Calypsi tools guide for the Nut

Release 5.18

Håkan Thörngren

Jul 15, 2026

Contents

1 Introduction 1

2 Installation 3
- 2.1 macOS . . . 3
- 2.2 Debian Linux . . . 4
- 2.3 Arch and Manjaro Linux . . . 4
- 2.4 Windows . . . 5
- 2.5 Using multiple versions . . . 5

3 Programming the HP-41 7
- 3.1 Using this guide . . . 7
- 3.2 Further reading . . . 7
- 3.3 Document conventions . . . 8

4 Source files and file extensions 9
- 4.1 File extensions . . . 9

5 Nut assembler 11
- 5.1 Source file format . . . 12
- 5.2 Preprocessor . . . 12
- 5.3 Sections . . . 13
- 5.4 Expressions . . . 13
- 5.5 Location counter . . . 14
- 5.6 Local labels . . . 14
- 5.7 Defining constants . . . 15
- 5.8 Constants that are locations . . . 16
- 5.9 Mainframe entry points . . . 16
- 5.10 Directives . . . 16
- 5.11 List files . . . 22
- 5.12 Macro language . . . 25
- 5.13 Object file format . . . 27
- 5.14 Relocations . . . 27

6 RPN compiler 31
- 6.1 RPN language . . . 32
- 6.2 Preprocessor . . . 32
- 6.3 A basic example . . . 34

6.4 Branches and labels . . . 35
6.5 Global alpha labels . . . 36
6.6 Using other plug-in modules . . . 37
6.7 Invoking local MCODE instructions . . . 38
6.8 Converting `.raw` files . . . 39
6.9 Merged XROM instructions . . . 40

7 Module tool 43
7.1 Extraction mode . . . 44
7.2 Creation mode . . . 45
7.3 Module description file . . . 45
7.4 Module exports file . . . 46
7.5 Extracting ROM pages . . . 46
7.6 Technical details . . . 46

8 Linker 47
8.1 Memory . . . 48
8.2 Linker rules file . . . 49
8.3 Linker list files . . . 51
8.4 Module files . . . 54
8.5 More on linker rules . . . 54

9 Librarian 55
9.1 Creating a library . . . 55

10 NEWT 57
10.1 Speed annotation . . . 57
10.2 Module files . . . 58

11 Bank switching 59
11.1 Hardware behavior . . . 59
11.2 Single page . . . 60
11.3 Exiting back . . . 62
11.4 Calling between banks . . . 63

12 Barcode generator 65
12.1 Language . . . 65
12.2 Preprocessor . . . 66
12.3 Branches and labels . . . 66

13 Clonix tool 67
13.1 Differences . . . 68
13.2 Load process . . . 69
13.3 Building the image . . . 70
13.4 Programming a module . . . 70
13.5 Setting up a Makefile . . . 70

14 Other useful tools 71
14.1 Editor . . . 71
14.2 Build engine . . . 72
14.3 Source code control . . . 72

15 Instruction sets 73
15.1 Non-local jumps . . . 73
15.2 Local branches . . . 74

15.3 Other jump and subroutine instructions . . . 74
15.4 Page relocatable jumps . . . 74
15.5 Pointer instructions . . . 75
15.6 Arithmetic instructions . . . 75
15.7 Flag instructions . . . 76
15.8 Data memory . . . 76
15.9 Keyboard . . . 76
15.10 Miscellaneous . . . 77

16 HP instruction set 79
16.1 Non-local jump instructions . . . 79
16.2 Local branches . . . 80
16.3 Page relocatable jumps . . . 80
16.4 Other jump and subroutine instructions . . . 81
16.5 Pointer instructions . . . 81
16.6 Arithmetic instructions . . . 81
16.7 Flag instructions . . . 83
16.8 RAM memory instructions . . . 83
16.9 ROM memory instructions . . . 84
16.10 Keyboard instructions . . . 84
16.11 Miscellaneous instructions . . . 85
16.12 LCD instructions . . . 85
16.13 Timer chip instructions . . . 87
16.14 Card reader instructions . . . 87
16.15 Wand instructions . . . 88
16.16 HP-IL instructions . . . 88
16.17 Peripheral instructions . . . 88
16.18 Hepax instructions . . . 88
16.19 NEWT specific instructions . . . 89

17 Jacobs-DeArras instruction set 91
17.1 Non-local jump instructions . . . 91
17.2 Local branches . . . 91
17.3 Page relocatable jumps . . . 92
17.4 Other jump and subroutine instructions . . . 92
17.5 Pointer instructions . . . 92
17.6 Arithmetic instructions . . . 93
17.7 Flag instructions . . . 94
17.8 RAM memory instructions . . . 94
17.9 ROM memory instructions . . . 95
17.10 Keyboard instructions . . . 95
17.11 Miscellaneous instructions . . . 96
17.12 Hepax instructions . . . 96
17.13 NEWT specific instructions . . . 96
17.14 Peripheral instructions . . . 97

18 4K page layout 99
18.1 FAT order . . . 100

19 Character sets 103
19.1 HP-41 character set . . . 103
19.2 HP-41 display characters . . . 104

20 Examples 107
20.1 Hello world . . . 107

20.2 RPN games module 108
20.3 Modules on the internet 110

Index 113

CHAPTER 1

Introduction

Despite being almost 40 years old, the classic HP-41 calculator is still being used and loved. While the actual user base today is small, some people still enjoy the fun of this, way ahead of its time, amazing calculator.

The high level language of the HP-41 was named FOCAL, but that was an unfortunate choice as that name was already used by another language. The name RPN is used here as it is often used as a family name for languages of this kind. RPN also goes well with the naming of the language used by the successors of the HP-41, which is called RPL. RPL is however quite different from RPN.

The native assembly language of the HP-41 is usually called MCODE and it gives full control over the calculator. Due to the way memory system on the HP-41 is constructed, MCODE programs can only be executed from plug-in modules.

Hewlett-Packard and third parties produced many plug-in modules back in the 1980s. The majority were mask programmed by HP and typically required large volumes to make sense, but there were also EPROMs used for low volume modules. A small cottage industry existed at the time. Today modules are developed as software images that can be loaded into module simulators (re-programmable hardware) which offer higher capacity than ever before.

Examples of such re-programmable custom hardware are the Clonix and NoV series of modules. If you are the lucky owner of an HP-41CL, it can hold all HP-41 module images known to mankind inside the calculator!

If you use an emulator, it will most likely provide support for loading module images.

The Calypsi tool chain aims to provide command line tools that makes it easy to develop your own module images for the HP-41, both using MCODE and RPN. In addition to this, there are tools to work with module images and Clonix images. There is also a tool to generate barcodes to allow RPN programs to be loaded into the RAM based program memory of the calculator. You will need a postscript printer and an HP 82153A Wand to use this barcode feature.

CHAPTER 2

Installation

This section contains installation information for supported platforms.

2.1 macOS

The macOS installer places the tools under `/usr/local`. Simply double-click the installation package to install the software.

After installation, files are in `/usr/local/calypsi-!targetLowercase-`, where "" is the installed version. Each the Calypsi tool chain release creates a new directory, preventing overwrites of previous installations.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` environment variable to use the most recent installed version of the Calypsi tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-!targetLowercase` to the installation directory, making it easy to refer to the installation without a version number.

2.1.1 Setting the PATH

To set the `PATH` environment variable on macOS, edit `/etc/paths` and add `/usr/local/bin`.

2.1.2 Uninstalling

To uninstall the Calypsi tool chain, run the uninstall script:

```
$ sudo /usr/local/lib/calypsi-!targetLowercase-/uninstall
```

This will remove the installation and the links in `/usr/local/bin`.

2.2 Debian Linux

The Debian package (`.deb` extension) is built and tested on Ubuntu 20.04. Installation on other Debian-based systems may be possible, but remains untested.

The package can be installed from the command line using:

```
$ sudo gdebi calypsi-nut-.deb
```

Note: Download the Debian package to a local file before installation. `gdebi` does not fetch packages from a server.

If you do not have `gdebi` installed already, install it using:

```
$ sudo apt install gdebi-core
```

This installs the Calypsi tool chain in `/usr/local/lib/calypsi-!targetLowercase-`, where "" is the installed version number.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` to use the latest the Calypsi tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-!targetLowercase` to the installation directory. This allows referring to the installation without a version number.

It is also possible to install using `dpkg` in the following way:

```
$ sudo dpkg -i calypsi-!targetLowercase-.deb
```

This may fail if package dependencies are missing. The output will advise on necessary steps before retrying installation.

After installing any requested dependencies, retry the sudo dpkg installation command.

2.2.1 Uninstalling

To uninstall the Calypsi tool chain, use `dpkg` with the `-r` flag:

```
$ sudo dpkg -r calypsi-!targetLowercase
```

2.3 Arch and Manjaro Linux

Manjaro is based on Arch Linux and uses `pacman`. Install the package file from the command line using:

```
$ sudo pacman -U calypsi-!targetLowercase--x86_64.pkg.tar.zst
```

Note: Download the package file to your local file system before installation. `pacman -U` does not fetch packages from a server.

This installs the Calypsi tool chain in `/usr/local/lib/calypsi-!targetLowercase-`, where "" is the installed version number.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` to use the latest the Calypsi tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-!targetLowercase` to the installation directory. This allows referring to the installation without a version number.

2.3.1 Uninstalling

To uninstall the Calypsi tool chain, use `pacman` with the `-R` flag:

```
$ sudo pacman -R calypsi-!targetLowercase
```

2.4 Windows

Extract the Windows product `calypsi-!targetLowercase-.zip` to a suitable location. It will expand into a folder `calypsi-!targetLowercase-`, allowing easy management of multiple versions.

2.4.1 Setting the PATH

To use the tools, add the bin folder inside the extracted archive to your system's search path. Search for "path" in the Start menu, then select "Edit the system environment variables - control panel".

2.4.2 Uninstalling

Uninstall by deleting the extracted archive via file explorer or command line.

2.5 Using multiple versions

On Linux, upgrading the software removes previous versions. On macOS, previously installed versions remain.

Sometimes it can be useful to have access to multiple versions of the software product. This can be done by copying the entire installation directory hierarchy to another location.

Use such copied installations by locally adjusting your `PATH` environment variable or by using an explicit path in the command line.

To remove a copied product, simply delete it. Do not run the `uninstall` script, as it will affect the normally installed copy.

Note: The software product is licensed under terms that govern copying and how it can be used.

CHAPTER 3

Programming the HP-41

Welcome to Calypsi, cross-platform development tools for the HP-41 calculator.

In order to make full use of Calypsi you need a working knowledge about:

- The HP-41 calculator.
- Using command line tools on the host operating system you are using.
- Either the HP-41 RPN language, or the Nut assembly language and architecture (MCODE), or both.

3.1 Using this guide

Unless you plan to jump ahead and read here and there, there are two suggested approaches to this guide.

To get started using the tools quickly, see Installation, then try the examples in the Examples chapter.

If you want to learn in more detail how to use the tools to make your own modules, just read ahead and skip any chapter that contains facilities you do not plan to use.

3.2 Further reading

To familiarize yourself with programming the HP-41, or brush up your knowledge, here are some suggested starting points:

- The manuals supplied with the HP-41.[1]
- Extend your HP-41, W Mier-Jedrzejowicz, 1985.
- HP-41 MCODE for Beginners, Ken Emery, Synthetix 1985.
- HP-41 VASM listings.[2]

[1] The later models came with a two volumes set and the earlier a thicker single manual.
[2] These are original source code for the HP-41.

- http://www.hp41.org
- http://www.hpmuseum.org

3.3 Document conventions

When referring to a directory in the product installation, the path is given relative to the installation directory, which is assumed.

Version numbers may not match exactly the tool version you are using.

Table 3.1: Typographic conventions

Style	Used for
nop	• source code examples • file paths • command lines • commands • output
`[option]`	• optional part

CHAPTER 4

Source files and file extensions

Common to all text source files are that they are expected to be in UTF-8, which is backwards compatible with the ASCII character set. For non-ASCII characters, UTF-8 uses multibyte sequences with the highest bit set (which is unused in ASCII). Thus, it is possible to type a character such as Σ or μ[3] which are not present in ASCII. See Alpha strings for available special characters.

Unfortunately the HP-41 character set uses some very special characters, known as the hangman characters that is not available in UTF-8. If you want to use such characters, you will need to resort to using numeric constants or hex escape sequences, as they cannot be typed.

4.1 File extensions

The following table lists file extensions used with the tools.

Table 4.1: File extensions

Extension	Purpose
`.s`	Assembler source
`.o`	Object file
`.a`	Library (collection of object files)
`.lst`	List file
`.scm`	Linker rules file
`.moddesc`	Module description (XML)
`.modexport`	Module exports (XML)
`.mod`	HP-41 module format (binary output), packed 10-bit word representation
`.mod2`	HP-41 module format (binary output), 16-bit words representation for NEWT
`.elf`	HP-41 ELF output (binary output)
`.rom`	Module page as raw data (16-bit words)
`.raw`	RPN binary images (raw bytes)
`.rpn`	RPN source program

[3] When such character is used in a character or string literal, it will be translated to the corresponding HP-41 character, or LCD character depending on the context. Unicode characters can also be used in back quoted symbols.

CHAPTER 5

Nut assembler

The Nut assembler accepts a source file with HP mnemonics and generates a relocatable object file as output. This is the same mnemonics[4] used in the VASM listings[5] which makes it easier to follow them, compared to if you adopt another mnemonic set.

There are at least two other instructions sets in use, Jacobs-DeArras and ZENROM. The assembler can be configured to accept Jacobs-DeArras mnemonics using the `--jda` option, but not ZENROM.

The Nut assembler is a command line tool named `asnut`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi assembler for Hewlett-Packard Nut version 5.18

Usage: asnut [--version] [-o|--output-file OUTPUT-FILE] [-l]
             [--list-file LIST-FILE] [-I DIRECTORY] [-D IDENTIFIER]
             [-U IDENTIFIER] [-g|--debug] [--rtattr NAME=VALUE] [--weak-symbols]
             [--jda] [--core CORE] FILE
  use 'asnut --help' for detailed help

Available options:
  --version                Display version number
  -o,--output-file OUTPUT-FILE
                           Name of output file
  -l                       Generate a list file, named by appending '.lst' to
                           input file
  --list-file LIST-FILE    Generate list file, using given name
  -I DIRECTORY             Include directory
  -D IDENTIFIER            Predefine a macro
  -U IDENTIFIER            #undef a predefined macro
  -g,--debug               Produce debugging information
  --rtattr NAME=VALUE      Define a runtime attribute (identifier or quoted
```

(continues on next page)

[4] HP allows space inside some mnemonics, which is quite non-standard in assembly languages of today. This habit is only partially supported here in that you can optionally do it for instructions with an empty argument field.

[5] These are list files from HP covering major portions of the HP-41 internals and is an important source of information about its inner workings.

(continued from previous page)

```
                        string value accepted)
  --weak-symbols        Make all public symbols entries weak
  --jda                 Use Jacobs-DeArras mnemonics
  --core CORE           Core, one of 'Nut' or 'NEWT' (defaults to 'Nut')
  -h,--help             Show this help text
```

5.1 Source file format

Source lines to the assembler follow the traditional style, with a label field starting in the first column, followed by an instruction and any needed operands. Comments must be preceded by a semicolon:

```
[label[:]]   [instruction  [operands]]  [; comment]
```

The instruction can either be a mnemonic (target instruction name) or a directive. Directives start with a dot.

A label is a symbol that describes a source location. It can be defined by placing it in the first column of a source line, optionally followed by a colon.

Labels can also be declared by importing them using the `.extern` directive.

Pre-defined words used in instructions (mnemonics and operands) are case insensitive, however symbols used in operands are case sensitive. Some examples of source lines follows:

```
; Comments start with a semi-colon
;
lab:          c=0   w
              nop         ; a comment
              GOTO  lab
```

5.1.1 Symbol syntax

Symbols are case sensitive and can be of arbitrary length. A symbol starts with a letter or underscore and can be followed by letters, underscores and digits. Examples of symbols are `_4`, `a_symbol`, `abc` and `A1`.

If you want to use other characters in a symbol, it is possible to do so by surrounding the symbol by back quotes. Such symbols can be `` `another symbol` `` and `` `Table: 5` ``.

5.2 Preprocessor

The assembler uses a full featured C preprocessor to handle the input source file. The preprocessor provides the normal features you will find in a C preprocessor, the ability to include header files, macro expansions, conditional compilation and use of C style comments.

Wikipedia is a good place to look for an introduction with many examples on how to use the C preprocessor[6].

When used in the assembler, the following macros are predefined:

[6] http://en.wikipedia.org/wiki/C_preprocessor

Table 5.1: Predefined processor symbols

Preprocessor symbol	Description
`__CALYPSI_NUT__`	An integer that is 1 when the Nut target (including NEWT variant) is used.
`__CALYPSI_ASM__`	An integer that is 1 when the assembler is used.

5.3 Sections

The assembly source file is divided into sections using the `.section` directive. A section is a unit of code or data that cannot be split up in smaller pieces. Sections are laid out in memory by the linker according to rules provided by a rules file.

Before reading any input, an implicit `.section code` is applied. This means you will start off in a section that is named `code`.

Using multiple sections allow more flexible placement of the code at link time compared to using a single section. A section name can be used multiple times and each use results in an individual section fragment.

All sections are relocatable and must be defined in the linker rules file.

5.4 Expressions

Numeric expressions work in signed (2-complement) mode with range checking depending on how the end result is used. If the result exceeds the allowed range, it is reported as an error. The Operators table shows the existing standard operators. In addition to these, there are also some rather specialized relocation and section operators, refer to Relocation operators and Section operators for more details.

You can use optionally use spaces between values and operators in an expression.

Table 5.2: Operators

Operator	Precedence	Purpose
`~`	9	Unary bit-wise not
`!`	9	Unary logical not
`-`	9	Unary negate
`+`	9	Unary plus
`*`	8	Multiply
`/`	8	Divide
`%`	8	Modulo
`+`	7	Add
`-`	7	Subtract
`<<`	6	Bit shift left
`>>`	6	Bit shift right
`>`	5	Greater than
`<`	5	Less than
`>=`	5	Greater than or equal
`<=`	5	Less than or equal
`==`	4	Equal
`!=`	4	Not equal
`&`	3	Bit-wise and
`^`	2	Bit-wise exclusive or
`|`	1	Bit-wise or

5.4.1 Numeric constants

Integer constants values can be entered in decimal, binary, octal or hexadecimal. A sequence of digits that do not start with a zero is considered to be a decimal integer value. Any sequence starting with `0x` is considered a hexadecimal integer value, a `0b` prefix is considered a binary integer value and any other sequence of digits starting with `0` is considered to be an octal integer value.

Character constants can also be used and they are replaced by their corresponding ASCII value.

Some examples:

```
        .con    0x1ff           ; hexadecimal number
        .con    077             ; octal number (corresponds to decimal 63)
        .con    65              ; decimal 65
        .con    'A'             ; ASCII 65 (decimal)
        .con    0b1011          ; binary (corresponds to decimal 11)
        .con    0               ; zero (actually octal, but it is written
                                ;       the same way in decimal)
```

5.5 Location counter

The assembler converts the source program into machine code that can run on the target processor. Each instruction will end up at some location in memory in consecutive order until the next `.section` directive (or end of file is reached). The address of the current instruction can be accessed by a single period (.), usually referred to as "dot".

The dot label which contains the current instruction location is also called the location counter and it is incremented after each instruction so that it always contains the value of the start address of the current instruction.

The actual address value of the location counter is not known before the program is linked, but it can still be used in expressions. Short branches can be expressed using the location counter:

```
        gonc  .+2           ; skip next instruction
        c=c+1 x
```

However, it is often better to use labels or local labels instead, see below.

5.6 Local labels

Local labels exists in two variants. They are useful for local branch destinations in assembly source files. The first style is dollar postfix alphanumeric labels and the other variant makes use of plus and minus sign characters.

5.6.1 Dollar postfix style

Local labels end with a single `$`. The label name can either be an identifier or numeric, i.e. `loop$` or `3$`. A local label is active between two non-local labels and cannot be exported to the linker. They are meant to be used as temporary locations for short distance branching, typically short skips or local loops.

```
            a=a-c x             ; adjust counter to be 0-7
            gonc  skip$
20$:        c=c+c m             ; shift one left A.X steps
skip$:      a=a-1 x
            gonc  20$
```

After a non-local label, you can no longer refer to any local label before it:

```
foo:        c=0   w
            gonc  20$           ; error, will no longer know about 20$ above
```

As an alternative, you can just use ordinary labels and perhaps add a number to make it unique. What you do is mostly a matter of taste.

5.6.2 Sign style

Local labels can also be created using sequences of `+` or `-` characters. The entire label name needs to use the same character and the length is used when matching.

References to labels with minus characters goes backwards to the closest matching label and references to labels with plus characters goes forward to the closest matching label.

This makes it easy to see whether the destination label is before or after an instruction. Sign style local labels are allowed to pass over non-local labels.

```
+
            goto    +           ; this one goes to first + below
--
              goc     --          ; backward
+:            goc     ++++        ; goes over foobar
foobar:                           ; I am not in the way
              nop
++++:
```

Note: Sign style local labels are only usable with branch style instructions. They are not allowed in more elaborate operands where an ordinary label is allowed,

5.7 Defining constants

Constants are defined using the `.equ` directive. As with labels, you can use an optional colon after it:

```
BufNo       .equ  7
BufSize:    .equ  2 + ContentSize
```

Any expression can be used as a value, however using external symbols is subject to certain limitations imposed by relocations in the ELF object file format. In the case of valid expressions with external symbols, the value will be resolved by the linker.

5.8 Constants that are locations

Constants can also be defined with the `.equlab` directive. This is similar to the `.equ` directive, with the difference that it defines a label, which describes a location, rather than a plain number:

```
PCTOC:        .equlab  0xD7
```

Where this matters is in the interpretation of the debugging information. Labels defined using `.equlab` are treated the same as other location labels. The debugger will understand that a symbol defined using `.equlab` can be used as a location when generating disassembly listings, while symbols defined using `.equ` will not be used for locations in the disassembly listing.

5.9 Mainframe entry points

When writing MCODE programs for the HP-41, you will sooner or later find a need to call functions defined as entry points in the mainframe (HP-41 firmware).

In the `include` directory of the tools installation there is a `mainframe.h` file for this purpose. To include header files, use the `#include` preprocessor directive, see Preprocessor. The assembler is configured to find this file in the installation, so a simple:

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

early in the file will make all the mainframe entry points available.

5.10 Directives

All directives start with a single dot character to distinguish them from instructions.

The following table summarizes the directives known to the assembler:

Directive	Purpose
`.section` section-name, argument-list	Generate code for given section. The argument list is optional and consists of words separated by commas, see section kinds and modifiers below.
`.fat` symbol	Specify an entry in the function address table.
`.fatrpn` symbol	Specify an entry in the function address table for an RPN program.
`.name` string	Define function header name.
`.name` string, prompt-bits	Define function header name with prompt bits.
`.messl` string	Format a text for mainframe entry `MESSL`.
`.text` string	Text literal, lcd character encoding.
`.equ` expr	Define a symbol value.
`.equlab` expr	Define a symbol value that corresponds to a memory location.
`.public` symbol-list	Export symbols to the linker.
`.pubweak` symbol-list	Export weak symbols to the linker.
`.extern` symbol-list	Import symbol from other module.
`.require` symbol-list	Require symbol from other module.
`.suppress`	Suppress warning on the next source line.
`.rtmodel` symbol, string	Define a run-time model attribute.
`.shadow` expr	Define placement relative to another section.
`.newt_timing_start`	Annotate following instructions for execution at normal speed (NEWT only).
`.newt_timing_end`	End annotating instructions for execution at normal speed. (NEWT only).
`.macro` parameter-list	Defines a macro
`.endm`	Ends a macro definition
`.argdelim` delimiters	Define delimiters for macro arguments that contains a comma
`.end`	Stop processing the source file

A symbol-list is a list of symbols separated by commas.

5.10.1 The section directive

The `.section` directive takes the name of the section as the first argument. It can optionally be followed by a kind and modifiers to describe the section further.

```
; A code section named "code" (Text)
        .section code

; A code section named "code" (Text) that are not stored
; in relative order to other "code" section fragments in
; the same compilation unit.
        .section code, reorder

; A data section named "storage"
        .section storage, data

; A constant area in ROM that are always included in output,
; even when put in a library (provided that something else
; in the compilation unit is imported).
       .section table, rodata, root
```

Section kinds

There are four section kinds, Text, Data, RODdata, BSS and NoInit. If not specified, the section is assumed to be Text.

Section kinds and modifiers are case insensitive.

Section kind	Description
Text	Executable code.
Data	An initialized data section in read/write memory (RAM).
RODdata	An initialized data section in read only memory (ROM).
BSS	"Block Started by Symbol", intended to hold variables that are not given a value yet. A C compiler would normally zero fill such area before calling `main()`.
NoInit	Similar to BSS, but do not zero fill it.

Section modifiers

A section modifier can be used to describe further behavior of the section. There are two such modifiers which can be specified either as positive or negative.

Section modifier	Description
`noreorder`	Obey the relative order between section fragments of the same name as encountered in a translation unit.
`reorder`	Allow section fragment to be placed in arbitrary order relative to other sections with the same name. (This is the default).
`root`	Always include this section fragment in the program. This is the default for object files.
`noroot`	Only include this section fragment in the program if someone refer to a label inside it. This is the default for library files.

Note: Section fragments with the same name an `noreorder` modifier are combined by the linker into a placement group that is placed as a single unit. The effect of `reorder` (or lack of `noreorder`) is that the section fragment is given its own placement group. This also have the effect that any following section fragments are combined in a new group separate from any section fragments before it. Thus, a section fragment with `reorder` not only causes it to be placed separately from the previous ones, it also makes a placement or `noreorder` sections before and after it separate placement groups.

Section alignment

The `.align` directive specifies a given alignment in address units. It ensures that the next location will have the specified alignment. This is done by advancing the location and inserting fillers if needed.

```
; Ensure that "table" label is placed at an address that can be
; evenly divided by 4.
        .section data
        ...
        .align   4
table:  ...
```

5.10.2 Function address table entry

Each non-banked 4K ROM page has its function address table (FAT) and the `.fat` directive is used to define an entry in this table. It points the first execution address of an MCODE function. To make a proper function entry, you also need to precede it by the name of the function using the `.name` directive.

```
              .section fat
              .con    1             ; XROM number
              .con    .fatsize FatEnd ; number of entry points
              .fat    entry_HEADER  ; header word
              .fat    entry_CODE    ; a function

FatEnd:       .con    0,0           ; end of function address table

              .section code
              .name   "-MY ROM 1A"
entry_HEADER: rtn

              .section code
              .name   "CODE"
entry_CODE:
```

Each function table entry takes up two words in ROM. Normally a FAT entry will point to an MCODE function within the same 4K page, but it can actually point to another 4K page before or after. In that case they need to be at the same relative position to each other at run-time.[7]

Banked pages do not follow the traditional page layout and should therefore not have any function address table.

5.10.3 Naming functions

MCODE functions can be named using the `.name` directive that will take care of laying out the characters in the correct way in memory. This means convert the ASCII characters to the corresponding LCD characters, reverse the order and mark the end character properly. The previous example shows how to use it.

You can optionally follow the name with two values that are the prompt bits. These two values must be between `0` and `3`:

```
              .name   "FIXENG",1,3
              ;; code
              rtn

              .name   "FS?S",2,2
              ;; code
              rtn
```

[7] Recall that most 4K pages are page independent and can move at run-time. If you have FAT entries going to another 4K page, they most move together. A traditional 8K plug-in module moves this way, though it may or may not have FAT entries that go between the 4K pages.

5.10.4 ASCII and LCD characters

Character constants are encoded in ASCII while strings are translated to the corresponding LCD character set if needed, which are not necessarily identical to ASCII.

This is because it is judged to be more natural to deal with individual characters as ASCII since that is how it is in most programming languages and the tools can not anticipate from a single character constant how it will be used. It may be intended for the LCD (in which case you need to manually adjust it), but it may also be for the Alpha register, RPN program or intended for transmissions to other devices, such as a printer. In these case any automatic LCD character translation would be undesirable.

On the other hand, `.messl` and `.name` directives expects LCD characters and the `.text` directives is provided as a more generic directive of similar kind. Thus, they will all translate ASCII characters to corresponding LCD characters.

To allow for ASCII strings to be encoded more easily than using individual characters with the `.con` directive, a special case with `.con` followed by a string is allowed:

```
                .con  'A','B','C'     ; ASCII ABC
                .con  "ABC"           ; same as above
                .text "ABC"           ; This, however, will be converted to corresponding LCD characters
```

5.10.5 LCD messages

Messages to the display are often written using the `.messl` function in the mainframe. This function expects the string in a special format which is easy to achieve with the `.messl` directive:

```
                .extern MESSL

                gosub MESSL
                .messl "NO BUF"      ; string encoded in a way suitable for MESSL entry
```

5.10.6 Global symbols

Symbols are by default local to the file being assembled. Such symbols can be exposed to the global scope and imported by other source files.

For shared definitions, an alternative is to put them in an include file and define them using the `.equ` directive.

Use the `.public` directive to export named symbols and the `.extern` directive to import symbols exported from some other source file. Both directives take a comma separated list of symbols.

```
                .public BufNo, entry_HEADER
                .extern MESSL, NFRPU, BCDBIN

BufNo:          .equ   7

                .name  "BUF ROM 1B"
entry_HEADER:   rtn
```

5.10.7 Weak symbols

A weak symbol is created using the `.pubweak` directive in a similar way to `.public`. The difference is that a weak symbol may exist in multiple copies. Of these potentially multiple copies, one is selected by the linker. If there is a non-weak symbol among the weak ones, it will be picked by the linker.

Weak symbols serve a couple of purposes. They can be used for library replaceable objects where you can override a default library object using a non-weak public symbol. They are also useful for tools that generate assembly code where an identical construct may be generated multiple times, though only one is needed in the end.

5.10.8 Required symbols

A required symbol can be specified with the `.require` directive. It works similar to the `.extern` directive, with the difference that you do not need to actually use the symbol in any expression, it is being pulled in by the linker regardless.

This is mostly of interest when building modular software using libraries.

The typical use of this is that you have initialization code somewhere else built up using section fragments with the no-reorder property. The no-reorder property ensures that the code fragments appear next to each other. A section fragment is only active if someone actually refers to it. In this case the `.require` directive can be used to refer to it without actually using it. The result is that code which relies on certain initialization code fragment exists, can request that such code fragment becomes active.

5.10.9 Run-time model

It is possible to define run-time model attributes using the `.rtmodel` directive. Such attributes are checked at link time to ensure object file consistency.

One example could be that you have an attribute telling how it uses page 4. One source file that makes use of Library#4 could define an attribute:

```
; Can only be used with Library#4 in page 4
              .rtmodel page4, "Library#4"
```

Another source file makes use of page 4 for a Forth system could have:

```
; Can only be used with some Forth in page 4
              .rtmodel page4, "Forth"
```

If you try to link these two modules together will result in an error message describing that run-time model attribute `page4` has a mismatch.

Source files that do not define the `page` attribute can be linked with either. It is also possible to use the special `*` value which also means it works with either. It essentially says that I know what I am doing and it will work with whatever value is in this attribute:

```
; I work with whatever you decide to put in page 4
              .rtmodel page4, "*"
```

Functionally it equivalent to not having the attribute defined. The difference is more in the eye of the reader, you have actively considered the attribute and concluded it works with whatever interpretation there may be.

Note: A defined run-time attribute affects the entire compilation unit it appears in. If you need to have a more narrow scope for some run-time model attribute, you need to break up the source file into smaller pieces.

5.10.10 Suppressing warnings

Due to bugs in the HP-41 microprocessor, some combinations of instructions cause unpredictable behaviors. The assembler will try to spot potential problem cases and report them as warnings. There may be false warnings as the analysis made is rather shallow. There is also a risk that some potential problems are not reported as these unpredictable behaviors are not that well documented.

Currently the assembler will warn if any of the following instructions follow an arithmetic class two instruction:

```
            c=cora
            c=c&a
```

Whether this is actually a problem depends on whether the previous instruction generates a carry out of digit 13.

If you are certain that the warning is false, you can use the `.suppress` directive on the line before to suppress the warning:

```
            c=c-1   xl
            .suppress
            c=c&a
```

Another well known bug in the HP-41 microprocessor is related to setting the pointer to 13 and accessing the G register. This will cause a transfer between the G register and the byte pointed out in the C register. Since the highest pointer value is 13, it would mean it should wrap and use 0 for the upper 4 bits. This, however, will not happen. This is an example of a CPU bug that is not flagged by the assembler.

5.10.11 Shadowing sections

The `.shadow` directive makes it easy to align code in different banks to be located relative to each other in a simple, yet flexible way.

Refer to Bank switching for examples on how it can be used and a discussion on different approaches to bank switching.

5.11 List files

List files are valuable output that allows you to get an overview of the program you are writing. Both the assembler and the linker can emit list files.

Given the following example source file:

```
;;; Mainframe entry points
            .extern CLA, APPEND

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

(continues on next page)

(continued from previous page)

```
;;;
;;; DECODE - Decode the number in X into a hex number and
;;;  append it to ALPHA.  X may contain any binary data.
;;;
;;; ************************************************************

              .name "DECODE"
entry_DECODE: st=1? 13              ; clear ALPHA if called from keyboard
              gsubnc CLA
              c=regn x
              m=c
              ldi   13              ; counter
              bcex  x
1$:           pt=   0               ; loop start
              c=m                   ; get next digit
              rcr   13
              m=c
              rcr   1               ; digit to s
              ldi   3               ; 0x30-0x3f
              rcr   13              ; to C[2:0]
              acex  x
              ldi   0x3a
              ?a<c  x               ; 0-9?
              goc   2$              ; yes
              ldi   7               ; no, A-F, adjust value
              a=a+c x
2$:           acex  x
              g=c                   ; ASCII to g (for append)
              gosub APPEND          ; append character to ALPHA
              abex  x               ; decrement counter
              a=a-1 x
              rtnc                  ; done
              abex  x
              goto  1$
```

The generated list file will look as follows:

```
################################################################################
#                                                                              #
# Calypsi assembler for Hewlett-Packard Nut                     version 5.18 #
#                                                          15/Jul/2026  16:45:26 #
# Command line: -l decode.s                                                    #
#                                                                              #
################################################################################

0001              ;;; Mainframe entry points
0002                          .extern CLA, APPEND
0003
0004              ;;; ************************************************************
0005              ;;;
0006              ;;; DECODE - Decode the number in X into a hex number and
0007              ;;;  append it to ALPHA.  X may contain any binary data.
0008              ;;;
0009              ;;; ************************************************************
0010
0011  0000 085004               .name "DECODE"
```

(continues on next page)

(continued from previous page)

```
0011  0002 00f003
0011  0004 005004
0012  0006 2cc    entry_DECODE: st=1? 13              ; clear ALPHA if called from keyboard
0013  0007 ......                gsubnc CLA
0014  0009 0f8                   c=regn x
0015  000a 158                   m=c
0016  000b 13000d                ldi   13             ; counter
0017  000d 0e6                   bcex  x
0018  000e 39c    1$:           pt=   0              ; loop start
0019  000f 198                   c=m                  ; get next digit
0020  0010 2fc                   rcr   13
0021  0011 158                   m=c
0022  0012 33c                   rcr   1              ; digit to s
0023  0013 130003                ldi   3              ; 0x30-0x3f
0024  0015 2fc                   rcr   13             ; to C[2:0]
0025  0016 0a6                   acex  x
0026  0017 13003a                ldi   0x3a
0027  0019 306                   ?a<c  x              ; 0-9?
0028  001a 027                   goc   2$             ; yes
0029  001b 130007                ldi   7              ; no, A-F, adjust value
0030  001d 146                   a=a+c x
0031  001e 0a6    2$:           acex  x
0032  001f 058                   g=c                  ; ASCII to g (for append)
0033  0020 ......                gosub APPEND         ; append character to ALPHA
0034  0022 066                   abex  x              ; decrement counter
0035  0023 1a6                   a=a-1 x
0036  0024 360                   rtnc                 ; done
0037  0025 066                   abex  x
0038  0026 343                   goto  1$

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 39 words
```

The list file starts with a header that contains the name of the tool that generated it, the command line and the time it was generated.

The body contains four columns, a) the line number; b) the location counter in the current section; c) the generated opcodes; and d) the source line.

If the opcodes cannot be resolved by the assembler, they will be represented by dots.

5.12 Macro language

The `.macro` directive allows you to generate new commands that can create assembler output. A simple example follows:

```
foo           .macro  a, b
              .byte   \a
              .word   \b - 1
              .long   0
              .endm
```

This creates a new macro named `foo` which takes two arguments `a` and `b`. To use an argument inside the macro, prefix the parameter name with a backslash `\`.

5.12.1 Rules for argument substitutions

When looking for argument substitutions, the longest match is favored. This means if you have parameters called `a` and `aa`, substituting the longer name is always tried before shorter names, ignoring the order the parameters are given. As there is no way to explicitly specify the end of a parameter name inside the body, a parameter may accidently try to match characters that comes after the parameter. A good rule of thumb is to make use of space to separate entities whenever possible. This also tends to improve readability.

5.12.2 Use of local labels

Each macro expansion will create a new unique context for local labels inside the macro body. Any previous local label context is restored after the macro is expanded. Thus, local labels inside a macro will not clash or interfere with any local labels surrounding the use of the macro.

This also works when using nested macro expansions. If a macro uses another macro inside its body, that inner macro expansion will have its own private local label context, and the previous context of the outer macro expansion will be restored when the inner macro has been expanded.

Thus, you are able to do:

```
waitfield     .macro field
1$:           c=c-1 \field
              gonc  1$
              .endm

              ldi   100
              waitfield x
1$:           c=0   w
```

Which would create the following list file:

```
##############################################################################
#                                                                            #
# Calypsi assembler for Hewlett-Packard Nut                    version 5.18 #
#                                                        15/Jul/2026  16:45:26 #
# Command line: -l macro-local.s                                             #
#                                                                            #
##############################################################################
```

(continues on next page)

(continued from previous page)

```
0001              waitfield     .macro field
0002              1$:           c=c-1 \field
0003                            gonc  1$
0004                            .endm
0005
0006  0000 130064               ldi   100
0007                            waitfield x
    \ 0002 266    `1$`:       c=c-1   x
    \ 0003 3fb                gonc    `1$`
0008  0004 04e    1$:          c=0   w
0009

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 5 words
```

5.12.3 Arguments with comma

Arguments to a macro are comma separated. This poses a problem in a situation where you want an argument to contain a comma character. The `.argdelim` directive defines a start and stop character that can be used to create an argument that contains a comma character.

```
              .argdelim <>
access        .macro  arg1, arg2
              ...
              .endm

              access  0, <2,a>
```

Here `arg1` is bound to `0` and `arg2` is bound to the value `2,a`.

The delimiter can be either one or two characters and you can pick any suitable character combination. By default there are no delimiter characters defined.

If a single character combination is not suitable, you can use two characters, e.g. `<-` and `->` which would be defined as follows:

```
              .argdelim <-->
access        .macro  arg1, arg2
              ...
              .endm

              access  0, <-2,a->
```

All delimiter characters are stripped and `arg2` is bound to `2,a` here as well.

5.13 Object file format

The object file format used is ELF (Executable and Linkable Format). While ELF is a flexible and extensible format, it imposes certain limitations on what can be represented in the format.

All sections are relocatable and are given their location by the linker (or loader). There is no support for absolute sections. As a result, there are no directive to create a section that starts at a fixed address in this assembler.

Expressions that need to be resolved at link time are limited in what they can contain. They may contain a location or external symbol, and optionally a fixed offset. This basically means that you can only have a symbol, or a symbol with an constant added to or subtracted from it. The following expressions are allowed:

```
; Examples of accepted expressions
              .extern  NFRPU, buffer
              gsbp     buffer
              golong   NFRPU + 1
```

The following expressions will result in errors:

```
; Examples of rejected expressions
              .extern  NFRPU, buffer, offset
              gsbp     buffer + offset  ; error: offset not known
              golong   NFRPU * 2        ; error: only add or subtract allowed
```

The assembler will simplify expressions making it possible to use non-trivial expressions, but in the end any value used with an external symbol must be possible to reduce down to a fixed offset.

5.14 Relocations

A relocation is an entity stored in the object file format that indicates a location in the generated code that needs to be altered by the linker. Relocations are generated automatically by the assembler and you do not normally need to think about them.

A relocation entry contains the location in the generated code, expression to be relocated and which kind of relocation to perform.

5.14.1 Relocation operators

In certain contexts some additional prefix operators are available. They can be seen as relocation operators as they can be used to optionally introduce a relocation.

As they are relocations, they allow the operator to be executed at link time when the final addresses are known. However, they have the limitation that they must appear at the top level of the expression.

For the `LC` instruction which loads a nibble, you can use the `.nib0`, `.nib1`, `.nib2` and `.nib3` operators to specify that you want to get a specific nibble of a relocatable expression:

```
; Load the address of a location inside a module page.
              gosub    PCTOC          ; Get my address (page)
              pt=      5
              lc       .nib2 table    ; set lower 12 bits to 'table'
              lc       .nib1 table
              lc       .nib0 table
```

For the LDI instruction and the .CON directive the .low10 operator allows you to get the lower 10 bits of a relocatable expression.

```
; Load the lower (10 bits) part of an address
              ldi      .low10 table
```

The .low12 operator works in a similar way, but it allows the address to be anywhere inside a 4K page, provided that is aligned on an address that is a multiple of 4. It uses the available 10 bits to store the lower 12 bits, assuming that the last two bits are 0. If you use it to form an address, you need to scale the address:

```
; Load full address of a table in my own 4K page
              gosub    PCTOC          ; C[6:3]= my own location
              rcr      3              ; C[3]= page
              ldi      .low12 table
              c=c+c    x              ; scale the page offset
              c=c+c    x
                                      ; C[3:0]= address of table


              ...
              .align   4
table:        .con     data, etc
```

There are also .low8 and .high8 operators to extract the lower and upper half of a relocatable expression. They work similar to .low10, but extracts 8 bits at a time from different places to cover a 16-bit address.

For the .CON directive, the .fatsize operator can be used to get the size of the FAT. Simply use the .fatsize operator where you put the size of the FAT of your module (at location 1) and give it the address of the FAT end marker as argument. Here is an example:

```
;;; Start of module
              .section HEADER
              .con  21               ; XROM number
              .con  .fatsize fatend  ; The size of FAT
              .fat  header

;;; End marker for function address table
              .section FATEND
fatend:       .con  0,0
```

The .fatsize operator does not enforce that it should be at location 1, so feel free to use for other creative purposes. What it does is to take the address of its argument, subtract its own location and divide by 2.

Note: Relocation operators have high precedence like other unary prefix operators. If you want to access an address with an offset, you need to surround the expression by parentheses.

```
              lc    .nib2 (foo + 2)
```

5.14.2 Section operators

Section operators makes it possible to get hold of where a section is placed in memory.

Section names must be known to the assembler. If you want to refer to a section that is not used otherwise in the current assembly source file, simply declare it.

```
;;; Forward declaration
              .section elsewhere

              .section Code
              ...
```

All these operators are resolved by the linker as placement is not known before link time. An error is given if a section spans multiple memories.

Table 5.3: Section operators

Operator	Precedence	Purpose
`.sectionStart` sectionName	9	The first address of the given section.
`.sectionEnd` sectionName	9	The last address of the given section.
`.sectionSize` sectionName	9	The size of the given section.

Note: Section size corresponds to `1 + end - start`.

Warning: If you allow the linker to intermix different sections in the same allocation range, these operators will base their values on the first and last section of the given name. Different sections that are interleaved inside are silently included in the address range given by these operators.

CHAPTER 6

RPN compiler

The RPN compiler makes it possible to compile HP-41 RPN programs into the same output format as the Nut assembler. That is, it will output a relocatable object file and optionally a list file. The relocatable object file is suitable to be linked into a HP-41 module.

The RPN compiler is a command line tool named `rpncomp`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi RPN compiler for Hewlett-Packard Nut version 5.18

Usage: rpncomp [--version] [-o|--output-file OUTPUT-FILE] [-l]
               [--list-file LIST-FILE] [-I DIRECTORY]
               [--xeq-from-rom LABEL-LIST] [--rpn-output] [--raw-output]
               [--no-fat] [--prefix-labels PREFIX] FILE
  use 'rpncomp --help' for detailed help

Available options:
  --version                Display version number
  -o,--output-file OUTPUT-FILE
                           Name of output file
  -l                       Generate a list file, named by appending '.lst' to
                           input file
  --list-file LIST-FILE    Generate list file, using given name
  -I DIRECTORY             Include directory
  --xeq-from-rom LABEL-LIST
                           Specify XEQ global labels to keep as is (not change
                           to XROM calls, space separated list)
  --rpn-output             Expecting a .raw file, generate RPN source output
  --raw-output             generate .raw output
  --no-fat                 Omit auto generation of function address table entry
                           (fat)
  --prefix-labels PREFIX   Prefix created assembler labels
  -h,--help                Show this help text
```

The input source file can either be a source file or a raw binary RPN image (using `.raw` file extension).

The default behavior is to convert `XEQ` to a global alpha label, to be its corresponding ROM based `XROM` instruction. The `XROM` variant will always call the same function and uses less memory, see Global alpha labels.

Normally you do not need to use any particular option, unless you want to preserve an `XEQ` to a global alpha label which may be used as a mechanism to call a user written function stored in RAM. A more flexible way to make such call, is to store the global alpha label name in a register and call it indirectly.

In addition to creating relocatable files, the RPN compiler can convert `.raw` files to readable `.rpn` source files. Use the `--rpn-output` for this, see Converting .raw files for more details.

6.1 RPN language

The language used is similar to how it appears in the HP-41 with some minor differences due to special characters.

The RPN compiler expects the input file to be encoded in UTF-8, which means that ordinary ASCII characters are encoded as (7-bit) ASCII characters, while non-ASCII characters are encoded in sequences of multiple bytes (with the highest bit set, making them different from all ASCII characters). Thus, it is possible to type an instruction such as `ΣREG` in the way it appears in the HP-41, which is impossible to do in ASCII.

In order to help typing programs, some synonyms are also accepted:

Table 6.1: RPN synonyms

RPN	Synonyms
CLΣ	CLSIGMA
ΣREG	SIGMAREG
Σ+	SIGMA+
Σ-	SIGMA-
÷	/
X≤Y?	X<=Y?
X≤0?	X<=0?
X≠Y?	X#Y?, X!=Y?
X≠0?	X#0?, X!=0?
ENTER^	ENTER

6.2 Preprocessor

The RPN compiler uses a full featured C preprocessor to handle the input source file. The preprocessor provides the normal features you will find in a C preprocessor, the ability to include header files, macro expansions, conditional compilation and use of C style comments.

Wikipedia is a good place to look for an introduction with many examples on how to use the C preprocessor[8].

When used in the RPN compiler, the following macros are predefined:

[8] http://en.wikipedia.org/wiki/C_preprocessor

Table 6.2: Predefined processor symbols

Preprocessor symbol	Description
`__CALYPSI_NUT__`	An integer that is 1 when the Nut target (including NEWT variant) is used.
`__CALYPSI_RPN_COMPILER__`	An integer that is 1 when the RPN compiler is used.
`__CALYPSI_MODULE_USE__`	An integer that is 1 when generating code for module use.
`__CALYPSI_RAM_USE__`	An integer that is 1 when generating code for RAM use. (`--raw-output`)

6.2.1 Comments

Comments start with a semi-colon and remains active for the rest of the line. Using C style comments are also possible thanks to the preprocessor.

6.2.2 Line numbers

HP-41 RPN programs have line numbers as listed. The RPN compiler will accept programs as input either with or without line numbers. Whether the program have line numbers is sensed automatically by looking at what appears to be the first instruction in the program. If that is a number, it is assumed that there will be a line number before each instruction.[9]

Line numbers are just read and ignored and there is no requirement that they make any sense, such as being in a strict increasing order.

6.2.3 Indentation

Spaces can be used to indent code, which can be useful to increase readability. Using empty lines and comments blocks may also be useful to make the program easier to read.

6.2.4 Alpha strings

As the input file is assumed to be in UTF-8, it allows you to use certain special characters not available in ASCII and type them as they look. Refer to the Unicode table in HP-41 character set.

Append alpha

If the append character appears first in an alpha literal, it means that the string literal will be appended to the alpha register rather than replacing it.

The append character is 127 which correspond to the delete character in ASCII. While it is possible to type it using the ⊦ Unicode character, or a hexadecimal or octal escape sequence (see below), the easiest way is to precede the string literal with the greater than sign:

```
>"FOO"    ; Append FOO
```

[9] This means that a program which does not have line numbers but where the first instruction is a numeric constant, cannot be handled properly. In reality, this is a minor limitation as most programs start with a global alpha label.

Control characters

Control or individual special characters inside an otherwise normal string literal, can be inserted using escape sequences in the same way as in the C language:

```
"ABC\r"    ; followed by CR
"ABC\x0d"  ; also followed by CR (hex 0D)
"FOO\\BAR" ; \ itself must be escaped
```

Synthetic alpha strings

If you want to enter an string literal where the actual numeric character codes are more important than readable characters, the alternative syntax with a bracketed list of character codes can be used:

```
[128,255,0,3]   ; 4 characters in decimal form
```

6.2.5 Numeric constants

Numeric constants are written just as they appears in an HP-41 program. You need to separate the exponent field from the number with a space. Synthetic exponent constants are also allowed:

```
-1.303 E-13
E
E3
-E
```

6.2.6 NULL separation of numeric constants

If the source program contains two numeric constants immediately after each other, an invisible NULL will be inserted between them. This is done automatically. Such NULL instruction will not get an RPN line number, as it is invisible when examining the program in the HP-41. The NULL instruction will however be visible in the list file, without any line number will be assigned to it.

6.3 A basic example

A simple RPN program to calculate the area of a circle can be implemented as follows:

```
;; Calculate the area of a circle given radius in X

LBL "AREA"
X^2
PI
*
END
```

Running the RPN compiler on it produces a list file with the following contents:

```
################################################################################
#                                                                              #
# Calypsi RPN compiler for Hewlett-Packard Nut                  version 5.18 #
```

(continues on next page)

(continued from previous page)

```
#                                                          15/Jul/2026  16:45:26 #
# Command line: -l area.rpn                                                      #
#                                                                                #
##################################################################################

0000                    .section FAT
0000 ......             .fatrpn AREA
0000                    .section RPN
0000 002270             ;; COPY header
0002 1c6000               01 LBL "AREA"
0004 0f5000
0006 041052
0008 045041
000a 151                  02 X^2
000b 172                  03 PI
000c 142                  04 *
000d 1c8001               05 END
000f 22f

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 18 words
```

As can be seen from the list file, there are two sections generated. The first one (FAT) is meant to go into the function address table. We are after all going to produce a module and the global alpha label needs to be included there in order to be accessible. The actual word values to be put into the image cannot be determined at this point, as we do not know where the program is going to be located in the module.

The second section (RPN) is the actual program. It starts with two words that describe some properties of the program meant to be used by the `COPY` instruction, which ensures that the program can be properly copied to RAM memory. The rest are the actual RPN instructions encoded for ROM use.

Also note that RPN line numbers are generated before each RPN instruction. The leftmost column of hex numbers are the offset for the instruction within its section.

6.4 Branches and labels

Relative branches are automatically compiled for ROM use. The HP-41 can make use of short and long branches and the smallest one that can reach the destination is used.[10]

[10] Short jump reachability differs slightly in ROM and RAM on the HP-41. In a ROM the distance covered by a short branch is slightly longer compared to RAM. This means that while a short branch may be compiled properly in ROM, it may not be compilable when copied to RAM. This will only affect the execution speed of the program, it will otherwise work in the same way.

6.5 Global alpha labels

Global alpha labels are automatically found and a suitable information is generated for populating the FAT for each label. By default you will get a small FAT section fragment with this program's contribution to the final FAT. You may want to control this by hand, see FAT order.

A call to a global alpha label using `XEQ` is assumed to be within the current module. It is converted to a 2-word XROM instruction, where the generated words depends on the final location in the FAT. The actual values of these words will be determined at link time.

In some cases you may actually want to call a global alpha label and avoid the automatic translation to XROM. You can do this using the `--xeq-from-rom` command line switch:

```
LBL "FOO"
XEQ "AREA"
XEQ "FN"
END
```

To prevent `FN` from being converted to an XROM call, use the following command line:

```
$ rpncomp foo.rpn --xeq-from-rom=FN
```

In the resulting list file, `FN` has been preserved as an `XEQ` to a global alpha label. This will cause the operating system to search for it starting in catalog 1.

```
###############################################################################
#                                                                             #
# Calypsi RPN compiler for Hewlett-Packard Nut                  version 5.18 #
#                                                            15/Jul/2026  16:45:26 #
# Command line: -l call.rpn --xeq-from-rom=FN                                 #
#                                                                             #
###############################################################################

0000                    .section FAT
0000 ......             .fatrpn FOO
0000                    .section RPN
0000 003220             ;; COPY header
0002 1c2001               01 LBL "FOO"
0004 0f4000
0006 04604f
0008 04f
0009 ......               02 XROM "AREA"
000b 11e0f2               03 XEQ "FN"
000d 04604e
000f 1cc001               04 END
0011 22f

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 20 words
```

If you want to prevent multiple routines from being converted to XROM calls, either use the `--xeq-from-rom` option multiple times, or specify a double quoted space separated alpha label list:

```
$ rpncomp foo.rpn --xeq-from-rom="FN AREA"
```

Note: The default behavior when calling a global alpha label with XEQ is to convert it into an XROM call instruction for the module being built. If that global alpha label is not defined in the same RPN program, the RPN compiler will silently create an external dependency on it. The linker will later take care of fixing the XROM call, provided that the global alpha label exists in another RPN program linked together in the same module.

6.6 Using other plug-in modules

An RPN program can use instructions and functions defined in other plug-in modules. Such modules are not known to the RPN compiler from start, but they can be made available using an .import directive in the source file.

In this example the optional HP-41CX instructions and the Math Pac module is imported. The CLRGX instruction is part of the HP-41CX and the RPN program ACOSH is defined in Math Pac:

```
.import 41cx, math

01 LBL "TEST"
02 3.065
03 CLRGX
04 XROM "ACOSH"
05 END
```

Note that instructions are just typed as any instruction, while the function ACOSH in Math Pac is an RPN program so it needs to be preceded by XROM, just as it appears when the HP-41 displays the program steps. Compiling this program produces the following list file output:

```
################################################################################
#                                                                              #
# Calypsi RPN compiler for Hewlett-Packard Nut                  version 5.18 #
#                                                            15/Jul/2026  16:45:26 #
# Command line: -l import.rpn                                                  #
#                                                                              #
################################################################################

0000                    .section FAT
0000 ......             .fatrpn TEST
0000                    .section RPN
0000 003260             ;; COPY header
0002 1c8000               01 LBL "TEST"
0004 0f5000
0006 054045
0008 053054
000a 11301a               02 3.065
000c 010016
000e 015
000f 1a6072               03 CLRGX
0011 1a0066               04 XROM "ACOSH"
0013 1c6002               05 END
0015 22f
```

(continues on next page)

(continued from previous page)

```
#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 24 words
```

As the XROM codes for these optional functions are known at compile time (the modules already exists, it is nothing we are generating now), the actual words are resolved at compile time.

The imported modules have to be supplied as separate module export files. How the imports can be generated is described in the Module tool chapter.

The installation contains a set of already generated module export files in the `module-export` directory. If the desired module is available there, you can just use it as the RPN provides contains this directory in its predefined search path.

The search path also includes the current directory. If the module export file is located elsewhere, you need to use the `-I` command line option to add that directory to the search path.

6.7 Invoking local MCODE instructions

A special case is if you are building a mixed RPN and MCODE module. There are a couple of ways of doing this. In general it boils down to whether:

1. you decide up front on the XROM numbers for your MCODE instructions, or
2. you want to let the tool help you in a similar way to local RPN functions.

If you want to have fixed numbers, you can define a module exports file and let the RPN functions import it. Then you just need to ensure that you link the code in such order that the MCODE instructions end up at the specified location in the FAT.

The alternative is to let the tools figure it out using a technique that piggybacks on the same mechanism that allows automatic calls of RPN functions in the same module.

To do this you need to tell the RPN compiler that there are local MCODE instructions using the `.local` directive.[11] It looks syntactically similar to the `.import` directive, but instead of reading a module exports file, it specifies individual MCODE instruction names. In the Poker game that is part of the supplied example games module (see RPN games module), there are three MCODE support instructions called `CARD`, `MV` and `TRI`. The start of the source file may then be as follows:

```
.import 41cx

#ifdef __CALYPSI_MODULE_USE__
.local CARD, MV, TRI
#else
.import exgames
#endif
```

(continues on next page)

[11] The `.local` directive only works when building a module, as there is no FAT in RAM memory. In order to make the RPN program work in both RAM and a module, use conditional preprocessor `#ifdef` on the `__CALYPSI_MODULE_USE__` symbol around it and provide a module exports file for RAM use, see the example.

(continued from previous page)

```
01  LBL "POKER+"
02  10
   ...
54  CARD
```

Note: Using `.local` is only needed when importing MCODE instruction names. It is not needed for RPN programs. The reason is that RPN programs are wth an `XROM` prefix, so the compiler have no problem understanding what is going on.

This makes the MCODE instructions available to the RPN program. The actual XROM code used is unknown at compile time and the mechanism works as follows. It is expected that for each such MCODE instruction, there is a special named public label at each FAT entry. In the MCODE source file you will need to do something like:[12]

```
              .section FAT
`FAT entry: CARD`:
              .fat  entry_CARD

`FAT entry: MV`:
              .fat  entry_MV

`FAT entry: TRI`:
              .fat  entry_TRI

              .public `FAT entry: CARD`, `FAT entry: MV`, `FAT entry: TRI`

              .section CODE
              .name "TRI"
entry_TRI:    c=regn c
              rcr   3
```

The FAT labels are expected to be named like `` `FAT entry: XX` `` where `XX` is the name of the MCODE instruction as given to the `.local` directive. Since this label contains non-alphanumeric characters, it needs to be surrounded by back quotes.

If you expect your functions and instructions to be used from other programs or modules, you may feel that you need control the order in which the entries go into the FAT. This is possible by achieve by using different section names and by the order in which the object files are specified to the linker on the command line.

6.8 Converting `.raw` files

Since `.raw` files are just raw binary data, they are not human readable. Sometimes it makes sense to convert such programs into readable text, for inspection or editing. You can perform such conversion using the `--rpn-output` option. In addition to this option, you need to specify a `.raw` file. The output file is written to the current directory, using the same base name as the `.raw` file, but with a `.rpn` file extension instead.

Calls to external ROMs currently shows up as `XROM` instructions with numbers.

If you have many `.raw` files you want to convert, here is a way using `xargs` which is available if you are using a POSIX.2 compliant system:

[12] This manual work is not needed when using RPN functions, as the compiler will create the FAT entries and public labels automatically.

```
$ ls path-to-raw-files/*.raw | xargs -n1 -J % rpncomp --rpn-output %
```

6.9 Merged XROM instructions

Some modules allow XROM instructions to have postfix operands. Such combined instructions are not supported by the HP-41 operating system, but is possible using some special techniques. There are a couple of variants available, and they rely on that the XROM instruction consumes the following step (which is an ordinary stand alone RPN instruction) and use it as an argument of some kind.

Current support for this is limited and only supports the style used by the Ladybug module.

The `.postfix` directive tells the RPN compiler that you have an XROM with some kind of postfix operand. It takes a list of instructions it applies to, followed by the operand style they use inside parentheses:

```
.import ladybug

.postfix DSZI, DECI, INCI, LDI, STI  (semi-merged-postfix 00)
.postfix CLRI, CB, SB, VIEWI, B?     (semi-merged-postfix 00)
.postfix TST                         (semi-merged-postfix 00)
.postfix SL, SR, ASR, RL, RR         (semi-merged-postfix 01)
.postfix RLC, RRC                    (semi-merged-postfix 01)
.postfix MASKL, MASKR                (semi-merged-postfix  8)
.postfix SEX, WSIZE                  (semi-merged-postfix 16)
.postfix ALDI, BITSUM                (semi-merged-postfix  X)
.postfix CMP                         (semi-merged-postfix  Y)
.postfix #LIT                        (semi-merged-integer-literal)
.postfix =I, ≠I, <I, <=I, <>I        (semi-merged-dual)

#define integer #LIT

LBL "LADY"
WSIZE 16
integer 0xffe
<>I 01 M
STI IND 02
MASKL 4
LBL 00
VIEWI X
PSE
DSZI X
GTO 00
END
```

As can be seen, you still need to import the module to being in the available instructions. The `.postfix` directive is used to dress them up as instructions that actually take operands.

Note: The integer literal instruction (`#LIT`) starts with a hash symbol which will cause an error from the C preprocessor if used first on a line, as the preprocessor will try to interpret it as a preprocessor directive. To work around it, the `#define` defines a synonym `integer` which expands to the `#LIT` instruction.

Following the postfix style, the default argument can optionally be specified. If it is specified, the compiler can avoid generating an explicit postfix instruction when the default will perform the same operation. `WSIZE` uses an argument that is the same as the default in this example program.

In the example `VIEWI` is a secondary function and gets an extra byte that represents its function code due to this.

Dual argument functions as `<>I` are entered with the function name first followed by its two operands. This is also how they are entered from the keyboard on the HP-41.

The following list file shows the result:

```
################################################################################
#                                                                              #
# Calypsi RPN compiler for Hewlett-Packard Nut                  version 5.18 #
#                                                          15/Jul/2026  16:45:26 #
# Command line: -l ladybug-postfix.rpn                                         #
#                                                                              #
################################################################################

0000                    .section FAT
0000 ......             .fatrpn LADY
0000                    .section RPN
0000 007230             ;; COPY header
0002 1c0001               01 LBL "LADY"
0004 0f5000
0006 04c041
0008 044059
000a 1a4008               02 WSIZE
000c 1a4001               03 #LIT 0xffe
000e 1f200f               04 [15,254]
0010 0fe
0011 1a403a               05 <>I 01 M
0013 1f3004               06 [4,1,117]
0015 001075
0017 1a402c               07 STI IND 02
0019 1f1082               08 [130]
001b 1a4029               09 MASKL 04
001d 1f1004               10 [4]
001f 101                  11 LBL 00
0020 1a403a               12 VIEWI X
0022 1f2005               13 [5,115]
0024 073
0025 189                  14 PSE
0026 1a4031               15 DSZI X
0028 1f1073               16 "s"
002a 1b100d               17 GTO 00
002c 1c0006               18 END
002e 22f

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 49 words
```

In the generated listing, such XROM instructions are generated as two separate instructions. The postfix byte is held in alpha string literals following the actual instruction. To make it easier to read and save you from having to look it up from some table, instructions are shown together with their operands. However, close studying of the generated opcodes and line numbers reveals that they are actually two instructions from a program list point of view.

CHAPTER 7

Module tool

The module tool works with HP-41 modules files (`.mod` extension). It can either extract information from an existing module file or put together a new module.

The module tool is a command line tool named `modtool`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi module tool for Hewlett-Packard Nut version 5.18

Usage: modtool [--version] [-e|--export-files FILE] [-c|--verify-checksums]
               ([--extract-module-descriptor] |
               [--no-extract-module-descriptor]) ([--extract-module-export] |
               [--no-extract-module-export]) ([--extract-rom-pages] |
               [--no-extract-rom-pages]) ([--summary] | [--no-summary])
               [--no-secondaries] [-l] [--list-file LIST-FILE]
               [--memories-expression EXPRESSION] [--program-root SYMBOL]
               [--root-symbol SYMBOL] [--program-start SYMBOL]
               [--copy-initialize SECTION] [--no-copy-initialize SECTION]
               [--no-automatic-placement-rules] [--raw-multiple-memories]
               [--no-merge-raw-memories] [--force-output] [--no-auto-libraries]
               [--cstartup VALUE] [--no-tree-shaking] [--output-format FORMAT]
               FILE...
  use 'modtool --help' for detailed help

Available options:
  --version                Display version number
  -e,--export-files FILE   Base name of export files (defaults to base name of
                           input file)
  -c,--verify-checksums    Verify the checksum in each ROM page
  --extract-module-descriptor
                           Extract module header meta data
  --no-extract-module-descriptor
                           Do not extract module header meta data
  --extract-module-export  Extract a module exports file
  --no-extract-module-export
```

(continues on next page)

(continued from previous page)

```
                        Do not extract a module exports file
  --extract-rom-pages   Extract individual ROM pages to files
  --no-extract-rom-pages Do not extract individual ROM pages to files
  --summary             Show FAT summary
  --no-summary          Do not show FAT summary
  --no-secondaries      Do not look for secondary FATs
  -l                    Generate a list file, defaults to 'modtool.lst'
  --list-file LIST-FILE Generate list file, using given name
  --memories-expression EXPRESSION
                        Expression that extracts the list of memory
                        descriptions from the .scm file, defaults to
                        'memories'
  --program-root SYMBOL Program root point, defaults to
                        '__program_root_section'
  --root-symbol SYMBOL  Add a root symbol
  --program-start SYMBOL Program start symbol, defaults to '__program_start'
  --copy-initialize SECTION
                        Override default and initialize this section by
                        copying
  --no-copy-initialize SECTION
                        Override default and do not initialize this section
                        by copying
  --no-automatic-placement-rules
                        Do not add rules to a .scm linker rules file
  --raw-multiple-memories Allow multiple memories for raw style output format
  --no-merge-raw-memories Never attempt to merge raw memories
  --force-output        Ignore (some) errors and attempt to generate output
  --no-auto-libraries   Do not automatically add runtime libraries
  --cstartup VALUE      Define the C startup to be used, synonym to '--rtattr
                        cstartup=VALUE'
  --no-tree-shaking     Do not perform tree shaking and keep unused section
                        fragment in output
  --output-format FORMAT Format, one of 'MOD1' or 'MOD2', defaults to 'MOD1'
                        (in addition to the ELF/DWARF output)
  -h,--help             Show this help text
```

7.1 Extraction mode

If a module file is given on the command line module tool runs in extraction mode. The following files and information can be extracted:

What	Description
module descriptor	Module meta data as an XML file. This file is input to the linker as the module header. Uses `.moddesc` file extension.
module export	Module entry points as an XML file. This file can be imported by the RPN compiler to allow XROM calls to be made to the imported module. Uses `.modexport` file extension.
ROM pages	These are the actual page images. Uses `.rom` file extension.
Summary	Human readable FAT table summary on stdout. No file is created.

By default all of them are done. You can pick individual ones by using the appropriate positive option. This will result in that only those specified are done.

If you specify negative options, all but those are done.

Some examples follow. To only see the easy readable FAT table summary (no files created):

```
$ modtool --summary file.mod
```

To extract everything but `.rom` image files:

```
$ modtool --no-extract-rom-pages file.mod
```

To get the summary and the module export (XML file):

```
$ modtool --summary --extract-module-export file.mod
```

7.2 Creation mode

In creation mode, you give module tool a single `.moddesc` module description file and as many `.rom` page files that are mentioned in the description file. A single output module file is created based on the input files.

7.3 Module description file

This file contains the meta data of the module file. It is a required input file when a new module file is to be created. Both the linker and the module tool can create module files and takes a module description file as input.

The easiest way to create a new module description file is to base it on an existing module with similar layout as the module you want to create. Simply locate a suitable module and run the module tool on it to extract its module description file. Then rename and edit the description file to fit its new purpose.

```
$ modtool advantage.mod
```

The resulting file `advantage.moddesc` looks as follows:[14]

```
<?xml version="1.0" encoding="UTF-8"?>
<ModFile FileFormat="MOD1" Title="Advantage Pac" Version="B"
         PartNumber="00041-15055" Author="Hewlett-Packard"
         Copyright="Hewlett-Packard"
         License="Hewlett-Packard Company makes no warranty as to the
                  accuracy or completeness of the foregoing information
                  and hereby disclaims any responsibility therefore."
         Comments="" Category="2" Hardware="0" MemoryModules="0" XMemoryModules="0">
  <Page Name="AdvL1-1B" Identity="ADV1" Page="Lower" PageGroup="1"
        Bank="1" BankGroup="1" RAM="0" WriteProtect="0" FAT="1"/>
  <Page Name="AdvU1-1B" Identity="ADV2" Page="Upper" PageGroup="1"
        Bank="1" BankGroup="1" RAM="0" WriteProtect="0" FAT="1"/>
  <Page Name="AdvU2-1B" Identity="ADV3" Page="Upper" PageGroup="1"
        Bank="2" BankGroup="1" RAM="0" WriteProtect="0" FAT="0"/>
</ModFile>
```

[14] The output have been formatted slightly to keep the line length down and to improve readability.

You can use this as a starting point. You probably want to, a) rename the file; b) change the page names to correspond to the memory names you have defined; c) change the header attributes; and d) change the `Identity` attributes.

In other words, you will need to make several changes, but on the good side, they are mostly trivial. If you choose the module file to start with some care, you get the correct structure and the more tricky items filled in.

7.4 Module exports file

The module exports is used by the RPN compiler and the barcode generating tools. It is useful when you write RPN programs that use plug-in modules. It makes it possible to refer to instructions and functions in a used module by name. The RPN compiler and barcode tool takes care of using the appropriate XROM numbers.

7.5 Extracting ROM pages

In extraction mode, all individual ROM pages will be extracted unless you specify the `--no-extract-rom-pages` option. A separate ROM file is created for each page. The filename is constructed from the page name found in module meta data with an file name extension `.rom`.

The `.rom` format is very simple, each 10-bits word is encoded as two bytes, with the upper part first (big endian).

7.6 Technical details

To implement the extraction of entry points, the module tool loads the pages of the module file into a simulated HP-41 memory system based on its layout. After that, it will look around in the fictive memory system and decode the function address tables, extracting MCODE entries and global RPN labels.

The HP-41 uses some special characters that cannot be represented in ASCII. If such character is encountered, it is converted to a suitable Unicode character.[13]

[13] Hint: If you have trouble typing these special Unicode characters in your text editor, open and copy the name from the module exports file.

CHAPTER 8

Linker

The linker is used to combine the output of the assembler and the RPN compiler into a module file.

The linker is a command line tool named `lnnut`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi linker for Hewlett-Packard Nut version 5.18

Usage: lnnut [--version] [-o|--output-file OUTPUT-FILE] [-g|--debug]
             [--semi-hosted] [--no-data-init-table-section]
             [--override IDENTIFIER] ([--cross-reference] |
             [--no-cross-reference]) [--rtattr NAME=VALUE] [--verbose] [-l]
             [--list-file LIST-FILE] [--memories-expression EXPRESSION]
             [--program-root SYMBOL] [--root-symbol SYMBOL]
             [--program-start SYMBOL] [--copy-initialize SECTION]
             [--no-copy-initialize SECTION] [--no-automatic-placement-rules]
             [--raw-multiple-memories] [--no-merge-raw-memories]
             [--force-output] [--no-auto-libraries] [--cstartup VALUE]
             [--no-tree-shaking] [--output-format FORMAT]
             [--extra-output-formats FORMAT] [FILE...]
  use 'lnnut --help' for detailed help

Available options:
  --version                Display version number
  -o,--output-file OUTPUT-FILE
                           Name of output file
  -g,--debug               Produce debugging information
  --semi-hosted            Enable debug stubs for semi-hosting
  --no-data-init-table-section
                           Do not generate any data_init_table section (mainly
                           useful for assembly projects)
  --override IDENTIFIER    Override symbol in archive library (treat it as weak)
  --cross-reference        Include cross reference and map information in the
                           list file (this is the default)
  --no-cross-reference     Do not include cross reference and map information in
```

(continues on next page)

(continued from previous page)

```
                          the list file
  --rtattr NAME=VALUE     Specify runtime attribute to select specific
                          alternative from library (e.g. printf=float,
                          scanf=nofloat)
  --verbose               Generate more detailed output
  -l                      Generate a list file, defaults to 'lnnut.lst'
  --list-file LIST-FILE   Generate list file, using given name
  --memories-expression EXPRESSION
                          Expression that extracts the list of memory
                          descriptions from the .scm file, defaults to
                          'memories'
  --program-root SYMBOL   Program root point, defaults to
                          '__program_root_section'
  --root-symbol SYMBOL    Add a root symbol
  --program-start SYMBOL  Program start symbol, defaults to '__program_start'
  --copy-initialize SECTION
                          Override default and initialize this section by
                          copying
  --no-copy-initialize SECTION
                          Override default and do not initialize this section
                          by copying
  --no-automatic-placement-rules
                          Do not add rules to a .scm linker rules file
  --raw-multiple-memories Allow multiple memories for raw style output format
  --no-merge-raw-memories Never attempt to merge raw memories
  --force-output          Ignore (some) errors and attempt to generate output
  --no-auto-libraries     Do not automatically add runtime libraries
  --cstartup VALUE        Define the C startup to be used, synonym to '--rtattr
                          cstartup=VALUE'
  --no-tree-shaking       Do not perform tree shaking and keep unused section
                          fragment in output
  --output-format FORMAT  Format, one of 'MOD1' or 'MOD2', defaults to 'MOD1'
                          (in addition to the ELF/DWARF output)
  --extra-output-formats FORMAT
                          Format, only accepted argument is 'MOD2'
  -h,--help               Show this help text
```

You need to specify the object files, a module description file and a linker rules file as input.

From each object file comes named relocatable section fragments that are to be placed at a final position in some memory. The link process is defined by a set of rules that govern where section fragments can be placed.

8.1 Memory

A memory is a named entity that gives a continuous range of storage locations. On the HP-41, a 4K page is normally treated as a memory. A description of a memory contains the following attributes:

name The name of the memory. The memory name is the identifier used for a particular memory area

address range The start and end address (inclusive) specify the address range of the memory.

position Tells whether the memory is fixed or position independent. For a page relocatable module you want to specify this as `(position independent)`. The alternative is `(position fixed)` if you are writing code for a fixed address.

section Here you put the section names you want to bind to the current memory.

checksum An optional address where a checksum will be stored. For the HP-41 it should be the highest address of a 4K page (last address in memory). You should also specify the checksum algorithm to be used. The normal value to use here is `(checksum #xFFF hp41)`.

fill word Value used to fill unused locations in the memory. Defaults to zero, but can specified as `(fill 0)`.

8.2 Linker rules file

The rules file (extension `.scm`) describes how the section fragments are to be laid out in memory. This file serves three purposes, a) it defines the actual memories; b) it defines which sections to expect; and c) it describes where the sections can be placed in memory.

8.2.1 Memory rule

A memory rule defines a memory. It also defines which sections (by name) that will go into that memory. There can be multiple memory rules in a rule file and you will need one for each memory block.

A simple rule file for a relocatable 4K module where you have used a single section `CODE` may look as:

```
(define memories
  '((memory Page1 (address (#x0 . #xFFF) (position independent))
            (section CODE)
            (checksum #xFFF hp41)
            (fill 0))))
```

While the use of spacing for indentation here is unimportant to the linker tool, it makes the file easier to read. However, the use of characters like the single quote, period and parentheses are important to make the linker tool able to parse the file correctly.

A slightly more advanced example with a 8K relocatable module with some sections:

```
(define memories
  '((memory Page1 (address (#x0 . #xFFF)) (position independent)
            (section (FAT1 #x0) Code1)
            (checksum #xFFF hp41)
            (fill 0))
    (memory Page2 (address (#x1000 . #x1FFF)) (position independent)
            (section (FAT2 #0) Code2)
            (checksum #x1FFF hp41)
            (fill 0))))
```

8.2.2 Section placement

There are several ways to control how sections are placed. Here we walk through some typical cases, consider this linker rules file:

```
(define memories
  '((memory Page (position independent)
            (bank 1) (address (#x0 . #xFFF))
            (section (ID FAT FATEND #x0)
                     (CodeQ0 (#x0 . #x3FF))
```

(continues on next page)

(continued from previous page)

```
                Code1 Code2
                (Tail #xFF4))
          (checksum #xFFF hp41)
          (fill 0))))
```

Free placement

The `Code1` and `Code2` sections appear alone (no surrounding parentheses). Sections with these names can be placed anywhere in the defined memory.

Restricted placement

The `CodeQ0` section has an address range, which should be a sub-range of the memory range. It can be placed anywhere within the sub-range.

Fixed placement

The `Tail` section will be placed starting at address `FF4`.

Fixed section group placement

A FAT (function address table) consists of three parts. First an identifier, then a table of functions and finally an end marker. The section group `ID`, `FAT` and `FATEND` are placed together starting at address 0. The order in which section fragments are placed are important when setting up a FAT, as this also gives the XROM number allocation.

In this example section fragments are placed so that all `ID` fragments comes first, followed by all `FAT` fragments and finally the `FATEND` fragments.

In a real setup, `ID` and `FATEND` probably consist of single section fragments. However, the `FAT` section is likely to consist of multiple section fragments.

The order in which `FAT` section fragments are placed here, is the same as the order in which the object files appear on the command line to the linker. If there are multiple `FAT` sections within the same object file, the order among them are decided by the order in which they were encountered when compiling the source file.

Note: You will most likely want to ensure consistent ordering in the FAT, as the actual XROM number allocated for each function depends on it.

8.2.3 Banking

To make a module with banked pages,[16] you place more than one memory at the same address, but with different bank numbers using the `bank` attribute.

The following rule file shows how to use the `bank` attribute for banked memory:

[16] This means that more than one page are at the same address, though only one will be visible to the microprocessor at a given time.

```
(define memories
  '((memory CodeBank1 (address (#x0 . #xFFF) (position independent))
            (section FAT CodeBank1)
            (checksum #xFFF sum-carry)
            (bank 1)
            (fill 0))
    (memory CodeBank2 (address (#x000 . #xFFF) (position independent))
            (section CodeBank2)
            (checksum #xFFF sum-carry)
            (bank 2)
            (fill 0))))
```

These bank numbers are only used for detecting problems when resolving external references at the link stage.

The actual bank and page layout of the module is described in the module description file (`.moddesc`). The memories in the linker control file are paired with pages in the module description file based on their names. The meta data of the module description file is combined with the memory images to form the final module image.

8.3 Linker list files

The linker list file that can be optionally generated shows how the section fragments are distributed in the memories and the total size of memory allocated.

```
################################################################################
#                                                                              #
# Calypsi linker for Hewlett-Packard Nut                        version 5.18 #
#                                                             15/Jul/2026  16:45:26 #
# Command line: -l poker.o roulette.o header.o PokerSupport.o Plugin4K.scm    #
#               games.moddesc -o games.mod                                     #
#                                                                              #
################################################################################

####################
#                  #
# Memories summary #
#                  #
####################

Name Range     Size    Used    Checksum  Largest unallocated
-------------------------------------------------------------
Page 0000-0fff 1000     45.1%  136       08c6


####################
#                  #
# Sections summary #
#                  #
####################

Name   Range      Size    Memory Fragments
-----------------------------------------
HEADER 0000-0003  0004    Page   1
FAT    0004-000d  000a    Page   3
```

(continues on next page)

(continued from previous page)

```
FATEND 000e-000f  0002    Page   1
RPN    0010-037f  0370    Page   1
Code   0380-0572  01f3    Page   1
RPN    0573-0720  01ae    Page   1
Code   0721-072d  000d    Page   1
TAIL   0ff4-0ffe  000b    Page   1


##########################
#                        #
# XROM allocation summary #
#                        #
##########################

XROM      Address   Kind        Function
-----------------------------------------------
21,00     872d      MCODE       -GAMES EX 1A
21,01     8549      MCODE       CARD
21,02     8528      MCODE       MV
21,03     8383      MCODE       TRI
21,04     8575      RPN         "ROULETT"
21,05     8012      RPN         "POKER+"


################
#              #
# Object files #
#              #
################

Unit Filename       Archive
--------------------
  0  poker.o        -
         >  FAT 0002
         >  RPN 0370
  1  roulette.o     -
         >  FAT 0002
         >  RPN 01ae
  2  header.o       -
         >  Code   000d
         >  FATEND 0002
         >  HEADER 0004
         >  TAIL   000b
  3  PokerSupport.o -
         >  Code 01f3
         >  FAT  0006

###################
#                 #
# Cross reference #
#                 #
###################

Section 'HEADER'  placed at address 0000-0003 of size 0004
(header.o unit 2 section index 2)
```

(continues on next page)

(continued from previous page)

```
`FAT entry: CARD` in section 'FAT'  placed at address 0004-0009 of size 0006
(PokerSupport.o unit 3 section index 2)
    Defines:
        `FAT entry: TRI` = 10000008
        `FAT entry: MV` = 10000006
        `FAT entry: CARD` = 10000004
    Referenced from:
        (poker.o unit 0 section index 3)

`FAT entry: ROULETT` in section 'FAT'  placed at address 000a-000b of size 0002
(roulette.o unit 1 section index 2)
    Defines:
        `FAT entry: ROULETT` = 1000000a

`FAT entry: POKER+` in section 'FAT'  placed at address 000c-000d of size 0002
(poker.o unit 0 section index 2)
    Defines:
        `FAT entry: POKER+` = 1000000c

fatend in section 'FATEND'  placed at address 000e-000f of size 0002
(header.o unit 2 section index 3)

`POKER+` in section 'RPN'  placed at address 0010-037f of size 0370
(poker.o unit 0 section index 3)
    References:
        `FAT entry: CARD` in (PokerSupport.o unit 3 section index 2)
        `FAT entry: MV` in (PokerSupport.o unit 3 section index 2)
        `FAT entry: TRI` in (PokerSupport.o unit 3 section index 2)

entry_TRI in section 'Code'  placed at address 0380-0572 of size 01f3
(PokerSupport.o unit 3 section index 3)

ROULETT in section 'RPN'  placed at address 0573-0720 of size 01ae
(roulette.o unit 1 section index 3)

header in section 'Code'  placed at address 0721-072d of size 000d
(header.o unit 2 section index 4)

Section 'TAIL'  placed at address 0ff4-0ffe of size 000b
(header.o unit 2 section index 5)

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 1849 words
```

The Memories summary section summarizes the different memories (pages) used in the module. Here you will see the size of the memory, how much memory is in use and the largest unallocated continuous memory block. The checksum column lists the calculated checksum, which is also saved in the memory image.

The Sections summary gives an idea of where the sections are located and how many section fragments there are for each one.

The XROM allocation summary lists the FAT and gives the entry point for each routine (using some fictive page address if the page is relocatable).

Finally the Memory sizes summarizes the total amount of memory in use. Note that the amount of memory is given in decimal (addresses and sizes above are in hexadecimal).

8.4 Module files

Output from the linker is in a binary format called module file. This is a special binary format defined for the HP-41 which is inspired by how actual modules looks. A one single file is used to describe a plug-in module. Multiple 4K pages can reside in a module and they can either be loaded to a fixed addresses, being relocatable and even have multiple pages that are to be relocated together. It can also describe banked 4K pages and tell whether the module contains special hardware.

To create a module file, a module description file is needed. This contains the meta data of the module file.

The easiest way to create a `.moddesc` file is to use the module tool `modtool` to extract one from an existing `.mod` file that have similarities with the module you are going to create (see Module description file).

8.5 More on linker rules

The linker rules file is actually a source file for the Scheme programming language, which is the reason for the choice of the `.scm` file extension.

You do not need to be familiar with Scheme to specify linker rules, simply follow the examples to set things up.

The shown examples are just a Scheme "program" that consists of a single variable named `memories` which is bound to a data structure. The linker runs a Scheme interpreter to read the file and then look for the resulting `memories` variable and peek into its contents. This has two implications, a) the syntax of the file is dictated by Scheme, which is based on s-expressions[15]; and b) it is possible to use a more elaborate program with Scheme macros to generate the memory rules.

[15] https://www.wikipedia.org/wiki/S-expression

CHAPTER 9

Librarian

The librarian makes it possible to combine multiple object files into a single library. This is sometimes called an archiver.

The librarian is a command line tool named `nlib`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi librarian for none version 5.18

Usage: nlib [--version] [FILE...]
  use 'nlib --help' for detailed help

Available options:
  --version                Display version number
  -h,--help                Show this help text
```

The main use-case for a librarian is to build collections of smaller fragments, which can be anything from support routines to full fledged applications. Such collection is called a library, static library or an archive.

9.1 Creating a library

To create a library, simply list all object files (`.o` extension) and a single output file (`.a` extension).

```
$ nlib lib41.a obj1.o obj2.o ashfx.o xtoal.o
```

The created library is typically used as input to the `lnnut` linker by just specifying it on the command line. When linking with a library, only those section fragments that are actually used are included in the final output.

CHAPTER 10

NEWT

The NEWT (Nut Enhanced With Turbo, also known as the HP-41CL) target variant is supported by the tools. The main difference when programming for the NEWT is that it uses speed annotation bits to control the pace of timing loops. This is accomplished using 16-bit words compared to the usual 10-bit words used by the Nut target.

In order to use 16-bit words in output you need to enable it using the `--core=NEWT` command line option, which is accepted by all tools where it is relevant.

In addition to speed annotation, the `--core=NEWT` command line option also enables recognition of the `WCMD` instruction.

10.1 Speed annotation

Use NEWT directives to insert speed annotation bits in a fragment of code where normal speed is to always to be obeyed:

```
nullTest:     ldi     200
              .newt_timing_start
              disoff
72$:          rst kb
              chk kb
              gonc    73$
              c=c-1   x
              gonc    72$
              distog
              .newt_timing_end
              gosub   NULTST
```

When compiled with `--core=NEWT` the list file shows the generated 16-bit words:

```
0367  0049 013000c8 nullTest:     ldi     200
0368                              .newt_timing_start
```

(continues on next page)

(continued from previous page)

```
0369  004b 32e0                disoff
0370  004c 33c8     72$:       rst kb
0371  004d 33cc                chk kb
0372  004e 303b                gonc    73$
0373  004f 3266                c=c-1   x
0374  0050 33e3                gonc    72$
0375  0051 3320                distog
0376                           .newt_timing_end
0377  0052 03190038            gosub   NULTST
```

Here you can see that generated words are four digits compared to the usual three. `0x3000` is also added to each instruction inside the NEWT timing annotated block.

10.2 Module files

The traditional modules files (`.mod` file extension) stores packed 10-bit words. There is no place for the speed annotation bits. Prior to the `.mod` files the use of `.rom` files were common. These are single separate files for a single page using 16-bit words. Any rules associated with a multi-page module need to be handled manually. A module file allows pages to be combined together with some meta data that describes the combination and also contains rules for how the pages can be placed in memory.

The Calypsi tool chain provides a module file variant with extension `.mod2` which is like `.mod` files, but the pages are stored as 16-bits words the same way as `.rom` files.

10.2.1 Dual format output

If your module contains NEWT speed annotations it is desirable to output a `.mod2` file. However, many tools only support ordinary `.mod` files and there are lots of people using standard HP-41 calculators. In such cases it makes sense to output and distribute both formats, which can be accomplished by giving the `--extra-output-formats=mod2` command line option to the linker.

CHAPTER 11

Bank switching

The bank system on the HP-41 allows for up to four pages to reside in the same 4K address page. The bank switch mechanism is actually implemented by the memory system. The CPU itself is totally unaware of it.

In practice, there are two basic ways to approach writing bank switched code.

The first approach is to use (at least) two address pages. Bank switching can then be done in one of the pages (which is not bank switched), and the bank switch affect the other page. This makes it fairly easy to control and layout memory and is the approach used in the HP-41CX operating system.

The second approach is to switch bank inside the page you are executing in, essentially pulling the rug on yourself, or rather replacing it on the fly. For this to work, you need to place code so that the other bank is ready to take over immediately where you happen to be executing when the bank switch takes place.

The advantage of the first approach is that it simpler and allows for somewhat more flexibility. The disadvantage is that it occupies two address pages.

This tool set provides support to do either way and makes the latter single page approach virtually as simple to implement as the two page approach.

11.1 Hardware behavior

The original bank switch mechanism as defined by HP, relied on a specific memory chip that was used to load the ROM images. This chip had some peculiarities in that it put constraints on what instruction could appear before the bank switch instruction. One can see this if one study the code of the HP-41CX or the Advantage ROM, many bank switch instructions are preceded by a `goto` instruction (branching to the next line). HP also imposed some restrictions on that certain areas in bank switched modules were required to contain specific code, otherwise the load software HP used would refuse to accept the module image.

As HP no longer produce these memory chips or provide any module building services, we are no longer restricted by these limitations and rules.

The reason for the imposed specific code was most likely to make it possible to enable bank pages from diagnostic software, which makes it possible to verify the checksum of banked pages.

Current hardware available are Clonix, NoV series and NEWT (41CL). The MLDL2000 is reasonable recent, but sadly not available anymore. All of them support running banked modules.

The MLDL2000 and NEWT provide bank switching by "pairing pages", that is, the two pages that are covered by the same physical port are bank switched together. Some special rules apply for low memory in NEWT as it needs to handle the HP-41CX mainframe image where bank switching take place between page 3 and 5. Clonix differs in that bank switching takes place in all pages in the module, not just the two pages next to each other.

What it means in practice is that provided we follow the golden rule that we should never leave a secondary page active when we give control back to the operating system, then we are safe. Calling mainframe entry points are OK, provided that they return to the caller. For routines that will (or may) not return, we should only call them from the main bank (bank 1). We can also relax the rules that HP used to impose one banked modules.

If we use the approach to have two pages next to each other, where one switches the bank of the other and then make calls into it, the first page will actually also switch bank! This means that we need to mirror the unbanked page into the secondary banks of that page. Fortunately, there is no need to duplication the page, the hardware just need to the configured to show that page in the other bank too. This is either done automatically by the load tool, or is trivial to do in the hardware configuration, but can be worth knowing.

11.2 Single page

To make it possible to arrange code to appear in the right place in different banks, a mechanism exists for setting up a section fragment to shadow another one. Essentially, it means that you can specify that a section fragment (the shadow) is placed relative to another section fragment, but in a different memory bank.

To make it work you need to insert the appropriate bank switch instruction at the right place and provide linker rules to get the different section fragments to reside in the desired bank.

The `.shadow` directive creates a shadow relation. The following macro makes it easy to switch bank on the fly:

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

Provided you have you have defined this macro, you can use it to switch bank in the following way:

```
              .name   "ADD"
ADD:          s9=0
              gsbp    FindBufferGetXSaveL
              switchBank 2
              c=regn  X
```

This prelude of an `ADD` instruction does an initial few instruction in the main bank, then switch over to bank 2 where the `c=regn X` and following instructions are.

In a list file, it looks as follows:

```
1786  004a 084004               .name   "ADD"
1786  004c 001
```

(continues on next page)

(continued from previous page)

```
1787  004d 244    ADD:          s9=0
1788  004e ...... ADD_2:        gsbp    FindBufferGetXSaveL
1788  0050 ...
1789                             switchBank 2
    \ 0051 180                 enrom2
    \ 0052        10$:
    \ 0000                     .section Code2
1790  0000 0f8                   c=regn  X
```

As can be seen, there is just a single `enrom2` instruction inserted in the code. The code starting with `c=regn X` appears in a new section fragment.

11.2.1 Details of the switchBank macro

The macro takes a single argument, the bank it should switch to, which can be any number from 1 to 4. This number is both used as part of the `enrom` instruction and the new section fragment. It just happens that the `enrom` instruction wants a number 1 to 4, and we can just use make up a suitable section name based on it as well.

After executing the `enrom` instruction, the CPU will want to execute the instruction immediately following it. As we just switched bank, that instruction will be fetched from the bank we switched to.

To make it work, we want to align the code in the other bank so that it ends up at the location following the `enrom` instruction of the previous bank. To get a reference to that location, the macro defines a label after the `enrom` instruction (the local `10$` label).

As the following code has to be in a different bank, the macro starts with a new section fragment and names it with the bank number as suffix. There is nothing magical about the section name here, it can be anything, but doing it this way provides a suitable name that contains the bank number.

The `.shadow` directive takes a location expression as its argument. In this case we want the address after the `enrom` instruction. It adds an annotation to the current section (the newly created `Code2` section fragment) that later will get emitted in the object file to inform the linker.

The linker will find this `.shadow` annotation and arrange so that the first instruction of this `Code2` section fragment will be placed at the address just after the `enrom2` instruction.

For this to work, you need to specify that section `Code2` should be in bank 2 in your linker rules file. Here is how such file can look:

```
(define memories
  '((memory Page1 (position independent) (bank 1) (address (#x0 . #xFFF))
            (section (FAT #x0) Code1 (Tail #xFD4))
            (checksum #xFFF hp41)
            (fill 0))
    (memory Page2 (position independent) (bank 2) (address (#x0 . #xFFF))
            (section (Bank2Header #x0) Code2 (Tail2 #xFF1))
            (checksum #xFFF hp41)
            (fill 0))))
```

It is possible to switch banks more than once and it is also possible to have more than one section fragment as shadows to the same (parent) section fragment. However, only one `.shadow` directive is allowed in a section fragment, meaning that you can only anchor yourself relative to one other section fragment.

Note: Switching banks like this will cause small code slices to be sprinkled out in the other bank. To get best allocation performance out of that bank, it is a good idea to also provide routines in it that can be

placed more freely, i.e. subroutines.

Note: As a general rule, it is better to have several smaller section fragments rather than few large ones, as the smaller section fragments give the linker more freedom to place code.

11.3 Exiting back

To exit back to the main bank from a secondary bank, you may want to go through some support routine you have defined that does necessary cleanup before exiting back to the operating system.

To exit from a secondary bank, you can provide an alternative entry point for such routine that just consists of a single `enrom1` instruction to enble bank 1 and place it so that it appears at the address location immediately before the actual routine in bank 1.

Here is one way of doing it with the `.shadow` directive:

```
              .section Code2
              .shadow PutXDrop-1
PutXDrop_rom2:
              enrom1
```

This just outputs a single `enrom1` instruction in bank 2, which gets located at the address before `PutXDrop` which is in bank 1.

From the secondary bank, we exit by jumping to `PutXDrop_rom2` (i.e. using a `goto`, `golong` or `golp` instruction). Inside the bank the `PutXDrop_rom2` routine looks like a single bank switch instruction located in the middle of nowhere. This instruction opens a trap door and we drop into bank 1 and end up in the routine that perform the exit.

Note: The `PutXDrop` routine does not need to be the first location of the section fragment it belongs to, it can be located anywhere in its section fragment.

Note: If you have more banks, you will need one routine like this for each bank you plan do drop into `PutXDrop` from.

As mentioned, to use it simply jump there using either a `goto`, `golong` or `golp` instruction from a bank 2 section:

```
someEntry:
            ...
            golp    PutXDrop_rom2
```

11.4 Calling between banks

Calling routines in another bank can be done, but it consumes one additional subroutine level. To do it, a stub routine can be used that temporarily switch the bank during the actual subroutine call, basically surrounding it by `switchBank` macros on both sides:

```
          .section Code2
findBufferUserFlags_rom2:
          c=regn  14
          rcr     12
          st=c
          switchBank 1
          gsbp    findBuffer
          switchBank 2
          cstex
          rtn
```

If you are in a page relocatable module, the `gsbp` instruction temporarily takes one additional subroutine level. Making calls like this quickly consumes the preciously small subroutine stack. In some cases it can still make sense to do it.

There may also be another routine like it in bank 1:

```
          .section Code
findBufferUserFlags:
          c=regn  14
          rcr     12
          st=c
          gsbp    findBuffer
          cstex
          rtn
```

In this case we get the same routine available in both bank 1 and 2 at the cost of some small code duplication. Stack usage will be the same in this case, and the called routine `findBuffer` is defined only once.

Note: In most cases, some planning on how to arrange code to minimize the need of bank switching, is often well spent time.

Note: Small code bduplication can be a better way. In such cases, macros can help in defining the actual code sequence only once.

CHAPTER 12

Barcode generator

The Barcode generator makes it possible to generate barcode for RPN programs. To use it you will need a Postscript level 2 (or later)[18] printer. To read the barcode into the HP-41, an HP 82153A Wand is required.

The Barcode generator is a command line tool named `barcode`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi barcode generator for Hewlett-Packard Nut version 5.18

Usage: barcode [--version] [--media MEDIA] [--private] [-I DIRECTORY] FILE
  use 'barcode --help' for detailed help

Available options:
  --version                Display version number
  --media MEDIA            Media, one of 'A4' or 'Letter', defaults to 'A4'
  --private                Private program barcode
  -I DIRECTORY             Include directory
  -h,--help                Show this help text
```

The input source file can either be an RPN source file or a raw binary image (using `.raw` file extension).

12.1 Language

The Barcode Generator will accept the same kind of source files as the RPN compiler, see RPN language. A source file that contains the `.local` directive is not accepted, see Invoking local MCODE instructions.

[18] The reason it does not work with Postscript level 1 is that a feature to compensate for accumulating errors is used. Today, level 1 printers are rare, level 2 was introduced in 1991.

12.2 Preprocessor

The Barcode generator uses a full featured C preprocessor to handle the input source file. The preprocessor provides the normal features you will find in a C preprocessor, the ability to include header files, macro expansions, conditional compilation and use of C style comments.

Wikipedia is a good place to look for an introduction with many examples on how to use the C preprocessor[17].

When used in the Barcode generator, the following macros are predefined:

Table 12.1: Predefined processor symbols

Preprocessor symbol	Description
`__CALYPSI_NUT__`	An integer that is 1 when the Nut target (including NEWT variant) is used.
`__CALYPSI_BARCODE__`	An integer that is 1 when the Barcode generator is used.
`__CALYPSI_RAM_USE__`	An integer that is 1 when the Barcode generator is used.

12.3 Branches and labels

Local branches are automatically selected depending on branch distance. The distance for a compiled small branch is slightly shorter in RAM compared to ROM, which is taken into account.

A fixed local `GTO` and `XEQ` instruction[19] in a program has reserved space in the instruction to store the offset to its destination label. The first time such instruction is executed, the branch offset is unknown and the calculator will search for the destination label. Once found, the offset is stored together with the instruction to speed up execution the next time the instruction is encountered.

However, some `GTO` instructions exists in a smaller (space saving form) that only can store short offsets. The RPN compiler will encode such branches in their long version if needed, to ensure that there is room to store the compiled offset to the destination label.

[17] http://en.wikipedia.org/wiki/C_preprocessor
[19] This excludes indirect branches and branches to alpha labels. Such branch instructions lack space to store the offset the the label.

CHAPTER 13

Clonix tool

Clonix is a series of programmable modules for the HP-41. A Clonix module contains a PIC-18 micro controller inside a HP-41 module shell which can be programmed outside the calculator. When plugged into the HP-41, it works by interacting directly with the HP-41 CPU over its bus, just like any other plug-in module.

The Clonix tool is an optional way to assemble HP-41 module images into configurations and have source code generated suitable for generating a Clonix module firmware.

The Clonix tool is a command line tool named `clonix`. Running it from the command line with the "–help" option will give a sign-on message and display accepted options:

```
Calypsi clonix tool for Hewlett-Packard Nut version 5.18

Usage: clonix [--version] [--module CLONIX-MODULE] [--start-page NUMBER]
              [--lock-pages HEX-DIGITS] [--calypsi-source]
              [-o|--output-file OUTPUT-FILE] [--signature-file NAME]
              [--lowercase-romimg-pages] [FILE...]
  use 'clonix --help' for detailed help

Available options:
  --version                Display version number
  --module CLONIX-MODULE   Clonix module, one of 'Clonix-41', 'Clonix-D' or
                           'NoV-64' (defaults to 'Clonix-41')
  --start-page NUMBER      Page to start loading at (defaults to '8')
  --lock-pages HEX-DIGITS  Pages that are to be left available (string of
                           individual hex digits)
  --calypsi-source         Generate experimental single source output
  -o,--output-file OUTPUT-FILE
                           Name of output file (valid only for experimental
                           single source output)
  --signature-file NAME    Name of the signature file (defaults to
                           'sgnature.asm')
  --lowercase-romimg-pages Use lowercase page numbers (defaults to upper case)
  -h,--help                Show this help text
```

Documentation of Clonix modules together with source files and tools for Windows are available at the Clonix web size[20]. If you are using Windows either directly, or by emulation, you will find everything you need here.

If you are on Linux or macOS, and you prefer to use your own platform, the Clonix tool is an option to the Windows tools.

In addition to the Clonix tool, you will need:

- A physical Clonix module (obviously).
- A suitable Clonix source file from the Clonix distribution, available from http://www.clonix41.org/.
- HP-41 module file images. A good place to look for these are http://www.hp41.org/.
- A PIC-18 assembler that is compatible with the Clonix source code. You can get MPLAB X from Microchip http://www.microchip.com/mplabx/ which contains `mpasmx`.
- A PIC-18 programmer wired to an adapter which attaches to the physical module. Such adapter can be obtained from the same source as the Clonix modules. Depending on which programmer you use, it may need some customization to fit the module adapter to the programmer. (The PICKit-2 mentioned below will need customization.)

One programmer that has been successfully used on macOS is PICkit-2. To use PICkit-2 from macOS or Linux, you need to download and build the command line tool `pk2cmd` from http://ww1.microchip.com/downloads/en/DeviceDoc/pk2cmdv1.20LinuxMacSource.tar.gz.

13.1 Differences

Comparing the Clonix tools to the `ClonixConfig` provided in the Clonix distribution, you may find there are situations where you prefer one over the other.

A brief comparison between the two is that `ClonixConfig` runs on Windows only, provides a GUI and support all modules from the Clonix and NoV families. It is an all-in-one solution that also will start the programming software (support is provided for two programmers).

The Clonix tool on the other hand runs on multiple platforms and currently supports a more limited set of modules. It provides more flexibility in generating double configurations (Clonix-D and NoV-64). It is also more of a building block, which means you may want to write some kind of script or build file to tie it all together.

`ClonixConfig` loads ROM images (lose module pages) and requires you to put them in the desired location manually. The Clonix tool loads module files, and will do the page layout for you.

One can say that `ClonixConfig` is more user friendly out of the box, but forces you to do the page layout manually every time. The Clonix tool is more of a building block for your own module creating scripts. Once created, you have a very powerful and easy way to keep your Clonix module (or modules) up to date with whatever configurations you want to use.

[20] http://www.clonix41.org/

13.2 Load process

To assemble modules into a Clonix module, the Clonix tools goes through a load process where one or several modules are processed. Basically, the module image files are loaded into a fictive memory system following the load constraints in the module file meta data.

There are several possible outcomes from loading modules. A success load process will output the page map:

```
$ clonix advantage.mod ppc.mod library4_Q.mod
4   Library#4 Rev.Q
8   Advantage Pac
9:1 Advantage Pac
9:2 Advantage Pac
a   PPC ROM
b   PPC ROM
(all Clonix pages filled up)
```

If you try to load too many module pages into you Clonix image, the Clonix tool will abort and tell you about it:

```
$ clonix smath44_3x3.mod library4_Q.mod
fatal error: trying to load too many module pages, Clonix-41 supports up to 6 ROM pages
Terminating due to errors
```

It is also possible that the load process went fine, but you are not happy with where the modules were loaded. There are a couple of parameters you can use to tune the load process. By default, loading will take place at page 8 (which is the first page available to any module plugged into the first port of the HP-41). Information in the module meta data may dictate that specific pages are to be used, or that it wants several pages allocated next to each other.

You can request load to start at another page using the `--start-page` option. You can further prevent pages from being used by specifying such pages to the `--lock-pages` option:

```
$ clonix --start-page=a --lock-pages e ppc.mod advantage.mod library4_Q.mod
4   Library#4 Rev.Q
a   PPC ROM
b   PPC ROM
c   Advantage Pac
d:1 Advantage Pac
d:2 Advantage Pac
(all Clonix pages filled up)
```

Here we start loading from page A (decimal 10) and ensure that we leave page E free (this is where a card reader would like to go). This went fine, but we realize that it would be good to leave page C free also so we can plug in a physical 4K application ROM there. As the PPC ROM wants two adjacent pages it will no longer fit, so we choose two 4K ROMs instead:

```
$ clonix --start-page=a --lock-pages cf advantage.mod tvm_1e.mod ccd-osx.mod library4_Q.mod
4   Library#4 Rev.Q
a   Advantage Pac
b:1 Advantage Pac
b:2 Advantage Pac
d   TIME VALUE of MONEY
e   CCD OS/X Module
(all Clonix pages filled up)
```

13.3 Building the image

The Clonix tool just emits various source files that need to be built together with the appropriate Clonix source file using `mpasmx`.

```
$ mpasmx ClonixLP.asm
```

If successful, you will now have a `CLonixLP.HEX` output file. You may also want to inspect the `ClonixLP.ERR` diagnostic file for any errors.

13.4 Programming a module

Programming a module requires a flash programmer. There are many possible choices, but one that has been tested on macOS is PICkit-2 from Microchip.

PICkit-2 uses USB, so just plug it in and try the following, assuming your output file is named `ClonixLP.HEX` and you have `pk2cmd` (and its device file) in the `PATH`:

```
$ pk2cmd -P -FClonixLP.HEX -A3.3 -M
Auto-Detect: Found part PIC18F252.

PICkit 2 Program Report
19-1-2016, 11:30:21
Device Type: PIC18F252

Program Succeeded.
```

13.5 Setting up a Makefile

If you want to simplify the process of maintaining your Clonix module (or even modules), you can set up a Makefile for building and maintaining a directory with your collection of HP-41 module images. If you later want to make minor changes to the list of modules, simply edit the commands in the Makefile accordingly:

```
MPASMX = /Applications/microchip/mplabx/v3.20/mpasmx/mpasmx
VPATH = /Users/me/modules41

# Build my Clonix-41 module
ClonixLP.HEX: ext-io.mod ttrkall.mod library4_Q.mod tvm_1e.mod ccd-osx.mod games.mod
        clonix --module clonix-41 --lock-pages cde $^
        $(MPASMX) ClonixLP.asm

clean:
        -rm -f ClonixLP.HEX ClonixLP.ERR ClonixLP.LST ClonixLP.O

programLP:
        pk2cmd -P -FClonixLP.HEX -A3.3 -M
```

As you add more modules to your growing collection and store them in your `modules41` directory, they become available to the set of modules you can choose between.

The example above relies on GNU Make, which provides the very useful `VPATH` directive. It basically makes it possible to set up a search path for input files, making it possible to just list places where you have module files in a colon separated list of directories.

CHAPTER 14

Other useful tools

In addition to using Calypsi to compile and link HP-41 MCODE and RPN programs, you will probably find some other tools useful. A text editor or IDE will be essential, while some other tools may not be essential, but they may make your life easier.

14.1 Editor

To write code you will also need a text editor and any decent text editor can be used. Another way is to use a customizable IDE. If you already have a favorite environment, just go ahead and use it.

I you do not have or are not happy with your current text editor/IDE environment, there are many to choose from.

You can go for old school powerful editors like Emacs or Vim. They may have a somewhat long learning curve, but you may find as many other have, that it is well worth it. If you are going to edit code and text, these tools are proven and designed for the purpose. They are keyboard oriented as they have their roots from before the mouse and graphical user interfaces become common. Fear not, by not having to lift your hand from the keyboard to reach the mouse, you will be able to work faster. You also have a huge set of very powerful editing commands at your fingertips. Aimed with these, you can edit fast.

If you want to go for a more IDE like experience, there are Eclipse and VS Code.

In fact, there are many editors on the market today, Atom, Sublime Text, just to name two. Consult your favorite search engine and surf ahead.

14.2 Build engine

IDEs and some editors allow building inside the tool. This makes the editing, build and fix loop very quick as you can do everything inside the same tool. This is very natural for IDEs, but powerful editors often provide it as well.

There are many build engines to choose from. One old trusted tool is `make` which takes a Makefile that describe the input files, the output files and provide rules for how put things together. Invoking `make` will result in running the minimal set of commands needed to bring the output up to date.

There are many alternatives today, like `cmake` and Ninja that can be worth looking at. Most IDEs provide some internal build engine and may allow external ones like `make` to be used from inside the IDE.

As an example, Emacs allows build inside the editor in a compile buffer. The easiest way is to create a simple Makefile with your project and then use the `make` command to build your project inside Emacs. A simple Makefile (suitable for GNU Make) can look as follows:

```
SRCS_16C = 16c.s mainframe.s
OBJS_16C = $(SRCS_16C:%.s=%.o)

all:    16c.mod

%.o: %.s
        asnut $<

16c.mod: $(OBJS_16C)
        lnnut $(OBJS_16C) Plugin4K.scm 16c.moddesc -o 16c.mod
```

14.3 Source code control

A source code control system is very useful to keep track of different revisions of your software. It is strongly recommended to use a source control system, as it will help you keep track of what you are doing over time. It gets even more useful if you collaborate with other people, or move your software between different computers.

If you are not using a source control system and feel it is too much hassle to set it up, take a look at tools like Git and Mercurial. Both Git and Mercurial are so easy to get started with and once you feel familiar with it, you will not hesitate to use it even for small projects. You do not need to set up any server and can move your project around on different computers and collaborate easily.

IDEs and powerful editors provide suitable functionality to work with tools like Git and Mercurial. You may find that you do perform some actions inside you IDE/Editor, but others outside on a command line.

The Emacs mode Magit[21] is especially useful. It provides a lot of functionality and control using Git inside the editor and will make your life a lot easier, should you go that route.

[21] https://magit.vc/manual/magit/#Top/

CHAPTER 15

Instruction sets

Both the HP and Jacobs-DeArras instruction sets are supported. The tools will use HP by default.

The reason for that there are two popular instruction sets is that HP did not originally publish its instruction set. Based on available fragments of information and a tremendous work by the user community, the instruction set was figured out.

Jim DeArras even went on to document the disassembled listings of the HP-41 firmware and when HP realized that the community was succeeding, they released the original listings.

However, the momentum was ongoing and we ended up with two popular instruction sets.

Why choose one over the other? The main reason to choose the HP instruction set is that it makes it easier to read the available source code listings. The main reason to choose the Jacobs-DeArras instruction set is that it is very popular.

No matter which you choose, you will most likely end up having a decent knowledge of the other instruction set as well. This section will take you through the main differences.

15.1 Non-local jumps

The 64K conditional jump and jump to subroutine instructions are named differently.

Table 15.1: Non-local jumps

HP	Jacobs-DeArras
`gosub`	`?ncxq`
`gsubc`	`?cxq`
`golong`	`?ncgo`
`golc`	`?cgo`

HP uses variants `gsubnc` and `golnc` which typically used after a test instruction (that sets carry) to say that we do care about the carry here not to be set. The `gosub` and `golong` variants are used when we know

the carry will be clear and the instruction is meant to be unconditional (the carry auto-resets after each instruction that does otherwise affect it).

15.2 Local branches

Short branches also have different names.

Table 15.2: Local branches

HP	Jacobs-DeArras
goto	jnc
goc	jc

Similar to non-local jumps, the gonc instruction is used after a test when we want to make it clear that we branch on carry not set.

15.3 Other jump and subroutine instructions

Table 15.3: Subroutine related

HP	Jacobs-DeArras
gotoc	gotoadr
stk=c	pushadr
c=stk	popadr
spopnd	xq>go
rtnnc	?ncrtn
rtnc	?crtn

15.4 Page relocatable jumps

Table 15.4: Page relative jumps

HP	Jacobs-DeArras
gsbp	rxq
golpc	rgo
gsb41c	rxq
gol41c	rgo

15.5 Pointer instructions

Table 15.5: Pointer related

HP	Jacobs-DeArras
selp	slctp
selq	slctq
decpt	r=r-1
incpt	r=r+1
pt=	r=
?pt=	?r=

15.6 Arithmetic instructions

These are mostly the same in both the instruction sets. There are a couple of differences as follows:

Table 15.6: Arithmetics

HP	Jacobs-DeArras
abex	a<>b
acex	a<>c
bcex	c<>b
c=-c	c=0-c

HP favors alphabetic order in the exchange instructions, while Jacobs-DeArras favors the more versatile register first.

The operand field to the arithmetic instructions shows more variation:

Table 15.7: Arithmetic operand fields

HP	Jacobs-DeArras
pt	@r
x	s&x
wpt	r<-
w	all
pq	p-q
xs	xs
m	m
s	ms

HP has the benefit of using only letters, which makes it easier to parse them as register fields in expressions (used in the debugger).

15.7 Flag instructions

This is a significant areas where the instruction sets differ. Also note that `st=0` exists in both instruction sets, but are used for different instructions.

Table 15.8: Flags

HP	Jacobs-DeArras
`st=0`	`crlf`
`s5=0` etc	`clrf 5`
`st=1`	`setf`
`s5=1` etc	`setf 5`
`?st=1`	`?fset`
`st=1?`	`?fset`
`?s5=1` etc	`?fset 5`
`clrst`	`st=0`
`cstex`	`c<>st`

15.8 Data memory

This is another area of significant differences in naming.

Table 15.9: Data memory

HP	Jacobs-DeArras
`dadd=c`	`ramslct`
`data=c`	`writedata`
`c=data`	`readdata`
`regn=c`	`writ`
`c=regn`	`read`

15.9 Keyboard

Table 15.10: Keyboard

HP	Jacobs-DeArras
`rstkb`	`clrkey`
`chkkb`	`?key`
`c=keys`	`c=key`
`gokeys`	`gtokey`

15.10 Miscellaneous

Here are the remaining instructions that differs.

Table 15.11: Miscellaneous

HP	Jacobs-DeArras
cxisa	fetch
lc	ld@r
selpf	selp
?flg=1	?fi
cnex	c<>n
cmex	c<>m
cgex	c<>g
f=sb	t=st
sb=f	st=t
fexsb	st<>t
?lld	?lowbat
clrabc	a=b=c=0
pfad=c	prphslct
disoff	dspoff
distog	dsptog

CHAPTER 16

HP instruction set

This chapter summarizes the instruction set without going into deep details. If you want more details, consult the NEWT manual[22] which describes the enhanced NEWT architecture. Instruction wise it is almost identical to the original Nut.

16.1 Non-local jump instructions

These instructions will jump to an absolute address within the 64K memory space. They are used for calling entries in the mainframe ROM and some fixed page located ROMs, typically in the address range `0000`–`7FFF`.

There are in reality 4 instructions and everyone will examine the carry to decide whether the jump should be taken or not. While `gosub` and `gsubnc` are the same instruction, the mnemonics serve two different purposes.

1. Unconditional non-local jumps using `gosub` or `golong` . The previous instruction should leave the carry flag cleared, and most instructions behave like this,
2. Test instruction followed by a non-local jump. In this case you want to use a variant that reflect that you are making use of the status of the carry flag, like `gsubnc` (even though it is the same instruction as `gosub`).

Table 16.1: Non-local jumps

Mnemonic	Operand	Description
`gosub`	expr	Absolute jump to subroutine
`gsubnc`	expr	Synonym for `gosub`
`gsubc`	expr	Absolute jump to subroutine on carry set
`golong`	expr	Absolute jump
`golnc`	expr	Synonym for `golong`
`golc`	expr	Absolute jump on carry set

[22] http://systemyde.com/pdf/newt.pdf

16.2 Local branches

Local branch instructions are position independent branches. They are rather short distance, -63 to 64 words.

There are in reality 2 instructions, both will examine the carry to decide whether the jump should be taken or not.

The `goto` instruction is intended to be used when you make a unconditional jump knowing that the carry will always be clear.

Table 16.2: Local jumps

Mnemonic	Operand	Description
`goto`	expr	Short branch (on carry clear)
`gonc`	expr	Synonym for `goto`
`goc`	expr	Short branch on carry set

16.3 Page relocatable jumps

These instructions take 3 words and are made up by a 2-word non-local `gosub` to a mainframe routine that reads the third word and uses that as part of the destination. They work inside a 4K ROM page.

As they are implemented with the `gosub` instruction, you need to ensure that the carry flag is not set when executing the instruction, otherwise the `gosub` will be skipped and you end up executing the third (page offset) word as an instruction, which is probably not what you intended.

The linker will pick the appropriate mainframe routine to use depending on the jump location and its destination.

Table 16.3: Page relative jumps

Mnemonic	Operand	Description
`gsbp`	expr	4K page relocatable jump to subroutine
`gsb41c`	expr	Synonym for `gsbp`
`golp`	expr	4K page relocatable jump
`gol41c`	expr	Synonym for `golp`

Note: The `gsbsam` and `golsam` that exist in the original HP instruction set are currently not supported. They are limited to have the call and destination in the same 1024-word sub-page, while the more general routines can reach anywhere in the 4K page. The linker will choose the 1024-word variant whenever possible, but it will not do active section placement to ensure it happens.

The benefits of the 1024-word variants are that it does not temporarily use a stack level (limiting it to three levels of the four available) and that it runs slightly faster.

If you want to ensure that the 1024-word variant is used, simply use a specific section name for each such 1024-word area and limit the address range accordingly in the linker description. Then use that section name for code that belongs together in this respect. This way you can ensure that the 1024-word variants are used when it matters, but still give the linker freedom to place things when it does not matter.

16.4 Other jump and subroutine instructions

Mnemonic	Description
`gotoc`	Jump to address in `C[6:3]`
`stk=c`	Push `C[6:3]` on stack
`c=stk`	Pop stack to `C[6:3]`
`spopnd`	Pop stack and discard value
`rtn`	Return from subroutine
`rtnnc`	Return from subroutine if carry clear
`rtnc`	Return from subroutine if carry set

16.5 Pointer instructions

There are two pointers, P and Q. They are used to describe which part of a 56-bit register to operate on, but they can also be used as counters. One pointer register is active at any given time.

Table 16.4: Pointer instructions

Mnemonic	Operand	Description
`selp`		Select pointer P
`selq`		Select pointer Q
`?p=q`		Test if P and Q pointers are equal
`decpt`		Decrement current pointer
`incpt`		Increment current pointer
`pt=`	expr	Set current pointer
`?pt=`	expr	Test if current pointer equals value

16.6 Arithmetic instructions

Arithmetic instructions take a field argument describing which part of the register to operate on.

Table 16.5: Fields

Field	Description
`pt`	At pointer
`x`	Exponent
`wpt`	Word up to pointer
`w`	Word
`pq`	Between pointers
`xs`	Exponent sign
`m`	Mantissa
`s`	Sign of mantissa

Table 16.6: Arithmetic instructions

Mnemonic	Operand	Description
a=0	field	Clear (part of) A register
b=0	field	Clear (part of) B register
c=0	field	Clear (part of) C register
abex	field	Exchange (part of) A and B registers
b=a	field	Move (part of) A to B register
acex	field	Exchange (part of) A and C registers
c=b	field	Move (part of) B to C register
bcex	field	Exchange (part of) B and C registers
a=c	field	Move (part of) C to A register
a=a+b	field	Add (part of) register
a=a+c	field	Add (part of) register
a=a+1	field	Increment (part of) register
a=a-b	field	Subtract (part of) register
a=a-1	field	Decrement (part of) register
a=a-c	field	Subtract (part of) register
c=c+c	field	Bit shift left (part of) C register
c=a+c	field	Add (part of) register
c=c+a	field	Synonym for previous instruction
c=c+1	field	Increment (part of) register
c=a-c	field	Subtract (part of) register
c=c-1	field	Decrement (part of) register
c=-c	field	One complement (part of) register
c=-c-1	field	Two complement (part of) register
?a#0	field	Test if (part of) A is non-zero
?b#0	field	Test if (part of) B is non-zero
?c#0	field	Test if (part of) C is non-zero
?a<c	field	Test if (part of) A is less than C
?a<b	field	Test if (part of) A is less than B
?a#c	field*	Test if (part of) A is not equal to C
asr	field*	Nibble shift (part of) A right
bsr	field*	Nibble shift (part of) B right
csr	field*	Nibble shift (part of) C right
asl	field	Nibble shift (part of) A left

The following two word pseudo instructions are supported:

Mnemonic	Operand	Description
b=c	field	Move (part of) C to B register
c=a	field	Move (part of) A to C register
a=b	field	Move (part of) B to A register

The following synonyms are supported:

Mnemonic	Operand	Description
cbex	field	Exchange (part of) B and C registers
baex	field	Exchange (part of) A and B registers
caex	field	Exchange (part of) A and C registers

16.7 Flag instructions

Mnemonic	Operand	Description
st=0	expr	Clear flag
sN=0		Clear flag, replace N with flag number 0 to 13
st=1	expr	Set flag
sN=1		Set flag, replace N with flag number 0 to 13
?st=1	expr	Test flag
st=1?	expr	Test flag
?sN=1		Test flag, replace N with flag number 0 to 13
clrst		Clear flags 0–7
st=c		Copy C[0:1] to flags 0–7
c=st		Copy flags 0–7 to C[0:1]
cstex		Exchange C[0:1] with flags 0–7

16.8 RAM memory instructions

RAM comes in units of 16 registers. You select a register to operate on and then read or write that particular register. It is also possible to access registers within the same 16 register unit,[23] but in practise, this is only useful for the very first 16 registers.

Table 16.7: RAM memory instructions

Mnemonic	Operand	Description
dadd=c		Select RAM memory register location
data=c		Write C register to selected RAM location
c=data		Read from selected RAM location to C register
regn=c	register	Write C register to given RAM register
c=regn	register	Read given RAM register to C register

[23] Reading register 0 is special because it does not exist. Instead that opcode is used for c=data, read the current selected register.

Table 16.8: RAM registers

Register number	Alternative
0	T
1	Z
2	Y
3	X
4	L
5	M
6	N
7	O
8	P
9	Q
10	+
10	|-
11	a
12	b
13	c
14	d
15	e

16.9 ROM memory instructions

Mnemonic	Description
cxisa	Read word from ROM memory
wmldl	Write to simulated MLDL ROM
enrom1	Enable ROM bank 1
enrom2	Enable ROM bank 2
enrom3	Enable ROM bank 3
enrom4	Enable ROM bank 4

16.10 Keyboard instructions

Mnemonic	Description
c=keys	Read keycode to C[4:3]
gokeys	Put keycode into low part of PC
rstkb	Reset keyboard
chkkb	Test if key is down

16.11 Miscellaneous instructions

Mnemonic	Operand	Description
nop		No operation
powoff		Stop CPU
lc	expr	Load 4-bit value to C register at pointer selected nibble
ldi	expr	Load 10-bits immediate to C.X
selpf	expr	Select smart peripheral
?flg=1	expr	Test if I/O flag is pulled
?fN=1	expr	Test if I/O flag is pulled, replace N with flag number 0 to 13
rcr	expr	Rotate C register right, both positive and negative values are accepted
sethex		Set binary arithmetic mode
setdec		Set BCD arithmetic mode
c=n		Move N to C register
n=c		Move C to N register
cnex		Exchange C and N registers
c=m		Move M to C register
m=c		Move C to M register
cmex		Exchange C and M registers
c=g		Move G to C register at pointer position
g=c		Move C to G register at pointer position
cgex		Exchange C and G registers at pointer position
f=sb		Move status byte to tone register
sb=f		Move tone register to status byte
fexsb		Exchange tone register and status byte
?lld		Low level detect, check for low battery
c=cora		Bitwise or A to C register
c=c&a		Bitwise and A to C register
clrabc		Clear A, B and C registers
pfad=c		Select peripheral

The following synonyms are supported:

Mnemonic	Operand	Description
ncex		Exchange C and N registers
mcex		Exchange C and M registers

16.12 LCD instructions

I/O instructions like the ones for the LCD are shared with the RAM access instructions. The assembler provides synonym instructions for the LCD as follows:

Table 16.9: LCD instructions

Mnemonic	Description
disoff	Turn display off
distog	Toggle display on or off
srlda	Shift right long, load A

continues on next page

Table 16.9 – continued from previous page

Mnemonic	Description
srldb	Shift right long, load B
srldc	Shift right long, load C
srldab	Shift right long, load AB
srlabc	Shift right long, load ABC
slldab	Shift left long, load AB
sllabc	Shift left long, load ABC
srsda	Shift right short, load A
srsdb	Shift right short, load B
srsdc	Shift right short, load C
slsda	Shift left short, load A
slsdb	Shift left short, load B
srsdab	Shift right short, load AB
slsdab	Shift left short, load AB
srsabc	Shift right short, load ABC
slsabc	Shift left short, load ABC
wrten	Write annunciators
flda	Read left long A
fldb	Read left long B
fldc	Read left long C
fldab	Read left long AB
flabc	Read left long ABC
readen	Read annunciators
frsda	Read right short A
frsdb	Read right short B
frsdc	Read right short C
flsda	Read left short A
flsdb	Read left short B
flsdc	Read left short C
frsdab	Read right short AB
flsdab	Read left short AB
rabcr	Read right short ABC
rabcl	Read left short ABC
frsabc	Read right short ABC
flsabc	Read left short ABC

Note: The register names A, B and C used here have nothing to do with the CPU registers with the same name.

16.13 Timer chip instructions

Mnemonic	Description
`wrtime`	Write time
`wdtime`	Write time and hold
`wralm`	Write alarm
`wrsts`	Write status
`wrscr`	Write scratch
`wsint`	Write interval timer
`stpint`	Stop interval timer
`dswkup`	Disable test mode
`enwkup`	Enable test mode
`dsalm`	Disable alarm
`enalm`	Enable alarm
`stopc`	Stop clock
`startc`	Start clock
`pt=b`	Select timer B
`pt=a`	Select timer A
`rdtime`	Read clock
`rctime`	Read clock and correct
`rdalm`	Read alarm
`rdsts`	Read status
`rdscr`	Read scratch
`rdint`	Read interval timer
`alarm?`	Test if alarm
`fatl?`	Test timer access failure

16.14 Card reader instructions

The card reader (peripheral `0xFC`). PPC calculator journal V11N6 page 18–20 has an article about the programming internals of the card reader.

Table 16.10: Card reader instructions

Mnemonic	Description
`stwrit`	Start write cycle
`enwrit`	End write cycle
`stread`	Start read cycle
`enread`	End read cycle
`crdwpf`	Test write protect
`crdohf`	Fetch card over head flag
`crdinf`	Test if card inserted
`tstbuf`	Test card read/write buffer
`trpcrd`	Set card trip flag
`tclcrd`	Test and clear card trip flag
`crdflg`	Fetch card reader external flag
`crdr?`	Set carry on card reader flag (peripheral flag 1)

16.15 Wand instructions

Table 16.11: Wand instructions

Mnemonic	Description
`wndb?`	Set carry when data in Wand buffer (peripheral flag 2)

16.16 HP-IL instructions

Mnemonic	Operand	Description
`orav?`		Test output register available
`frav?`		Test if frame is available
`ifcr?`		Test if IFC command received
`frns?`		Test frame received no as sent
`srqr?`		Test for service request received
`hpil=c`	IL-register	**Write to HP-IL register *n*** where IL register is 0-7
`c=hpil`	IL-register	**Read from HP-IL register *n*** where n is 0-7
`hpl=c`	IL-register	**Write 8-bit *expr* to HP-IL register *n*** where n is 0-7
`ch=`	expr	Specifies char used in hpl=c

16.17 Peripheral instructions

Mnemonic	Operand	Description
`?pfset`	0-15	Set carry if peripheral flag N is set
`printc`		Add `C[1:0]` to print buffer
`rdptrn`		Read status
`rdptrr`		
`rtncpu`		

16.18 Hepax instructions

Mnemonic	Operand	Description
`romblk`		Move ROM block
`wptog`		Toggle write protection

16.19 NEWT specific instructions

NEWT specific instructions are recognized if you specify the command line option `--core newt`.

Mnemonic	Operand	Description
wcmd		Write command

CHAPTER 17

Jacobs-DeArras instruction set

This chapter summarizes the Jacobs-DeArras instruction set without going into deep details.

17.1 Non-local jump instructions

These instructions will jump to an absolute address within the 64K memory space. They are used for calling entries in the mainframe ROM and some fixed page located ROMs, typically in the address range `0000`–`7FFF`.

Table 17.1: Non-local jumps

Mnemonic	Operand	Description
?ncxq	expr	Absolute jump to subroutine
?cxq	expr	Absolute jump to subroutine on carry set
?ncgo	expr	Absolute jump
?cgo	expr	Absolute jump on carry set

17.2 Local branches

Local branch instructions are position independent branches. They are rather short distance, -63 to 64 words.

Table 17.2: Local jumps

Mnemonic	Operand	Description
jnc	expr	Synonym for `goto`
jc	expr	Short branch on carry set

17.3 Page relocatable jumps

These instructions take 3 words and are made up by a 2-word non-local `?ncxq` to a mainframe routine that reads the third word and uses that as part of the destination. They work inside a 4K ROM page.

As they are implemented with the `?ncxq` instruction, you need to ensure that the carry flag is not set when executing the instruction, otherwise the `?ncxq` will be skipped and you end up executing the third (page offset) word as an instruction, which is probably not what you intended.

The linker will pick the appropriate mainframe routine to use depending on the jump location and its destination.

Table 17.3: Page relative jumps

Mnemonic	Operand	Description
`rxq`	expr	4K page relocatable jump to subroutine
`rgo`	expr	4K page relocatable jump

17.4 Other jump and subroutine instructions

Mnemonic	Description
`gotoadr`	Jump to address in `C[6:3]`
`pushadr`	Push `C[6:3]` on stack
`popadr`	Pop stack to `C[6:3]`
`xq>go`	Pop stack and discard value
`rtn`	Return from subroutine
`?ncrtn`	Return from subroutine if carry clear
`?crtn`	Return from subroutine if carry set

17.5 Pointer instructions

There are two pointers, P and Q. They are used to describe which part of a 56-bit register to operate on, but they can also be used as counters. One pointer register is active at any given time.

Table 17.4: Pointer instructions

Mnemonic	Operand	Description
`slctp`		Select pointer P
`slctq`		Select pointer Q
`?p=q`		Test if P and Q pointers are equal
`r=r-1`		Decrement current pointer
`r=r+1`		Increment current pointer
`r=`	expr	Set current pointer
`?r=`	expr	Test if current pointer equals value

17.6 Arithmetic instructions

Arithmetic instructions take a field argument describing which part of the register to operate on.

Table 17.5: Fields

Field	Description
@r	At pointer
s&x	Exponent
r<-	Word up to pointer
all	Word
p-q	Between pointers
xs	Exponent sign
m	Mantissa
ms	Sign of mantissa

Table 17.6: Arithmetic instructions

Mnemonic	Operand	Description
a=0	field	Clear (part of) A register
b=0	field	Clear (part of) B register
c=0	field	Clear (part of) C register
a<>b	field	Exchange (part of) A and B registers
b=a	field	Move (part of) A to B register
a<>c	field	Exchange (part of) A and C registers
c=b	field	Move (part of) B to C register
c<>b	field	Exchange (part of) B and C registers
a=c	field	Move (part of) C to A register
a=a+b	field	Add (part of) register
a=a+c	field	Add (part of) register
a=a+1	field	Increment (part of) register
a=a-b	field	Subtract (part of) register
a=a-1	field	Decrement (part of) register
a=a-c	field	Subtract (part of) register
c=c+c	field	Bit shift left (part of) C register
c=a+c	field	Add (part of) register
c=c+a	field	Synonym for previous instruction
c=c+1	field	Increment (part of) register
c=a-c	field	Subtract (part of) register
c=c-1	field	Decrement (part of) register
c=0-c	field	One complement (part of) register
c=-c-1	field	Two complement (part of) register
?a#0	field	Test if (part of) A is non-zero
?b#0	field	Test if (part of) B is non-zero
?c#0	field	Test if (part of) C is non-zero
?a<c	field	Test if (part of) A is less than C
?a<b	field	Test if (part of) A is less than B
?a#c	field*	Test if (part of) A is not equal to C
rshfa	field*	Nibble shift (part of) A right
rshfb	field*	Nibble shift (part of) B right
rshfc	field*	Nibble shift (part of) C right

continues on next page

Table 17.6 – continued from previous page

Mnemonic	Operand	Description
`lshfa`	field	Nibble shift (part of) `A` left

17.7 Flag instructions

Mnemonic	Operand	Description
`clrf`	expr	Clear flag
`setf`	expr	Set flag
`?fset`	expr	Test flag
`st=0`		Clear flags `0`–`7`
`st=c`		Copy `C[0:1]` to flags `0`–`7`
`c=st`		Copy flags `0`–`7` to `C[0:1]`
`c<>st`		Exchange `C[0:1]` with flags `0`–`7`

17.8 RAM memory instructions

RAM comes in units of 16 registers. You select a register to operate on and then read or write that particular register. It is also possible to access registers within the same 16 register unit,[24] but in practise, this is only useful for the very first 16 registers.

Table 17.7: RAM memory instructions

Mnemonic	Operand	Description
`ramslct`		Select RAM memory register location
`writedata`		Write `C` register to selected RAM location
`readdata`		Read from selected RAM location to `C` register
`writ`	register	Write `C` register to given RAM register
`read`	register	Read given RAM register to `C` register

[24] Reading register `0` is special because it does not exist. Instead that opcode is used for `c=data`, read the current selected register.

Table 17.8: RAM registers

Register number	Alternative
0	T
1	Z
2	Y
3	X
4	L
5	M
6	N
7	O
8	P
9	Q
10	+
10	|-
11	a
12	b
13	c
14	d
15	e

17.9 ROM memory instructions

Mnemonic	Description
fetch	Read word from ROM memory
wmldl	Write to simulated MLDL ROM
enrom1	Enable ROM bank 1
enrom2	Enable ROM bank 2
enrom3	Enable ROM bank 3
enrom4	Enable ROM bank 4

17.10 Keyboard instructions

Mnemonic	Description
c=key	Read keycode to C[4:3]
gtokey	Put keycode into low part of PC
clrkey	Reset keyboard
?key	Test if key is down

17.11 Miscellaneous instructions

Mnemonic	Operand	Description
nop		No operation
powoff		Stop CPU
ld@r	expr	Load 4-bit value to C register at pointer selected nibble
ldi	expr	Load 10-bits immediate to C.X
selp	expr	Select smart peripheral
?fi	expr	Test if I/O flag is pulled
rcr	expr	Rotate C register right, both positive and negative values are accepted
sethex		Set binary arithmetic mode
setdec		Set BCD arithmetic mode
c=n		Move N to C register
n=c		Move C to N register
c<>n		Exchange C and N registers
c=m		Move M to C register
m=c		Move C to M register
c<>m		Exchange C and M registers
c=g		Move G to C register at pointer position
g=c		Move C to G register at pointer position
c<>g		Exchange C and G registers at pointer position
t=st		Move status byte to tone register
st=t		Move tone register to status byte
st<>t		Exchange tone register and status byte
?lowbat		Low level detect, check for low battery
c=cora		Bitwise or A to C register
c=c&a		Bitwise and A to C register
clrabc		Clear A, B and C registers
prphslct		Select peripheral

17.12 Hepax instructions

Mnemonic	Operand	Description
romblk		Move ROM block
wptog		Toggle write protection

17.13 NEWT specific instructions

NEWT specific instructions are recognized if you specify the command line option `--core newt`.

Mnemonic	Operand	Description
wcmd		Write command

17.14 Peripheral instructions

Refer to the sections describing Peripheral instructions in the HP instructions chapter for the peripheral specific instructions recognized, as they are the same in both instructions sets (they were never defined for the Jacobs-DeArras instruction set).

CHAPTER 18

4K page layout

The HP-41 assumes a certain layout of 4K ROM pages in order to identify the functions it contains and how they are to be used. At the end of each 4K page there is also some poll vectors that allow the code to hook in and handle certain events. A stripped down 4K ROM page can look as follows:

```
              .con  4               ; XROM number
              .con  3               ; number of entries

              ;; Function address table
              .fat  entry_Header
              .fat  entry_Code
              .fat  entry_Decode
              .con  0,0             ; marks end of FAT

;;; Code goes here
              .name "DECODE"
entry_Decode: ;; code goes here

              .name "CODE"
entry_Code:   ;; code goes here

              .name "-EXAMPLE 1"    ; header
entry_Header: rtn

;;; Poll vectors
              .section PollVectors
              .con  0               ; Pause
              .con  0               ; Running
              .con  0               ; Wake w/o key
              .con  0               ; Powoff
              .con  0               ; I/O
              .con  0               ; Deep wake-up
              .con  0               ; Memory lost
              .text "A1XE"          ; Identifier EX-1A
```

(continues on next page)

(continued from previous page)

```
;;; Checksum will go here, but it is filled in by the linker
;;; so we leave it out.
```

The first word is the XROM number which should range from 1–31. You can pick any number, but should avoid those that your plug-in module are expected to co-exist with.[25]

The second word is the number of functions in the function address table. When looking up a given function, this number is consulted to see if the given instruction number exists in this module.

The function address table is easily populated using the `.fat` directive. In reality there are two words for each entry. The function address table must be ended with two `0` words.

After this header you enter the space where we can put our own code.

At offset `FF4` at the end comes the poll vectors. If you do not use them, or do not know what they are good for, just fill them out with `0` to indicate that they are unimplemented.

A four word module identifier, in reverse order, gives an indication of which module this is. You typically pick two letters related to the name of your module and a 2 digit revision number. The identifier in the example code is `EX-1A`, for "Example revision 1A".

The very final word of the 4K page is a checksum. This checksum can be calculated and filled in by the linker, so we should not specify it in the source file.

18.1 FAT order

The automatic generation of FAT contributions by the RPN compiler makes it easy to populate a complete FAT. However, its automatic nature also means that if you add or remove global alpha labels, the FAT entries may change which may result in that the same function is assigned a different XROM number.

As XROM functions in a module may be used in RPN programs, such programs will have a dependency not only on the module, but also on the order of the entries in the FAT. Thus, if you add or remove entries, you may want to take full control over the FAT entries in order to keep it consistent. This can only be done by defining it manually in an MCODE source file.

To define a FAT manually, disable generation of FAT section fragments from the RPN compiler using the `--no-fat` option. Then you will need a full FAT defined in MCODE, it may look something like:

```
              .section Header
              .extern readRom16, writeRom16
              .extern FUPDATE, HFUPDAT

              .public `FAT entry: RDROM16`

XROMno:       .equ    14

              .con    XROMno          ; XROM number
              .con    .fatsize FatEnd ; number of entry points
FatStart:
              .fat    Header          ; ROM header
`FAT entry: RDROM16`:
              .fat    readRom16
              .fat    writeRom16
              .fatrpn FUPDATE
```

(continues on next page)

[25] Unfortunately, the number range is not all that wide and the number of modules on the market severely outnumbers it.

(continued from previous page)

```
            .fatrpn HFUPDAT

FatEnd:     .con    0,0
```

In this table there are three MCODE functions and two RPN programs. The entries that are internally used by RPN programs in the module need to have the special `` `FAT entry: XXXX` `` label defined at its FAT entry. In this case only the MCODE program `RDROM16` is used internally.

The entries that exist in different source files must be imported using the `.extern` directive. In MCODE you are quite free to choose the name, but in RPN you get the name of the global alpha label, which is exported by the RPN compiler for this purpose.

CHAPTER 19

Character sets

Text files are assumed to be in Unicode and encoded in UTF-8. Generated list files are also encoded in UTF-8.

19.1 HP-41 character set

The HP-41 uses a character set that resembles ASCII with some additional characters which cannot be represented in ASCII. When using language constructs that are intended for the HP-41 alpha characters, some Unicode characters are converted to the corresponding HP-41 character. The following table lists those characters with the corresponding HP-41 character code and Unicode code point in hexadecimal.

Table 19.1: Unicode table

Symbol	Name	HP-41	Unicode
μ	micro	0C	03BC
∠	angle	0D	2280
≠	not-equal	1D	2260
Σ	sigma	7E	03A3
⊢	append	7F	251C

In addition to these, there are some extra characters that do not have any resemblance in Unicode, most notably the hangman style characters. They need to entered using numeric constants, either using the `.con` directive or by using numeric escape sequences in strings.

19.2 HP-41 display characters

The HP-41 display character set uses a different encoding. This character set is used at the MCODE level when dealing with the display directly.

19.2.1 Automatic conversion

In certain situations you need to enter literal strings in the display character set. Two assembler directives `.messl` and `.name` are provided for this. They take a string argument and convert the string in a suitable way for their respective use case.

`.messl` is intended to be used when calling the mainframe routine `MESSL` which sends inline literal text directly to the display.

`.name` helps you entering the name of a MCODE instruction. In addition to converting the string to display characters, the order of the characters are also reversed.

Both `.messl` and `.name` also set a bit in the final character to mark the end of the string.

19.2.2 Display character encoding

Display characters are encoded in 9 bits which can be visualized as `UPPNNNNNN`, where `PP` are punctuation bits between characters. The `U` bit can be seen as the next higher bit from `NNNNNN` which gives access to additional characters, i.e. lower case letters such as a–e.

19.2.3 Additinal halfnut display characters

In the middle of the HP-41 production run, the hardware was changed to make the HP-41 cheaper to produce and sell at a more competitive price. A slightly different display with additional characters was introduced at the same time. This display can be identified by its rounded corners. These later machines are normally referred to as Halfnuts.

While the HP-41 mainframe code was not changed to take advantage of the new display, the additional characters can be used from MCODE. However, before doing so, consider that your program will not display properly on earlier HP-41 calculators.

The extension of the character set is done by taking full advantage of the `U` bit described above. On the original display, only 16 additional characters are provided, while on a Halfnut 64 additional characters exist, for a full set of lower case letters.

19.2.4 Named MCODE instructions

Extended characters can be used in MCODE instruction names and they will display properly on a Halfnut HP-41, i.e. in `CAT 2`, when to assigned to a key and in program steps.[26]

However, trying to key such instruction name on an HP-41 will not work, which makes it hard to actually use it. You may find some use for this feature when constructing a ROM header, or perhaps for some internal instructions not normally intended to be executed by the user.

[26] This works because the bits are arranged slightly different way compared to how they are normally encoded for display characters. As a side-effect, no punctuation characters can be used in MCODE instruction names.

19.2.5 Text messages

Unfortunately, the mainframe routine `MESSL` which sends a string to the display does not perform the same kind of re-arrangement as is done when dealing with MCODE names. Since the string end is marked by setting the bit above `U`, they both end up in the same nibble (`U` is the ninth bit). As a result, a display character with the `U` bit set can only appear as the final character in a string intended for `MESSL`.

If you put such character at another position, your message string will terminate early and the HP-41 will try to execute the rest of your string as MCODE instructions, which is probably not what you intended.

CHAPTER 20

Examples

Now lets look at some examples on how to use the tools to build actual modules. HP-41 modules come in many variants and Calypsi will allow you to make virtually any kind of software module using RPN or MCODE, allowing multi-page layouts, with or without page banking.

20.1 Hello world

The first example is a very basic MCODE module that contains a simple instruction `HELLO` that will display `HELLO WORLD`. The purpose is to put together minimal, but complete module, following the module layout as described in 4K page layout.

While this example is simple, it is useful for getting familiar with the tools and how to use the resulting module on your HP-41 or HP-41 emulator.

The example can be found relative to the installation directory in `example/HelloWorld`. In order to build it you will need to, a) copy the example elsewhere; b) ensure that you have GNU Make installed; and c) have your path environment variable set up so that GNU Make and the Calypsi can be found.

When properly set up, you should be able to build the example by just saying `make` from the command line.

```
$ make
../../../../../../bin/asnut HelloWorld.s
../../../../../../bin/lnnut HelloWorld.o plugin4k.scm hello.moddesc -o hello.mod
```

20.1.1 Trying it out

The result will be a module file `hello.mod` that you need to plug in to your HP-41 or load in your HP-41 emulator. If you are using a real HP-41, you need either an MLDL, Clonix module or something similar, and some means to download the module to it. If you are using an emulator, it needs to support the module file format to be able to load the module.

Once installed in the HP-41, try the command `HELLO` which should display `HELLO WORLD`. You can also see the module in catalog 2 (using `CAT 2`).

20.2 RPN games module

In this example we will start with a games module that makes use of a couple of nice Poker and Roulette games written by Jean-Marc Baillard. The intention here is to use them as an example how to make a module (not document these programs in detail).

The example can be found in the `example/games` directory in the Calypsi installation.

Originally, these programs were obtained from internet by using copy and paste into a source file. Since they use features on the HP-41CX, an `.import` directive stating this has been added at the beginning. When importing source code, you should check which convention has been used to represent append alpha strings, as there are many variants in use. This is because there is no natural way to do this in ASCII as the HP-41 uses character 127, which is DEL in ASCII.

In the original sources for these programs, character code 126 is used as append character. This is the tilde in ASCII, but actually Σ on the HP-41. The RPN Compiler does not accept this and those strings need to be changed, i.e. change `"~NTH"` to `>"NTH"`.

20.2.1 The project

This example uses GNU Make which you need to have installed. The Makefile contains the following:

```
BIN=../../../../../../bin
RPN_SRCS = poker.rpn roulette.rpn
RPN_RAWS =
ASM_SRCS = header.s PokerSupport.s

OBJS = $(RPN_SRCS:%.rpn=%.o) $(RPN_RAWS:%.raw=%.o) $(ASM_SRCS:%.s=%.o)

MOD = games.mod

all:        $(MOD)

clean:
        -rm -f $(OBJS) *.lst $(MOD)

%.o: %.rpn
        $(BIN)/rpncomp -l $<

%.o: %.raw
        $(BIN)/rpncomp -l $<

%.o: %.s
        $(BIN)/asnut $<
```

(continues on next page)

(continued from previous page)

```
$(MOD): $(OBJS) Plugin4K.scm games.moddesc Makefile
        $(BIN)/lnnut -l $(OBJS) Plugin4K.scm games.moddesc -o $(MOD)
```

As can be seen, it is fairly simple to add source files to the project by adding files to the variables at the top. There are two RPN source files, no raw RPN files and two assembly source files.

20.2.2 The assembly files

The `PokerSupport.s` file contains some special MCODE routines used by the poker program to speed it up. These routines are imported into the `poker.rpn` program at the top:

```
.import 41cx
.local CARD, MV, TRI

 01  LBL "POKER+"
 02  10
 ...
```

The second one is `header.s` which defines everything related to the module layout:

```
;;; ************************************************************************
;;;
;;; Module header
;;;
;;; ************************************************************************

               .section HEADER
               .con  21              ; XROM number
               .con    .fatsize fatend
               .fat    header

               .section FATEND
;;; End marker for function address table
fatend:        .con  0,0

               .section Code
;;; Make an empty name function for the module to show up in CAT 2
               .name "-GAMES EX 1A"  ; The name of the module
header:        rtn

               .section TAIL
;;; Tail of the module with empty poll points and module ID
               .con  0,0,0,0,0,0,0
               .text "A1XG"          ; GX1A
```

This file can be used as a starting point for other projects simply by changing the XROM identity, the ROM header function and the module ID at the end.

The size of the FAT is automatically calculated thanks to the .fatsize relocation operator being used to point to the end marker of the FAT.

20.2.3 Linker rules

The linker rule file lists the order of the sections and ensures that the `TAIL` section ends up at the proper starting position (hex `FF4`):

```
(define memories
  '((memory Page (address (#x0 . #xFFF)) (position independent)
            (section (HEADER FAT FATEND #x0) Code RPN (TAIL #xFF4))
            (checksum #xFFF hp41)
            (fill 0))))
```

The `TAIL` section contains the poll vectors and module identity which shall go to the end of the module page.

20.2.4 Trying it out

The result will be a module file `games.mod` that you need to plugin to your HP-41 or load in you HP-41 emulator as described in the previous example, see Trying it out.

Once installed in the HP-41, you can try the games, but you are probably best advised to go to Jean-Marc Baillard's web site[27] and read the instructions.

20.3 Modules on the internet

Here is a collection of real projects that use Calypsi.

20.3.1 Amateur astronomy

Amateur astronomy ROM[28] contains a collection of telescope and observational utilities.

20.3.2 CLILUP

CLILUP[29] is an update module for HP-41CL using HP-IL. It is a mixed module using both MCODE and RPN.

20.3.3 Data and statistical analysis

Data and statistical analysis ROM[30] is a comprehensive statistical analysis of data sets on the HP-41 calculator.

[27] http://hp41programs.yolasite.com
[28] https://github.com/isene/hp-41_AMASTRO.ROM
[29] https://github.com/hth313/CLILUP
[30] https://github.com/isene/hp-41_DSTAT.ROM

20.3.4 EquationLibrary

EquationLibrary[31] implements an equation library on top of the Formula Evaluation module. It is mostly written in RPN with some MCODE.

20.3.5 Geir ROM

Geir ROM[32] is a collection of the most essential programs, utilities and tools as used by Geir Isene. This contains a mixture of useful RPN and MCODE programs.

20.3.6 Isene ROM

Isene ROM[33] contains the most useful of Geir Isene's RPN programs. Ultimate Alarm Clock, the PDA programs NOTES and REM as well as CRYPT to encrypt and decrypt ASCII files and other neat stuff. Makes use of HEPAX and XM files.

20.3.7 Ladybug

Ladybug[34] is a banked module that makes use of various techniques to switch banks on the fly. It converts the HP-41 into an integer mode calculator, similar to the HP-16C.

20.3.8 Lib41

Lib41[35] is a library (not a module) with routines that can be incorporated into your own module.

20.3.9 OS4

OS4[36] is an HP-41 operating system extension module that uses page 4.

[31] https://github.com/mafleming/EquationLibrary
[32] https://github.com/isene/hp-41_GEIR.ROM
[33] https://github.com/isene/hp-41_isene-rom
[34] https://github.com/hth313/ladybug
[35] https://github.com/hth313/lib41
[36] https://github.com/hth313/OS4

Index

Non-alphabetical

.. index:: section
    fragments, 51
4K page layout, 97
41CL
    speed annotation, 57
__CALYPSI_ASM__ (predefined symbol), 12
__CALYPSI_BARCODE__ (predefined symbol), 66
__CALYPSI_MODULE_USE__ (predefined symbol), 32
__CALYPSI_NUT__ (predefined symbol), 12, 32, 66
__CALYPSI_RAM_USE__ (predefined symbol), 32, 66
__CALYPSI_RPN_COMPILER__ (predefined symbol), 32

A

alignment, 18
allocated memory, 51
Arch Linux
    installation, 4
    uninstalling, 5
archive, 54
arithmetic instructions, 81, 92

B

back quoted symbols, 12
bank switching, 58, 60
banked placement, 22
banking, 50
banks, 62
barcode generator, 63
books, 7
branch length
    RPN, 35, 66
BSS section, 17

C

calling between, 62
card reader, 87
characters
    display, 104
    HP-41, 103
checksum, 49, 100
class two instructions, 81, 92
Clonix, 107
Clonix tool, 66
comments, 12
    RPN, 33
conditional assembly, 12
constants, 14

D

Data section, 17
Debian
    installation, 3
    uninstalling, 4
directive
    .con, 19
    .equ, 14, 15, 20
    .equlab, 14, 15
    .extern, 14, 20
    .fat, 14, 18
    .macro, 24
    .messl, 14, 19, 20
    .name, 14, 18, 19
    .public, 14, 20
    run-time model, 21
    .section, 13, 14, 17
    .shadow, 22, 60
    .suppress, 14, 22
    .text, 14, 19
directives, 14
display
    characters, 104
distribution
    installation, 1

E

ELF, 26
.equ directive, 14, 15, 20
.equlab directive, 14, 15
example
    games, 108
    hello world, 107
    rpn, 108
.extern directive, 14, 20

F

FAT, 18, 51, 100
    directives, 18
.fat directive, 14, 18
file
    extensions, 9
file extensions, 9
file inclusion, 12
fill word, 49
flag instructions, 82, 94
function address table, 18, 51, 100

G

global alpha labels, 35

H

Halfnut, 104
Hepax, 88, 96
HP
    instructions, 10, 72, 77
HP-41
    characters, 103
    Halfnut, 104
HP-41CX instructions, 37
HP-IL, 88

I

import
    functions, 37, 41
    instructions, 37, 41
include files, 12
installation, 1
    Arch Linux, 4
    Debian, 3
    Linux, 3, 4
    macOS, 3
    Manjaro, 4
    multiple, 5
    multiple versions, 5
    Ubuntu, 3
    Windows, 5
instructions
    card reader, 87
    class two, 81, 92
    display, 85
    flags, 82, 94
    Hepax, 88, 96
    HP, 10, 72, 77
    HP-41CX, 37
    HP-IL, 88
    indirect jumps, 80, 92
    Jacobs-DeArras, 10, 72, 89
    keyboard, 84, 95
    LCD, 85
    local MCODE, 38
    NEWT, 88, 96
    pointer, 81, 92
    RAM memory, 83, 94
    return, 80, 92
    ROM memory, 84, 95
    timer, 86
    wand, 88

J

Jacobs-DeArras
    instructions, 10, 72, 89
jumps
    local, 79, 91
    non-local, 79, 91
    page relocatable, 80, 91

K

keyboard instructions, 84, 95

L

labels, 12
    global, 20
    global alpha, 35
    local, 14
    location counter, 14
    RPN, 35
    weak, 20
language
    RPN, 32
LCD, 19, 85
librarian, 54
line numbers
    RPN, 33
linker, 46
linker rules, 49
Linux
    installation, 3, 4
    uninstalling, 4, 5
list files, 9
    assembler, 22
    linker, 51
local branch instructions, 79, 91
local jumps, 79, 91
local labels, 14
local labels inside macro, 25
location counter, 14

M

macOS
    installation, 3
    uninstalling, 3
.macro, 24
macro, 24
    local label, 25
macro language, 24
mainframe
    calling, 79, 91
    entry points, 16
    LCD messages, 20
Manjaro
    installation, 4
    uninstalling, 5
MCODE, 1, 10
memory, 48
    address range, 48, 49
    fill, 49
    name, 48
    rule linking, 49
    size, 51
.messl directive, 14, 20
MLDL, 107
module
    description file, 41
    layout, 97
module tool, 41

N

.name directive, 14, 18, 19
    prompt bits, 19
naming directive, 19
NEWT, 57
    instructions, 88, 96
    speed annotation, 57
non-local jumps, 79, 91
numbers, 14
    NULL separation, 34

O

object file format, 26

operator
    .sectionEnd, 28
    .sectionSize, 28
    .sectionStart, 28
operators, 13
    relocation, 27
    section, 28
order, 100

P

page relocatable jumps, 80, 91
placing sections, 22
plug-in modules, 37
pointer instructions, 81, 92
poll vectors, 100
postfix operands
    for XROMs, 40
predefined symbol
    __CALYPSI_ASM__, 12
    __CALYPSI_BARCODE__, 66
    __CALYPSI_MODULE_USE__, 32
    __CALYPSI_NUT__, 12, 32, 66
    __CALYPSI_RAM_USE__, 32, 66
    __CALYPSI_RPN_COMPILER__, 32
prompt bits, 19
.public directive, 14, 20

Q

quoted symbols, 12

R

RAM memory instructions, 83, 94
RAM registers, 83, 94
Read only data section, 17
references, 7
relocation operators, 27
relocations, 27
required symbols, 21
return instructions, 80, 92
ROM memory instructions, 84, 95
ROM revision, 100
RPN, 29
    alpha string literals, 33
    branch length, 35, 66
    comments, 33
    indentation, 33
    labels, 35
    language, 32
    line numbers, 33, 35
    synonyms, 32
rules for linking, 49
run-time model directive, 21

S

.section
    directive, 13
section, 13
    alignment, 18
    BSS, 17
    Data, 17
    directive, 17
    fragments, 13
    kinds, 17
    modifiers, 18
    operators, 28
    placement, 49
    Read only data, 17
    Text, 17
.section directive, 14, 17
.sectionEnd
    operator, 28
.sectionSize
    operator, 28
.sectionStart
    operator, 28
.shadow directive, 22, 60
speed annotation, 57
.suppress directive, 14, 22
suppressing warnings, 22
suppressing XROM calls, 35
symbols
    global, 20
    quoted, 12
    required, 21
    syntax, 12
    weak, 20

T

.text directive, 14
Text section, 17
timer, 86

U

Ubuntu
    installation, 3
    uninstalling, 4
Unicode, 33
    UTF-8, 8, 10, 32, 101
uninstalling
    Arch Linux, 5
    Debian, 4
    Linux, 4, 5
    macOS, 3
    Manjaro, 5
    Ubuntu, 4
    Windows, 5
used memory, 51

V

VASM listings, 7, 10

W

wand, 88
warnings, 22
weak symbols, 20
Windows
    installation, 5
    uninstalling, 5

X

XROM
    calling, 37
    defining, 35
    function, 35, 37, 41
    instruction, 37, 38, 41
    instruction from RPN, 38
    number, 100
    with operands, 40