary operation (CtoXDrop).
;;; The two latter also update L.
;;;
;;; In: C - binary integer (all bits)
;;; Out: X - floating point number
;;;
;;; ***************************************************************************
```

19.4.5 CtoXFill

Entry point: 4F9e

```
;;; ***************************************************************************
;;;
;;; CtoXFill - binary integer to floating point number, use FILLXL
;;;
;;; The converted binary number is saved in X. The 3 different main
;;; entry points correspond to push value (CtoX), update X after unary
;;; operation (CtoXFill) and update X after binary operation (CtoXDrop).
;;; The two latter also update L.
;;;
;;; In: C - binary integer (all bits)
;;; Out: X - floating point number
;;;
;;; ***************************************************************************
```

19.4.6 pausingReset

Entry point: 4F96

```
;;; ***************************************************************************
;;;
;;; pausingReset - test and reset OS4 pausing flag
;;;
;;; In: Nothing
;;; Out: Returns to (P+1) if pause is running
;;;      Returns to (P+2) if not OS4 pause in progress
;;; Uses: A, C, B.X, active PT=12, +1 sub level
;;;
;;; ***************************************************************************
```

19.4.7 getIndexX

Entry point: 4FA4

19.5 Error handling

These error routines are the same as found in the Extended Functions module and later 41CX. They are provided in OS4 as they do not exist in 41C and 41CV. Thus, if you rely on OS4 they are now available on all HP-41 variants.

19.5.1 displayError

Entry point: 4F82

```
;;; ************************************************************************
;;;
;;; displayError - display error message followed by ERR
;;;
;;; A call to this routine should be followed by the error message
;;; formatted in the same way as with the MESSL call.
;;;
;;; ************************************************************************
```

19.5.2 errorMessage

Entry point: 4F2A

```
;;; ************************************************************************
;;;
;;; errorMessage - display an error message
;;;
;;; A call to this routine should be followed by the error message
;;; formatted in the same way as with the MESSL call.
;;;
;;; ************************************************************************
```

19.5.3 errorExit

Entry point: 4F2C

```
;;; ************************************************************************
;;;
;;; errorExit - left justify LCD and handle error
;;;
;;; ************************************************************************
```

19.5.4 noRoom

Entry point: 4F28

```
;;; ************************************************************************
;;;
;;; noRoom - show NO ROOM error
;;;
;;; ************************************************************************
```

19.5.5 noSysBuf

Entry point: 4F4A

```
;;; ****************************************************************************
;;;
;;; noSysBuf - show NO SYSBUF error
;;;
;;; ****************************************************************************
```

19.5.6 ERRDE_resetMyBank

Entry point: 4D38

```
;;; ****************************************************************************
;;;
;;; ERRDE_resetMyBank - reset the bank of the caller and exit to ERRDE
;;;
;;; Use this routine to exit back to mainframe, giving a DATA ERROR
;;; message.
;;;
;;; Calling sequence:
;;;      <some test>
;;;      gsubc   ERRDE_resetMyBank
;;;
;;; ****************************************************************************
```

19.5.7 ERRNE_resetMyBank

Entry point: 4D3D

```
;;; ****************************************************************************
;;;
;;; ERRNE_resetMyBank - reset the bank of the caller and exit to ERRNE
;;;
;;; Use this routine to exit back to mainframe, giving a NONEXISTENT
;;; message.
;;;
;;; Calling sequence:
;;;      <some test>
;;;      gsubc   ERRNE_resetMyBank
;;;
;;; ****************************************************************************
```

19.5.8 ERRAD_resetMyBank

Entry point: 4D42

```
;;; ****************************************************************************
;;;
;;; ERRAD_resetMyBank - reset the bank of the caller and exit to ERRAD
;;;
;;; Use this routine to exit back to mainframe, giving a ALPHA DATA
```

(continues on next page)

(continued from previous page)

```
;;; message.
;;;
;;; Calling sequence:
;;;      <some test>
;;;      gsubc   ERRAD_resetMyBank
;;;
;;; ***************************************************************************
```

19.5.9 ERROF_resetMyBank

Entry point: 4D47

```
;;; ***************************************************************************
;;;
;;; ERROF_resetMyBank - reset the bank of the caller and exit to ERROF
;;;
;;; Use this routine to exit back to mainframe, giving a OUT OF RANGE
;;; message.
;;;
;;; Calling sequence:
;;;      <some test>
;;;      gsubc   ERROF_resetMyBank
;;;
;;; ***************************************************************************
```

19.5.10 errorExit_resetMyBank

Entry point: 4D4C

```
;;; ***************************************************************************
;;;
;;; errorExit_resetMyBank - reset the bank of the caller and exit to errorExit
;;;
;;; Use this routine to exit back to mainframe, finalizing a custom error
;;; message.
;;;
;;; Calling sequence:
;;;      <some test>
;;;      gsubc   ERROF_resetMyBank
;;;
;;; ***************************************************************************
```

19.5.11 CHK_NO_S_resetMyBank

Entry point: 4D51

```
;;; ***************************************************************************
;;;
;;; CHK_NO_S_resetMyBank - test for alpha data
;;;
;;; This is the CHK_NO_S routine in a variant that will reset the bank for
;;; caller if it exits to display ALPHA DATA
```

(continues on next page)

(continued from previous page)

```
;;;
;;; In C - floating point number
;;; Out: C - floating point number
;;;      decimal mode set
;;;
;;; ************************************************************************
```

19.6 Ensure environment

Some code may require certain optional hardware to be available for correct operation. These routines allow for testing such requirements and give a sensible error if the resource is not present.

19.6.1 ensureDrive

Check for the mass storage HP-IL drive. This tests both that we have an HP-IL module as well as some mass storage device connected to the HP-IL loop.

Entry point: `4F68`

```
;;; ************************************************************************
;;;
;;; ensureDrive - ensure that an HP-IL moduld and mass storage exists
;;;
;;; This routine only returns if the HP-IL module is present, and there
;;; is a mass storage drive.
;;;
;;; Uses: C, A, PT, +3 sub levels
;;;
;;; ************************************************************************
```

19.6.2 ensureHPIL

Entry point: `4F44`

```
;;; ************************************************************************
;;;
;;; ensureHPIL - ensure that an HP-IL module is inserted
;;;
;;; This routine only returns if the HP-IL module is present.
;;;
;;; Uses: C, A.X, PT, +0 sub levels
;;;
;;; ************************************************************************
```

19.6.3 ensure41CX

Testing for 41CX is intended to ensure that the entry points for accessing extended memory is available.

Entry point: `4F46`

```
;;; ***************************************************************************
;;;
;;; ensure41CX - ensure we are running on a 41CX style OS
;;;
;;; This routine only returns if we are running on a HP-41CX (or similar style)
;;; operating system.
;;;
;;; Uses: C, A.X
;;;
;;; ***************************************************************************
```

19.6.4 ensureTimer

The timeout related routines will politely tell you if there is no timer available. This routine is intended if you want to really bail out and prevent further actions when the timer is missing.

Entry point: 4F82

```
;;; ***************************************************************************
;;;
;;; ensureTimer - ensure there is a timer chip
;;;
;;; Display NO TIMER if there is no timer chip present.
;;;
;;; Uses: C, A.X
;;;
;;; ***************************************************************************
```

19.7 Data entry

Entry point: 4F2E

```
;;; ***************************************************************************
;;;
;;; clearSystemDataEntry - reset the system data entry flag
;;;
;;; Uses: A, B.X, C, enables chip 0
;;;
;;; ***************************************************************************
```

Entry point: 4F22

```
;;; ***************************************************************************
;;;
;;; fastDataEntry  - fast additional data entry handling
;;;
;;; Routine to handle additional data entry when a data entry routine finds
;;; that there is another key down. The idea here is to bypass the I/O poll
;;; vector to speed things up.
;;;
;;; ***************************************************************************
```

19.8 Assignments

Functions related to assignments are mainly for handling assignments of secondary functions which reside in the system buffer.

19.8.1 assignSecondary

Entry point: 4F54

```
;;; ************************************************************************
;;;
;;; assignSecondary - assign a secondary function
;;;
;;; In: A[1:0] - keycode, in 1-80 form
;;;     B[4:3] - XROM identity
;;;     B[2:0] - secondary function number
;;;
;;; Uses: A, C, B.X, N, M, DADD, +3 sub levels
;;;
;;; ************************************************************************
```

Note: If there is insufficient free space this function will cause a NO ROOM error exit. In this case any previous assignment made to the key is lost.

19.8.2 clearAssignment

Entry point: 4F52

This routine will remove an assignment to a given key code no matter if it is a primary or secondary assignment (or both).

```
;;; ************************************************************************
;;;
;;; clearAssignment - delete an assignment
;;;
;;; In: A[1:0] - keycode to be cleared, in 1-80 form
;;; Out: Nothing
;;; Uses: A, C, B.X, N, M, DADD, +3 sub levels
;;;
;;; ************************************************************************
```

19.8.3 clearSecondaryAssignments

Entry point: 4F5E

```
;;; ************************************************************************
;;;
;;; clearSecondaryAssignments - clear all secondary assignments
;;;
;;; In: Nothing
;;; Out: Nothing
```

(continues on next page)

(continued from previous page)

```
;;; Uses: A, C, B, G, PT, DADD, +2 sub levels
;;;
;;; ************************************************************************
```

19.8.4 findSecondaryAssignments

Entry point: 4F0C

```
;;; ************************************************************************
;;;
;;; findSecondaryAssignments - locate the first secondary KAR
;;;
;;; This routine is typically be used by routines that want to access all
;;; secondary key assignment registers.
;;;
;;; In: Nothing
;;; Out: Returns to P+1 if no secondary KARs
;;;      Returns to P+2 if secondary KARs exists with:
;;;      A.X - address of first secondary KAR
;;;      A.S - number of secondary KARs - 1
;;; Uses: A, C, B.X, active PT=12, DADD, +1 sub levels
;;;
;;; ************************************************************************
```

19.8.5 mapAssignments

This routine is called RSTKCA by HP. The variant presented here is based on the one found in the Extended Functions module, enhanced with the ability to choose either assignments from global program label assignments or the key assignment registers when both are active. This is mentioned in the comments in the Extended Functions module, but that logic is actually not present in the code. The original comes from the Card Reader which has this ability. Apparently, the code was slightly stripped when moved but the comment was left as-is.

Here it is presented in all its glory with the addition that it is now also handles secondary assignments and will rebuild those key assignments map too. If a secondary assignment happens to be shadowed by a primary assignment, as can be the result when loading primary assignments using functions that are unaware of secondary assignments, the secondary assignment is cleared.

Entry point: 4FA0

```
;;; ************************************************************************
;;;
;;; mapAssignments - rebuild the key reassignment bit maps
;;;
;;; This is useful is key assignments are load from some external media
;;; and as a way to rebuild the key assignment maps if they have been
;;; disabled for the local auto assign labels.
;;;
;;; The procedure is as following:
;;;  1. Clear the bitmap
;;;  2. Restore the key reassignments of mainframe functions & XROM functions
;;;  3. Restore key reassignments in alpha labels; if the key has already
;;;     been assigned to another function:
;;;    A. If it is after reading in a status track, clear the key code
```

(continues on next page)

(continued from previous page)

```
;;;       in the alpha label.
;;;    B. If it is after reading in a program, find the key code somewhere
;;;       else and clear it there.
;;;  4. Clear secondary bitmaps.
;;;  5. Restore secondary key assignments.
;;;
;;; In: S2=1 means we do 3A, status has been read in and we obey it.
;;;     S2=0 means 3B, user program assignments takes precedence
;;; Out: Nothing
;;; Uses: A, C, B, N, M, PT, S7, DADD, +3 sub levels
;;;
;;; ***************************************************************************
```

19.9 Buffers

Buffer routines cover both ordinary I/O buffers, as well as hosted buffers which are stored inside the system buffer.

19.9.1 findBuffer

Entry point: `4F06`

This routine is used to locate a buffer.

```
;;; ***************************************************************************
;;;
;;; findBuffer - locate buffer with ID in C.X
;;;
;;; If not found, return to (P+1)
;;; If found, return to (P+2) with:
;;;   A.X = address of buffer start register
;;;   DADD = first address of buffer
;;;   C[13] part of buffer header incremented
;;;   C[12:7] = part of buffer header
;;;   C[2:0] = part of buffer header
;;; Uses: A[12], A.X, C, B.X, active PT=12, DADD, +0 sub levels
;;;
;;; Note: For findBuffer, buffer number in C[0] and C[2:1] must be zero!!!
;;;       Use 'ldi' or 'c=0 x' to ensure that.
;;;
;;; This routine is called at every light sleep wake up, so it has to
;;; be fast.
;;;
;;; Typical execution time (here) is:
;;;
;;;   14 + KARs * 11 + otherBufs * 16 + 16 if found
;;;
;;; So for 8 assignments and skipping 2 buffers, it would take 106 cycles
;;; or 16ms on a standard HP-41.
;;;
;;; ***************************************************************************
```

Note: It is assumed that the buffer number has been loaded with the `LDI` instruction, which causes the upper

two nibbles (C[2:1] of the C.X field) to be zero.

19.9.2 ensureBuffer

Entry point: 4F08

```
;;; ****************************************************************************
;;;
;;; ensureBuffer - find or create an empty buffer
;;;
;;; Like findBuffer, but will create the buffer with only a header if it
;;; does not previously exist.
;;;
;;; In: C.X - buffer ID
;;; If not existing and cannot be created (not free space), return to (P+1)
;;; If found, return to (P+2) with:
;;;   A.X = address of buffer start register
;;;   B.X = address of chain head (permanent .END.)
;;;   DADD = first address of buffer
;;;   C[13] part of buffer header incremented
;;;   C[12:7] = part of buffer header
;;;   C[2:0] = part of buffer header
;;; Uses: A[12], A.X, C, B.X, PT, DADD, +1 sub levels
;;;
;;; ****************************************************************************
```

19.9.3 ensureBufferWithTrailer

Entry point: 4FA2

```
;;; ****************************************************************************
;;;
;;; ensureBufferWithTrailer - find or create an empty buffer
;;;
;;; Like ensureBuffer, but will create a buffer with a trailer register if
;;; it does not previously exist. The trailer register is set to
;;; 10000000000000.
;;;
;;; In: C.X - buffer ID
;;; If not existing and cannot be created (not free space), return to (P+1)
;;; If found, return to (P+2) with:
;;;   A.X = address of buffer start register
;;;   B.X = address of chain head (permanent .END.)
;;;   DADD = first address of buffer
;;; Uses: A[12], A.X, C, B.X, PT, DADD, +1 sub levels
;;;
;;; ****************************************************************************
```

19.9.4 reclaimSystemBuffer

Entry point: 4F30

```
;;; ***************************************************************************
;;;
;;; reclaimSystemBuffer - ensure the system buffers stays at power on
;;;
;;; This entry is intended for modules that uses the functionality
;;; provided by the system buffer, but does not use any shell.
;;; In that case, call this routine from the deep wake-up polling
;;; point to reclaim the system buffer.
;;;
;;; Uses: C, +1 sub level, DADD
;;;
;;; ***************************************************************************
```

19.9.5 growBuffer

Entry point: 4F0A

```
;;; ***************************************************************************
;;;
;;; growBuffer - add space to buffer
;;;
;;; Open up some register space inside a buffer. Newly allocated registers
;;; are filled with F000000000000 for two reasons. First to ensure that
;;; if we add space at the top of a buffer, it has to be non-zero, which
;;; relieves the "burden" from the caller. Second, it helps with debugging
;;; as new registers have a known static value rather than garbage that may
;;; look like active data in the buffer.
;;;
;;; In: A.X= buffer header address
;;;     C.X= offset where to add registers
;;;     G= number of registers to add
;;; Out: Returns to P+1 if it is not possible to grow the buffer
;;;      Returns to P+2 if successful with
;;;        B.X= the location of the newly added space
;;;        A.X= buffer header address
;;;        DADD= buffer header address
;;; Uses: A, B, C, G, DADD, S7, active PT, +1 sub levels
;;;
;;; ***************************************************************************
```

19.9.6 shrinkBuffer

Entry point: 4F38

```
;;; ***************************************************************************
;;;
;;; shrinkBuffer - remove registers from a buffer
;;;
;;; Remove a given number of registers from a buffer (at a given offset).
;;; There are no error checking here, you better know that you have a
;;; buffer that can drop this number of registers. Garbage or incorrect
;;; input will forsure corrupt the memory!
;;;
;;; In: A.X= buffer header address
```

(continues on next page)

(continued from previous page)

```
;;;     C.X= offset of first register to remove
;;;     G= number of registers to remove
;;; Out: A.X= buffer header address
;;;      DADD= buffer header address
;;; Uses: A, B[12:0], C, M, G, DADD, active PT set to 0, +1 sub levels
;;;
;;; ***************************************************************************
```

19.9.7 allocScratch

Entry point: 4F3A

```
;;; ***************************************************************************
;;;
;;; allocScratch - allocate (ensure) a given scratch size
;;;
;;; Allocate the scratch register area meant for transient applications.
;;; If the area is currently allocated it is assumed to be a left-over
;;; from a previous scratch register allocation and we will free that
;;; one first if not of the requested size.
;;;
;;; In: C[0] = size of scratch area (1-15)
;;; Out: Returns to P+1 if it is not possible to grow the buffer
;;;      Returns to P+2 if successful
;;;      B.X= the location of the newly added space
;;;      A.X= buffer header address
;;;      DADD= buffer header address
;;; Uses: A, B, C, G, S7, DADD, active PT set to 10, +2 sub levels
;;;
;;; ***************************************************************************
```

19.9.8 scratchArea

Entry point: 4F3E

```
;;; ***************************************************************************
;;;
;;; scratchArea - get pointer to transient application scratch area
;;;
;;; This routine assumes that the scratch area exists!!!
;;;
;;; In: Nothing
;;; Out: C.X - pointer to scratch area
;;;      A.X - address of buffer header
;;;      C[3] - size of scratch area (0-15 registers)
;;;      DADD - first scratch register selected
;;; Uses: A[12], A.X, C, B.X, active PT=12, DADD, +1 sub level
;;;
;;; ***************************************************************************
```

19.9.9 clearScratch

Entry point: 4F3C

```
;;; ***************************************************************************
;;;
;;; clearScratch - remove transient application scratch area
;;;
;;; A transient application can set up a transient scratch area typically
;;; to store its state. This works because there can only be one active
;;; transient application and making one active means
;;;
;;; In: Nothing
;;; Out: A.X= buffer header address
;;;      DADD= buffer header address
;;; Uses: A, B[12:0], C, M, G, DADD, active PT set to 0, +1 sub levels
;;;
;;; ***************************************************************************
```

19.9.10 newHostedBuffer

Entry point: 4F6E

```
;;; ***************************************************************************
;;;
;;; newHostedBuffer - reserve space for a hosted buffer.
;;;
;;; Typical use is to call findBufferHosted to find a specific buffer.
;;; If it is not there, it can be created using createBufHosted.
;;; !! NOTE: This routine assumes that the hosted buffer does not exist !!
;;;
;;; In: C[1:0] = size of buffer
;;;     N[1:0] = buffer number 0-127
;;;
;;; If out of memory, returns to (P+1)
;;; If successful, returns to (P+2) with:
;;;      A.X = hosted buffer header address
;;;      DADD = hosted buffer header
;;; Uses: A, B, C, N, G, DADD, S7, active PT, +2 sub levels
;;;
;;; ***************************************************************************
```

19.9.11 findBufferHosted

Entry point: 4F6A

```
;;; ***************************************************************************
;;;
;;; findBufferHosted - find a hosted buffer
;;;
;;; Locate a secondary buffer.
;;;
;;; In: C[1:0]= buffer number
;;; If not found, return to (P+1)
;;; If found, return to (P+2) with:
;;;   A.X= hosted buffer header address (selected)
;;;   B[5:3] = system buffer header address
;;; Uses: A, B, C, G, active PT, +1 sub level
```

(continues on next page)

(continued from previous page)

```
;;;
;;; ****************************************************************************
```

19.9.12 reclaimHostedBuffer

Entry point: 4F6C

```
;;; ****************************************************************************
;;;
;;; reclaimHostedBuffer
;;;
;;; Reclaim a hosted buffer, typically called at power on by modules that
;;; want to retain a hosted buffer.
;;;
;;; In: C.X - buffer number bitwise ORed with 0x80
;;; Out: Nothing
;;; Uses: A, C, B.X, N, active PT=12, +2 sub levels
;;;
;;; ****************************************************************************
```

19.9.13 packHostedBuffers

Entry point: 4F74

```
;;; ****************************************************************************
;;;
;;; packHostedBuffers - pack the hosted buffer area
;;;
;;; Remove all buffers marked for removal. Marking for removal is by
;;; setting the highest bit in register to one.
;;;
;;; In: Nothing
;;; Out: Nothing
;;; Uses: A, B[12:0], C, M, G, DADD, PT, +2 sub levels
;;;
;;; ****************************************************************************
```

19.9.14 growHostedBuffer

Entry point: 4F70

```
;;; ****************************************************************************
;;;
;;; growHostedBuffer - add space to a hosted buffer
;;;
;;; This routine expects inputs as after a call to findBufferHosted.
;;; In part this is because both findBufferHosted and growBuffer (which
;;; is the work-horse) both takes input in G. It is not for sure if it is
;;; even desirable to combine the routines, the user may need to consult
;;; the buffer before deciding to grow it and decide on where.
;;; As usual, we check for that there is space and the system buffer
;;; does not grow too large, but there is no sanity checking done on
```

(continues on next page)

(continued from previous page)

```
;;; the actual input. You are expected to pass valid pointers and
;;; offsets.
;;;
;;; In: A.X= hosted buffer header address
;;;     B[5:3] = system buffer header address
;;;     C.X= offset where to add registers
;;;     G= number of registers to add
;;; Out: Returns to P+1 if it is not possible to grow the buffer
;;;      Returns to P+2 if successful with
;;;        B.X= the location of the newly added space
;;;        A.X= hosted buffer header address
;;;        DADD= hosted buffer header address
;;; Uses: A, B, C, G, N, DADD, S7, active PT, +2 sub levels
;;;
;;; ************************************************************************
```

19.9.15 shrinkHostedBuffer

Entry point: 4F72

```
;;; ************************************************************************
;;;
;;; shrinkHostedBuffer - remove space from a hosted buffer
;;;
;;; In: A.X= hosted buffer header address
;;;     B[5:3] = system buffer header address
;;;     C.X= offset where to add registers
;;;     G= number of registers to add
;;; Out: Returns to P+1 if it is not possible to grow the buffer
;;;      Returns to P+2 if successful with
;;;        A.X= hosted buffer header address
;;;        C= hosted buffer header
;;;        DADD= hosted buffer header address
;;; Uses: A, B, C, G, M, N, DADD, S7, active PT, +2 sub levels
;;;
;;; ************************************************************************
```

19.10 Shells

19.10.1 activateShell

Entry point: 4F00

```
;;; ************************************************************************
;;;
;;; activateShell - activate a given Shell
;;;
;;; In: C.X - packed pointer to shell structure
;;;     !! Must be called from same page as shell structure !!
;;; Out: Returns to (P+1) if not enough free memory
;;;      Returns to (P+2) on success
;;; Uses: A, B, C, M, G, ST, active PT, +3 sub levels
```

(continues on next page)

(continued from previous page)

```
;;;
;;; ************************************************************************
```

19.10.2 exitShell

Entry point: 4F02

```
;;; ************************************************************************
;;;
;;; exitShell - dectivate a given Shell
;;;
;;; exitShell marks a given Shell as an unused slot, essentially removing it.
;;; We do not reclaim any memory here, it is assumed that it may be a
;;; good idea to keep one or two empty slots around. Reclaiming any
;;; buffer memory is a different mechanism.
;;; Transient application will have any scratch area removed.
;;;
;;; In: C.X - packed pointer to shell structure
;;; Out:
;;; Uses: A, B.X, C, M, ST, S8, DADD, active PT,
;;;       +1 sub level if reclaimShell
;;;       +3 sub levels if exitShell (due to exit via shellChanged)
;;;
;;; ************************************************************************
```

19.10.3 reclaimShell

Entry point: 4F04

```
;;; ************************************************************************
;;;
;;; reclaimShell - reclaim a Shell at power on
;;;
;;; reclaimShell marks a shell to activate it.
;;;
;;; In: C.X - packed pointer to shell structure
;;; Out:
;;; Uses: A, B.X, C, M, ST, S8, DADD, active PT,
;;;       +1 sub level if reclaimShell
;;;       +3 sub levels if exitShell (due to exit via shellChanged)
;;;
;;; ************************************************************************
```

19.10.4 exitTransientApp

Entry point: 4F40

```
;;; ************************************************************************
;;;
;;; exitTransientApp - exit the top level transient application
;;;
;;; Exit any top level transient application and its transient area.
```

(continues on next page)

(continued from previous page)

```
;;;
;;; In: Nothing
;;; Out: Nothing
;;; Uses: A, B.X, C, M, ST, active PT, DADD, +3 sub levels
;;;
;;; ************************************************************************
```

19.10.5 exitApp

Entry point: 4F78

```
;;; ************************************************************************
;;;
;;; exitApp - exit the top level application
;;;
;;; Exit the top application.
;;;
;;; In: Nothing
;;; Out: Nothing
;;; Uses: A[12], A.X, C, B.X, active PT, DADD, +2 sub levels
;;;
;;; ************************************************************************
```

19.10.6 hasActiveTransientApp

Entry point: 4F42

```
;;; ************************************************************************
;;;
;;; hasActiveTransientApp - is there an active top level transient application?
;;;
;;; In: Nothing
;;; Out: Returns to (P+1) if no active transient application
;;;      Returns to (P+2) if there is an active transient application
;;; Uses: A[12], A.X, C, B.X, active PT, DADD, +1 sub levels
;;;
;;; ************************************************************************
```

19.10.7 activeApp

Entry point: 4F36

```
;;; ************************************************************************
;;;
;;; activeApp - return pointer to active application
;;;
;;; In: Nothing
;;; Out: Returns to (P+1) if no active application
;;;      Returns to (P+2) if there is an active application with
;;;        A[6:3]= unpacked pointer to application
;;;        PT= 6
;;; Uses: A, B.X, C, active PT, DADD, +2 sub levels
```

(continues on next page)

(continued from previous page)

```
;;;
;;; ************************************************************************
```

19.10.8 topShell

Entry point: `4F14`

```
;;; ************************************************************************
;;;
;;; topExtension - find first extension point
;;; topShell - find first shell
;;; nextShell - find next shell (or extension point)
;;;
;;; topShell can be used to locate first active shell.
;;; The following active shells can be found by successive calls to
;;; nextShell.
;;;
;;; In:  Nothing
;;; Out: Returns to (P+1) if no buffer
;;;      Returns to (P+2) if no shells (active)
;;;          B.X - buffer address
;;;      Returns to (P+3) with
;;;          A[6:3] - pointer to shell
;;;          M - shell scan state
;;;          ST= system buffer flags, Header[1:0]
;;;          B.X= address of system buffer
;;;          S9= application seen
;;;          PT= 6
;;;          DADD= register where shell descriptor is
;;; Uses: A, B.X, C, M, DADD, S8, S9, active PT, +2 sub levels
;;;
;;; ************************************************************************
```

19.10.9 nextShell

Entry point: `4F18`

This routine can be used to find the next successive shell after starting with a call to `topShell`. You must preserve the `M` register while making successive calls.

19.10.10 topExtension

Entry point: `4F16`

This is similar to `topShell` but searches for extension points in the shell stack.

19.10.11 shellDisplay

Entry point: `4F10`

```
;;; ************************************************************************
;;;
;;; shellDisplay - show active shell display and set message flags
;;;
;;; This routine is meant to be called when a shell aware module wants
;;; to show the X register before returning to mainframe. We will look
;;; at the active application shell, do its display routine if mode is
;;; appropriate and set message flag to avoid having the normal show X
;;; routine update display, only to have it overwritten soon after.
;;; After calling this routine, jump back to a suitable NFR* routine
;;; which probably is NFRC.
;;;
;;; In: Nothing, do not care about DADD or PFAD
;;; Out: Nothing
;;; Uses: Worst case everything, +3 sub levels
;;;
;;; ************************************************************************
```

19.10.12 displayDone

Entry point: 4F32

```
;;; ************************************************************************
;;;
;;; displayDone - set flags indicating display is done
;;;
;;; You normally do not need to call this routine. It is meant to be used
;;; in certain cases when you have done the display early and do not
;;; want to have default display update the normal way.
;;; One situation where this is useful is for a command which purpose is
;;; to show an alternative display, like showing the 'X' value in an
;;; alternative way to default.
;;;
;;; In: Nothing
;;; Out: chip 0 selected
;;; Uses: A[12], A.X, C, B.X, active PT=12, DADD, +1 sub levels
;;;
;;; ************************************************************************
```

19.10.13 displayingMessage

Entry point: 4F56

```
;;; ************************************************************************
;;;
;;; displayingMessage - test if showing a message
;;;
;;; This routine tests if the display is currently showing a message.
;;; Normally you would test the message flag in the flag register for this,
;;; but it may also be set by a shell to tell that the display is done
;;; and the system default of showing X should not be done.
;;; This poses a problem if you really want to know if a message is being
;;; shown, typically when implementing an alternative backarrow logic,
;;; where you want to distinguish between clearing the display or clearing
```

(continues on next page)

(continued from previous page)

```
;;; the X register. In this case this routine is handy.
;;;
;;; Note: This will not report if a message is shown while running a
;;;       program! The reason is that this routine is intended for backspace
;;;       logic. In a running program you normally know if you are
;;;       displaying a message or not.
;;;       In a running program it suffices to inspect the ordinary message
;;;       flag.
;;;
;;; In: Nothing
;;; Out: Returns to (P+1) if showing message
;;;      Returns to (P+2) if normal display
;;; Uses: A[12], A.X, C, B.X, ST, active PT=12, DADD, +1 sub levels
;;;
;;; ************************************************************************
```

19.10.14 sendMessage

Entry point: 4F34

```
;;; ************************************************************************
;;;
;;; sendMessage - invoke an extension
;;; shellChanged - the shell stack was changed
;;;
;;; In:  C[1:0] - generic extension code
;;; Out:   Depends on extension behavior and if there is an active one.
;;;        If there is no matching generic extension, returns to the
;;;        caller.
;;;        If there is a matching generic extension, it decides on what to
;;;        do next and is extension defined.
;;;        Typical behavior include one of the following:
;;;        1. Return to sendMessage using a normal 'rtn'. This is
;;;           typical if it is some kind of notification or broadcast.
;;;           In this case the shell stack is further searched for more
;;;           matching generic extensions that will also get the chance
;;;           to be called.
;;;        2. As a single handler that bypasses further matches by returning
;;;           to the orignal caller. This can be done using:
;;;             spopnd
;;;             rtn
;;;           Which takes us back to the original caller. It is not possible
;;;           for it to tell whether the call was handled by a generic
;;;           extension, unless some told by the return value, for example
;;;           using the N register that is not used by sendMessage.
;;;           Another alternative is to return to (P+2) if the call was
;;;           handled (unhandled calls always return to (P+1)), this can
;;;           be done using:
;;;             golong dropRTNP2
;;;        Argument/accumulator:
;;;        You can pass information in for example N register to the
;;;        handler(s). Handler may update that information or whatever
;;;        is appropriate/useful. This is basically a protocol between
;;;        the original caller and the handlers, and is completely up to
;;;        the extension to define the protocol.
```

(continues on next page)

(continued from previous page)

```
;;; Note: An extension that returns to sendMessage must preserve
;;;       M, S9 and B.X and not leave PFAD active.
;;; Uses: A, B.X, C, M, ST, DADD, active PT, +3 sub levels
;;;
;;; ***************************************************************************
```

19.10.15 shellName

Entry point: 4F1A

```
;;; ***************************************************************************
;;;
;;; shellName - append the name of the current shell to LCD
;;;
;;; Using the shell scan state, shift in the name of the active shell
;;; from the right into the LCD.
;;; This works the same way as MESSL, but the string comes from the shell.
;;;
;;; In: C[6:3] - pointer to shell
;;; Out: LCD selected
;;; Uses: A.M, C, +1 sub level
;;;
;;; ***************************************************************************
```

19.10.16 shellKeyboard

Entry point: 4F4C

```
;;; ***************************************************************************
;;;
;;; shellKeyboard - get active keyboard
;;;
;;; Advance pointer to the field that holds the active keyboard handler.
;;;
;;; In: A[6:3] - pointer to shell
;;; Out: C[6:3] - pointer to active keyboard handler entry
;;; Uses: A[13:3], C, ST, DADD, +1 sub levels
;;;
;;; ***************************************************************************
```

19.11 Semi-merged

These functions define semi-merged prompting functions.

Note: A call to these routines are expected to appear first in a function. The OS4 code inspects the start of potential such functions and expects it to look in a particular way.

19.11.1 argument

Entry point: 4F1E

```
;;; ***************************************************************************
;;;
;;; argument - handle numerical arguments for functions in XROMs
;;;
;;; Start MCODE function as follows:
;;; XADR  nop
;;;       nop
;;;       gosub   argument
;;;       .con    DefaultOperand + modifiers
;;;
;;; IN: SS0 UP, CHIP0 selected
;;; OUT:  ST - numeric argument
;;;       C[1:0] - numeric argument
;;;       C[13:0] - 0
;;;       G - numeric argument
;;;
;;; Possible modifiers are:
;;; SEMI_MERGED_NO_STACK - do not allow direct stack addressing
;;;
;;; ***************************************************************************
```

19.11.2 dualArgument

Entry point: 4F76

```
;;; ***************************************************************************
;;;
;;; dualArgument - request two arguments
;;;
;;; Start MCODE function as follows:
;;; XADR  nop
;;;       nop
;;;       gosub   dualArgument
;;;       .con    modifiers
;;;
;;; IN: SS0 UP, CHIP0 selected
;;; OUT:  A[3:2] - first argument
;;;       A[1:0] - second argument
;;;
;;; Possible modifiers are:
;;; SEMI_MERGED_NO_STACK - do not allow direct stack addressing, first
;;;                        operand
;;; SEMI_MERGED_SECOND_NO_STACK - do not allow direct stack addressing,
;;;                               second operand
;;; SEMI_MERGED_QMARK - append a question mark after this function
;;;                     when shown in program mode
;;;
;;; ***************************************************************************
```

19.11.3 postfix4095

Entry point: 4F92

```
;;; ****************************************************************************
;;;
;;; postfix4095 - convert postfix operand to a value 0-4095
;;;
;;; This support routine takes a postfix operand and converts it to a
;;; number in the range 0-4095 (12 bits, or exponent field). For a direct
;;; argument the range is limited 0-127. Indirect arguments are needed for
;;; the full range.
;;;
;;; In: ST= postfix operand
;;; Out: C.X= numeric value of the operand
;;; Uses: A, B, C, M, N, +3 sub levels
;;; Note: may exit to ERRAD or ERRNE
;;;
;;; ****************************************************************************
```

19.12 Partial key sequences

19.12.1 XABTSEQ

Entry point: 4F5C

```
;;; ****************************************************************************
;;;
;;; XABTSEQ - abort partial key sequence
;;;
;;; This function terminates partial key sequence, typically when pressing
;;; backarrow to get out of it.
;;; Compared to the ABTSEQ routine in mainframe, this will also clear alpha
;;; mode and resets flags in the system buffer header related to key
;;; sequence parsing and secondary functions that may have partial key
;;; sequence handling (it does not harm to do it for all partial key
;;; handling).
;;;
;;; ****************************************************************************
```

19.13 Secondary functions

19.13.1 XASRCH

This is a variant of the ASRCH routine in mainframe. This routine works the same but will also locate secondary functions.

Entry point: 4F4E

```
;;; ****************************************************************************
;;;
;;;  XASRCH - alpha search
;;;
;;; Locate the address of an alpha string. The alpha string may apply to
;;; an alpha label in RAM or a function in the mainframe or plug-in ROMs.
;;; If the function is located in a plug-in ROM, return the XROM function
```

(continues on next page)

(continued from previous page)

```
;;; code. This function is similar to ASRCH in mainframe, but secondary
;;; FATs are also searched. If the function is located in the mainframe,
;;; return its function code. If the function is located in RAM, return
;;; the alpha label address.
;;
;;; In:  M[13:0] and REG 9[13:0] = alpha label (2 COPIES)
;;;
;;; Out: S6=0 implies primary instruction found:
;;;      C[3:0]=   address (if user lang, this is address of first
;;;                         byte of label)
;;;      C[7:4]= function code
;;;      S2=1/0 implies ROM/RAM address
;;;      C=0 implies not found
;;;      S9=1/0 implies microcode/user code
;;;      S5=1 implies a mainframe function
;;;      chip 0 enabled
;;;
;;;      S6=1 implies a secondary XROM was found:
;;;      N[6:3]= points to secondary FAT header
;;;      B[6:3]= points to the XROM page address (X000)
;;;      A[2:0]= sequence number of secondary
;;;      Note: Active bank is primary bank, which may not be the same
;;;            bank as the secondary.
;;;
;;; Uses: M, A, B, C, G, N, STATUS, ptr P, REG 9,
;;;       status bits 2,3,5,6,8,9
;;;       +2 sub levels
;;;
;;; ************************************************************************
```

19.13.2 resetBank

Entry point: 4F58

```
;;; ************************************************************************
;;;
;;; resetBank    - reset to primary bank (routine in bank 1)
;;;
;;; Call the XFC7 entry in given bank to reset to bank 1. This routine is "safe"
;;; in that it is guarded by looking at the bank bit presence in the
;;; ROM identifier.
;;; The bankswitcher is assumed to be an enromX instruction followed
;;; by RTN, if not the 'Uses' statement here may not apply!
;;; This routine is in bank 1, the resetBank_B2 is a duplicate version
;;; in bank 2.
;;;
;;; In: C[6]= page
;;; Out: Primary bank selected, PT=2
;;; Uses: C[6:3], active PT
;;;
;;; ************************************************************************
```

19.13.3 secondaryAddress

Entry point: 4F50

```
;;; ***************************************************************************
;;;
;;; secondaryAddress - look up a secondary function
;;;
;;; In: C[6]= page address
;;;     A.X= secondary function identity
;;; Out: If function does not exist:
;;;        A.M= 0
;;;      If function exists:
;;;        A[6:3]= address of secondary function
;;;        A[10:7]= address of bank switch routine
;;;        A.M= non-zero ([6:3] is the address which is non-zero if valid)
;;;        A.X= secondary function identity
;;;        active bank set for secondary
;;; Uses: A, B[12:0], C, active PT
;;;       +1 sub levels, or +2 sub levels called from outside OS4
;;;
;;; ***************************************************************************
```

19.13.4 runSecondary

Entry point: 4F60

```
;;; ***************************************************************************
;;;
;;; runSecondary - run-time handling of secondary invocation
;;;
;;; In: STK= called from ROM page, @STK is prefix function number
;;;
;;; Out: does not return to caller
;;; Uses: N/A
;;;
;;; ***************************************************************************
```

19.13.5 invokeSecondary

Entry point: 4F5A

```
;;; ***************************************************************************
;;;
;;; invokeSecondary - call a secondary function
;;;
;;; Invoke a secondary function, that is, execute it or store in a program
;;; as appropriate. This function does not return, control is given back
;;; to operating system or program to execute next command.
;;; If the given secondary function is not currently in the given module
;;; page, appropriate action is taken (shows XXROM identity number, and
;;; give NONEXISTENT error unless NULLed).
;;;
;;; In: C[6]= page address
;;;     A.X - secondary function identity
```

(continues on next page)

(continued from previous page)

```
;;;
;;; ****************************************************************************
```

19.14 Keyboard

19.14.1 keyKeyboard

Entry point: 4F5A

```
;;; ****************************************************************************
;;;
;;; keyKeyboard - act on a key using the given keyboard definition
;;;
;;; Handle a key press according to the given keyboard. This will resolve
;;; auto assignment, assignments, decode and execute instructions according
;;; to a keyboard table. Additionally, it allows for custom data entry
;;; and termination.
;;;
;;; Invoke this routine using:
;;;          gosub keyKeyboard
;;;          .con ...       ; keyboard descriptor
;;; The 'gosub' is to get the page address to be coupled with the lower
;;; 12 bits. This function will not return to that caller, instead it is
;;; assumed that the real return address is located in the previous slot
;;; on the stack (typically goes back to 'core' to try another shell
;;; when the key is not handled. If we do handle the key, we also drop
;;; that return address from the stack.
;;;
;;; In: KY - key register holds the key
;;;     Chip 0 selected
;;;     A.S - 0 if user mode (set in keyHandler)
;;;
;;; ****************************************************************************
```

19.14.2 keyDispatch

Entry point: 4F66

```
;;; ************************************************************
;;;
;;; keyDispatch - table dispatch on key (from TIME module)
;;;
;;; Read the keycode and jump to the corresponding handler using
;;; a jump table.  This is essentially KEY-FC in the TIME module.
;;;
;;; IN: Key down, C[2:0] holds table length minus 1
;;;     Last entry in table must be 000 to mark end of table.
;;;
;;; OUT: C.X= 0 (to make it easy to increment for data entry keys)
;;;
;;; USED: A, C
;;;
```

(continues on next page)

(continued from previous page)

```
;;; NOTE: The key is down when entering. Any wait for key release
;;;       and debounce handing is the reposibility of the caller.
;;;
;;; ************************************************************
```

19.14.3 assignKeycode

Entry point: 4F8E

```
;;; ************************************************************************
;;;
;;; assignKeycode - convert key code to the way it is shown by assign
;;;
;;;       Code taken from Extended Functions module.
;;; In: C.X - key code as returned by C=KEYS
;;; Out: C - key code as floating point number
;;; Uses: A, C, PT, +0 sub levels
;;;
;;; ************************************************************************
```

19.15 Timer

OS4 allows for using the interval timer which can be useful for timeouts or periodic updates of the display.

Only the active application will receive timeout events. The timeout handler is in the shell descriptor (at offset 6).

19.15.1 setTimeout

Entry point: 4F62

```
;;; ************************************************************************
;;;
;;; setTimeout - set a time out
;;;
;;; This borrows the interval timer in the Time module which is normally
;;; used for the clock display. Here we allow it to be used as a timeout
;;; intended for transient shells.
;;; The idea is that it can be used for some query for input and timeout
;;; if nothing was keyed in a given time or for animation, i.e. a blinking
;;; cursor or field.
;;;
;;; In: C[4:0] - timeout in hundreds of seconds, BCD coded
;;; Out: Returns to (P+1) if the timer does not exist
;;;      Returns to (P+2) if the timer exists and was initialized,
;;;          timer is now armed for a timeout.
;;;          DADD - system buffer header
;;;
;;; Uses: A, C, B, ST, DADD, PFAD, +1 sub levels
;;;
;;; ************************************************************************
```

19.15.2 clearTimeout

Entry point: 4F64

```
;;; ***************************************************************************
;;;
;;; clearTimeout - disable the time out
;;;
;;; Uses: C.X, DADD, PFAD, +1 sub levels
;;;
;;; ***************************************************************************
```

19.16 Catalog

The catalog functionality provides a framework that factors out most of the structure around running a catalog.

19.16.1 catalog

Entry point: 4F7C catalog

Entry point: 4F7E catalogWithSize

```
;;; ***************************************************************************
;;;
;;; catalog - generic catalog support
;;;
;;; Assume a single register state that is held in N while running.
;;; The state is saved in the Q register while stopped.
;;;
;;; Alternative entry point:
;;; catalogWithSize - same as catalog, but useful when the state needs
;;;   more than one register. Put the needed size in C.X which will
;;;   allocate the scratch area of this size. The N register is still
;;;   a single state register while running and it is saved in the
;;;   Q register. The scratch area is allocated up front and will
;;;   result in NO ROOM if not enough registers are available.
;;;
;;; The calling sequence is:
;;;           ldi     .low12 catalogDescriptor
;;;           gosub   catalog
;;;           ...
;;;           .align  4
;;;   catalogDescriptor:
;;;           goto    .low12 prepare
;;;           goto    .low12 step
;;;           goto    .low12 back
;;;           .con    .low12 transientShell
;;;           bankSwitcher code
;;;
;;; prepare - Show first line, N=state, return to (P+2).
;;;           If catalog is empty, return to (P+1)
;;; step - Step to next entry, show line, N=updated state, return to (P+2)
;;;           If nothing more, return to (P+1)
;;;
```

(continues on next page)

(continued from previous page)

```
;;; In: Nothing
;;; Out: Nothing (returns to mainframe when done)
;;;
;;; ************************************************************************
```

19.16.2 catalogEnd

Entry point: 4F88

```
;;; ************************************************************************
;;;
;;; catalogEnd - terminate the catalog
;;;
;;; This routine terminates the catalog by removing the transient
;;; application and doing usual catalog termination in mainframe.
;;; Does not return.
;;;
;;; The calling sequence is:
;;;           gosub catalogEnd
;;;
;;; ************************************************************************
```

19.16.3 catalogReturn

Entry point: 4F8C

```
;;; ************************************************************************
;;;
;;; catalogReturn - enter catalog with an entry to display
;;;
;;; This routine should be called stopped and some catalog specific
;;; action has been done, which resulted in setting a display and we
;;; want to dispatch on the next key.
;;;
;;; The calling sequence is:
;;;           gosub catalogDisplay
;;;
;;; ************************************************************************
```

19.16.4 catalogStep

Entry point: 4F84

```
;;; ************************************************************************
;;;
;;; catalogStep - handle catalog step
;;;
;;; This routine should be called from the transient application handling
;;; key input for the catalog. It steps to the next entry or ends the
;;; catalog if stepping outside it.
;;;
;;; The calling sequence is:
```

(continues on next page)

(continued from previous page)

```
;;;           ldi   .low12 catalogDescriptor
;;;           gosub catalogStep
;;;
;;; ************************************************************************
```

19.16.5 catalogBack

Entry point: `4F86`

```
;;; ************************************************************************
;;;
;;; catalogBack - handle catalog step back
;;;
;;; This routine should be called from the transient application handling
;;; key input for the catalog. It steps to the next entry or ends the
;;; catalog if stepping outside it.
;;;
;;; The calling sequence is:
;;;           ldi   .low12 catalogDescriptor
;;;           gosub catalogBack
;;;
;;; ************************************************************************
```

19.16.6 catalogRun

Entry point: `4F8A`

```
;;; ************************************************************************
;;;
;;; catalogRun - start running the catalog again
;;;
;;; This routine should be called when the R/S key is pressed while
;;; catalog is stopped. It startd running the catalog from the current
;;; position.
;;;
;;; The calling sequence is:
;;;           ldi   .low12 catalogDescriptor
;;;           gosub catalogRun
;;;
;;; ************************************************************************
```

19.17 Extended memory

19.17.1 getXAdr

If your function takes a postfix argument you will probably want to use `postfix4095` together with `getXAdr`, see *postfix4095*.

Entry point: `4F12`

```
;;; ***************************************************************************
;;;
;;; getXAdr - get the X memory address of given logical register
;;;
;;; Convert a register number to a physical address based on the active
;;; data file. If running, we try to speed it up using the cache in
;;; register 13. The first call during execution will set up the
;;; cache. If not running, we simply ignore the status of the cache
;;; and do the real lookup. This adds safety in case the some operation
;;; have invalidated the cache. When running we assume the program knows
;;; what it is doing and invalidates the cache explicitly whenever
;;; needed (after an operation that may shuffle around registers in the
;;; file system).
;;;
;;; In: C.X= logical register
;;; Out: A.X = physical register
;;; Uses: A, B, C, M, N, Q, PT, S0-7, DADD, +3 sub levels
;;;
;;; ***************************************************************************
```

CHAPTER 20

Glossary

Application

A new environment or mode to make the calculator behave differently.

Auto assignment

The assignments that are automatically bound to the upper two key rows. Corresponds to single letter local alpha labels in the current RPN program.

Execution token

The byte code(s) that defines a function.

Extension handler

A message system provided by OS4. Has some resemblance to poll vectors, but can be defined by application modules.

FAT

Function address table. Contains pointers to functions. Each entry uses two words.

Hosted buffer

A buffer that is contained inside the system buffer.

ID area

A four word sequence at the end of a module that contains a short identification of the module. Some bits are used to mark if the module is banked and if it has secondary functions.

Mainframe

The name HP used internally for the HP-41 firmware. (Normally it means a very powerful computer.)

MCODE

The machine language used on the HP-41. HP originally called it microcode, which is misleading as it usually means something else. The term M-Code was coined in the user community (where the M could refer to either), and later it became MCODE.

Primary bank

Bank 1 in a banked module.

Primary function

An ordinary XROM function or a built in function.

Packed pointer

A single word (10-bit value) that can refer to a location in a 4K module page. Address alignment (on 4) and a page address value from some outer context is used to construct a full 16-bit address from it.

Poll vectors

A fixed set of locations near the end of a module that are called (if defined) during certain key system events.

RPN

Reverse Polish Notation, also called postfix notation. Arguments are specified before the operation. In this manual also used as the name for user code program language of the HP-41. Some prefer to call it FOCAL.

Secondary bank

Any bank that is not the primary. This is bank 2, 3 or 4 in a banked module.

Secondary FAT

A function address table that goes together with a secondary FAT header. This table may be in any bank.

Secondary FAT header

A structure that defines a sequence of secondary functions. Multiple secondary FAT headers can exist as a linked list. Must be in the primary bank.

Secondary function

A function that is member of a secondary FAT.

Semi-merged

A program step that is actually two. The first is an XROM function and the second is a text literal that gives extra information to the previous step. The first step is displayed fully decorated based on the extra information in the text literal. The full decoration is the semi part.

System buffer

The buffer OS4 uses to store its state. Uses buffer identity 15.

System shell

Typically used for add-on keyboard modifications that can update keyboard behavior and be active at all time.

Transient application

A temporary mode like a catalog or updating clock display.

XADR

The first execution address in a function.

XKD

Execute direction function. If bound or assigned to a key executes on key press down.

XXROM function

A secondary function. The XXROM is similar to XROM and is displayed when it belongs to a module that is not plugged in.

Index

Non-alphabetical

A

B

C

D

E

F

G

H

U

V

X

Y