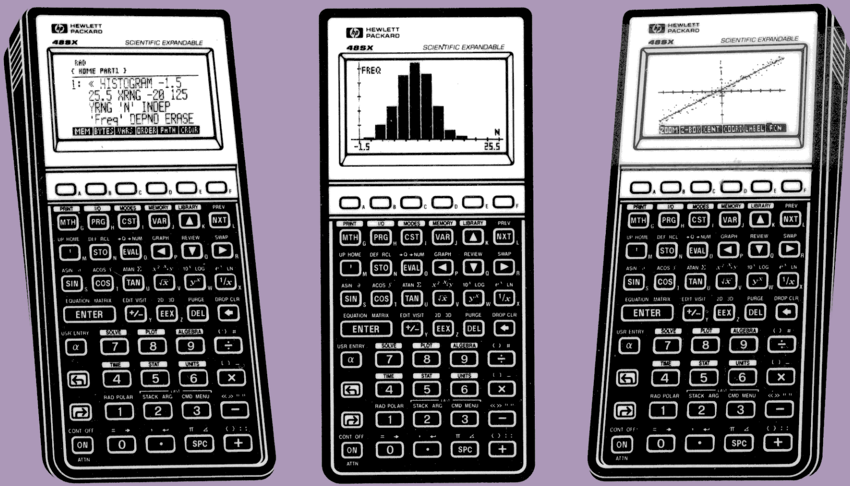
# HP 48 INSIGHTS

## PART I: Principles and Programming



William C. Wickes

# HP 48 Insights

## I. Principles and Programming

William C. Wickes

*Larken Publications*
*4517 NW Queens Avenue*
*Corvallis, Oregon 97330*

First Edition

First Printing, February 1991

ISBN 0-9625258-3-9

**Acknowledgements**

I thank my wife, Susan, and my children, Kenneth and Lara, for their help in the preparation of this manuscript.

Thanks also to the originators: Bill, Bob, Bob, Charlie, Diana, Gabe, Grant, Laurence, Max, Nathan, Pat, Paul, Stan, and Ted, for converting brainstorms into bits.

*Dedicated to Susan, Ken, and Lara*

## *Author's Note*

*Readers of this book who have previously read* HP-28 Insights *or* HP 41/HP 48 Insights *may notice that there is some material here that is common to one or both of those books. This is deliberate;* HP 48 Insights *was developed as a revision of* HP-28 Insights *just as the HP 48 is a revision of the HP 28. As the new book progressed, it became apparent that there was too much material to be contained in a single volume. Accordingly, the book has been split into three parts. The first of these is* HP 41/HP 48 Transitions, *which contains all of the HP 41-related material that was present in* HP-28 Insights, *plus additional content to make that book self-contained.* Part I *of* HP 48 Insights, *as its subtitle suggests, focuses on the principles of HP 48 design and various programming methods and resources.* Part II, *to be published later in 1991, will cover more of the integrated systems: HP Solve, unit management, plotting, statistics, etc.*

*Thanks to all of the readers of my books, starting with* Synthetic Programming on the HP-41C *back in 1980, for their continued support and encouragement. Even in this day of powerful desktop computing systems, there remains something special about a customizable handheld calculator like the HP 48 that makes it fun to write about as well as to use.*

*William C. Wickes*
*January 28, 1991*

CONTENTS

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 1
    1.1 The Evolution of the HP 48 . . . . . . . . . . . . . . . . . 2
    1.2 About This Book . . . . . . . . . . . . . . . . . . . . 5
    1.3 Notation . . . . . . . . . . . . . . . . . . . . . . . 8
    1.4 Terminology . . . . . . . . . . . . . . . . . . . . . . 12
    1.5 Easy to Use or Easy to Learn? . . . . . . . . . . . . . . . 13

2. Understanding RPN . . . . . . . . . . . . . . . . . . . . . 15
    2.1 The Evaluation of Mathematical Expressions . . . . . . . . . 16
    2.2 Calculator RPN . . . . . . . . . . . . . . . . . . . . 19
    2.3 RPL RPN . . . . . . . . . . . . . . . . . . . . . . . 21

3. Objects and Execution . . . . . . . . . . . . . . . . . . . . 25
    3.1 Operations . . . . . . . . . . . . . . . . . . . . . . 25
    3.2 Objects . . . . . . . . . . . . . . . . . . . . . . . 27
        3.2.1 Operations as Objects 29
    3.3 Execution and Evaluation . . . . . . . . . . . . . . . . 30
        3.3.1 When are Objects Executed? 31
    3.4 Data Objects . . . . . . . . . . . . . . . . . . . . . 32
        3.4.1 Real Numbers 33
        3.4.2 Complex Numbers 34
        3.4.3 Strings 36
            3.4.3.1 Concatenation 37
            3.4.3.2 String Comparisons 38
            3.4.3.3 Other String Manipulation Commands 38
        3.4.4 Arrays 40
        3.4.5 Lists 40
        3.4.6 Binary Integers 40
        3.4.7 Graphics Objects 42
        3.4.8 Tagged Objects 42
        3.4.9 Unit Objects 44
        3.4.10 Directories 45
        3.4.11 Libraries 45
        3.4.12 Backup Objects 46
    3.5 Procedure Objects . . . . . . . . . . . . . . . . . . . 47
        3.5.1 Program Objects 47
        3.5.2 Algebraic Objects 47
            3.5.2.1 Expression Structure 50
        3.5.3 Lists as Procedures 51
        3.5.4 Commands and Functions 51

3.5.5 Function Execution 52
3.5.5.1 Automatic Simplification 52
3.5.5.2 Symbolic and Numerical Execution; →NUM 53
3.5.6 Symbolic Constants 56
3.5.6.1 Other Symbolic Constants 57
3.5.6.2 Evaluation of Symbolic Constants 58
3.6 Name Objects . . . . . . . . . . . . . . . . . . . . 58
3.6.1 Global Names 59
3.6.2 Local Names 61
3.6.3 XLIB Names 61
3.7 Quoted Names . . . . . . . . . . . . . . . . . . . . 62
3.8 Quotes in General . . . . . . . . . . . . . . . . . . . . 62
3.9 EVAL . . . . . . . . . . . . . . . . . . . . 64
3.10 System Objects . . . . . . . . . . . . . . . . . . . . 64
3.10.1 SYSEVAL 64

4. The HP 48 Stack . . . . . . . . . . . . . . . . . . . . 67
4.1 Clearing the Stack . . . . . . . . . . . . . . . . . . . . 68
4.2 Rearranging the Stack . . . . . . . . . . . . . . . . . . . . 69
4.2.1 Exchanging Two Arguments 69
4.2.2 Rolling the Stack 69
4.2.3 Copying Stack Objects 70
4.2.4 How Many Stack Objects? 71
4.3 Recovering Arguments . . . . . . . . . . . . . . . . . . . . 72
4.4 Stack Manipulations and Local Variables . . . . . . . . . . 73
4.5 The Interactive Stack . . . . . . . . . . . . . . . . . . . . 74
4.6 Managing the Unlimited Stack . . . . . . . . . . . . . . . . 76
4.6.1 Stack Housekeeping 77
4.6.2 A Really Empty Stack 79
4.6.3 Disappearing Arguments 79
4.7 Design Insights . . . . . . . . . . . . . . . . . . . . 81

5. Storing Objects . . . . . . . . . . . . . . . . . . . . 83
5.1 Global Variables . . . . . . . . . . . . . . . . . . . . 83
5.1.1 DEFINE 86
5.1.2 Deleting and Renaming a Variable 86
5.1.3 Cancelling STO and PURGE 87
5.2 Directories . . . . . . . . . . . . . . . . . . . . 88
5.2.1 Organizing User Memory 92
5.2.2 Directory Objects 93
5.2.3 Purging Multiple Variables and Directories 96
5.3 Ports . . . . . . . . . . . . . . . . . . . . 96
5.3.1 Port Variables 97

5.3.1.1 The LIBRARY Menu 98
5.3.1.2 Altering Port Variables 99
5.3.2 Libraries 100
5.3.2.1 Other Library Commands 104
5.3.3 Plug-In Ports 105
5.3.4 Archiving Memory 106
5.4 Local Variables . . . . . . . . . . . . . . . . . . . 109
5.5 Name Resolution . . . . . . . . . . . . . . . . . . . 110
5.5.1 Command Line Entry 111
5.5.2 Executing Name Objects 112
5.5.2.1 Resolution Failures 113
5.5.3 Path Names 113
5.6 Named Objects vs. Registers and Files . . . . . . . . . . . 115
5.7 Additional Global and Local Variable Operations . . . . . . . . 116
5.7.1 Recalling Values 116
5.7.2 Altering the Contents of Variables 117
5.7.2.1 Store Menu Commands 118
5.7.2.2 Counter Variables 119
5.7.2.3 PUT and PUTI 119
5.7.2.4 Additional Array Commands 120
5.7.3 Cataloging and Finding Variables 121
5.7.4 Moving A Variable 122
5.8 Calculator Resets . . . . . . . . . . . . . . . . . . . 123

6. Methods . . . . . . . . . . . . . . . . . . . . . . . 125
6.1 The Basic Interface . . . . . . . . . . . . . . . . . . 125
6.2 Keyboard Mastery . . . . . . . . . . . . . . . . . . . 127
6.2.1 Keystroke Strategies 128
6.2.2 Navigating the Menus 129
6.2.2.1 Mechanics 129
6.2.2.2 Content 130
6.2.2.3 Exiting 132
6.2.3 ATTN 133
6.3 Hidden Operations . . . . . . . . . . . . . . . . . . 133
6.3.1 No-Command-Line Key Actions 134
6.3.2 Shift Key Meanings 134
6.3.2.1 Extended Actions 135
6.3.2.2 Store and Recall Actions 137
6.3.2.3 Other Hidden Shift Actions 138
6.4 Object Entry . . . . . . . . . . . . . . . . . . . . 139
6.4.1 Key Definitions and Entry Modes 140
6.4.2 Controlling the Entry Mode 142

6.4.3 ENTER in Detail 143
6.4.3.1 Comments 146
6.5 Object Editing and Viewing . . . . . . . . . . . . . . . . . 147
6.5.1 Viewing Objects 148
6.6 The Matrix Writer . . . . . . . . . . . . . . . . . . . . 149
6.6.1 Array Entry 151
6.6.1.1 Vector Entry 153
6.6.2 Editing Cells 154
6.6.2.1 Changing Array Dimensions 154
6.6.2.2 Stack Access 155
6.7 The EquationWriter . . . . . . . . . . . . . . . . . . . 156
6.7.1 The EquationWriter Display 158
6.7.1.1 Invoking the EquationWriter 159
6.7.2 Basic Expression Entry 160
6.7.2.1 Number Entry 162
6.7.2.2 Names and Prefix Functions 162
6.7.2.3 +, −, × 162
6.7.2.4 Division 163
6.7.2.5 Exponents 165
6.7.3 Special Forms 165
6.7.3.1 Square Root 165
6.7.3.2 *x*th-Root 166
6.7.3.3 Derivative 166
6.7.3.4 Integral 166
6.7.3.5 Summation 167
6.7.3.6 Where 168
6.7.3.7 Units 168
6.7.4 Correcting Mistakes 168
6.7.5 Stack Access 169
6.7.6 Subexpression Operations 170

7. Customization . . . . . . . . . . . . . . . . . . . . . . 175
7.1 Modes and Flags. . . . . . . . . . . . . . . . . . . . . 175
7.1.1 Flag Commands 176
7.1.2 System Flag Assignments 178
7.2 Key Assignments . . . . . . . . . . . . . . . . . . . . 180
7.2.1 Single Key Assignments 180
7.2.1.1 An Interactive Key Assignment Program 181
7.2.2 Multiple Key Assignments 182
7.2.3 Key assignments and memory 184
7.3 Custom Menus . . . . . . . . . . . . . . . . . . . . . 185
7.3.1 Built-in Menus 186

7.3.2 Custom Menu Object Types 186
7.3.3 Menu Key Labels and Shifted Menu Key Actions 189
7.4 Vectored ENTER . . . . . . . . . . . . . . . . . 190
7.4.1 Examples 192

8. Problem Solving . . . . . . . . . . . . . . . . . . 195
8.1 HP Solve . . . . . . . . . . . . . . . . . . . . . 196
8.2 Symbolic Manipulations . . . . . . . . . . . . . . 197
8.3 Programs . . . . . . . . . . . . . . . . . . . . . 201
8.4 Summary . . . . . . . . . . . . . . . . . . . . . 203
8.5 User-Defined Functions . . . . . . . . . . . . . . 204
8.5.1 User-Defined Function Structure 206
8.5.2 User-defined Functions as Mathematical Functions 207
8.5.3 Defining Programs 210
8.5.4 Additional Examples: Geometric Formulae 210

9. Programming . . . . . . . . . . . . . . . . . . . . 213
9.1 Program Basics . . . . . . . . . . . . . . . . . . 214
9.1.1 The ≪ ≫ Delimiters 214
9.1.2 The Program Body 215
9.1.3 Structured Programming 216
9.2 Program Structures . . . . . . . . . . . . . . . . 219
9.3 Tests and Flags . . . . . . . . . . . . . . . . . . 221
9.3.1 HP 48 Test Commands 223
9.3.2 Equality 223
9.4 Conditional Branches . . . . . . . . . . . . . . . 225
9.4.1 Simple Branches: The IF structure. 225
9.4.2 RPN Command Forms 227
9.4.3 The CASE Structure 228
9.5 Loops and Iteration . . . . . . . . . . . . . . . . 230
9.5.1 Definite Loops 230
9.5.1.1 Summations 231
9.5.1.2 Varying the Step Size 233
9.5.1.3 Looping with No Index 234
9.5.1.4 Exiting from a Definite Loop 234
9.5.2 Indefinite Loops 235
9.5.2.1 DO Loops 236
9.5.2.2 WHILE Loops 237
9.5.2.3 DO vs. WHILE 238
9.6 Error Handling . . . . . . . . . . . . . . . . . . 239
9.6.1 The ATTN Key 241
9.6.2 Custom Errors 241
9.6.3 Error Handling and Argument Recovery 242

9.6.4 Exceptions 243
9.7 Local Variables . . . . . . . . . . . . . . . . . . . 245
9.7.1 Comparison of Local and Global Variables and Names 248
9.8 Local Name Resolution . . . . . . . . . . . . . . . . 249
9.8.1 Resolution Speed 253

10. Display Operations and Graphics . . . . . . . . . . . . . . . . 255
10.1 Controlling the Display . . . . . . . . . . . . . . . . 256
10.1.1 Postponing the Standard Display 257
10.2 Text Displays . . . . . . . . . . . . . . . . . . 258
10.3 Graphics Displays . . . . . . . . . . . . . . . . . . . 260
10.3.1 Graphics Object Operations 260
10.3.2 Graphical Text 265
10.3.3 Displays on the Graph Screen 266
10.3.4 Logical Coordinates 269
10.3.5 Pixel Drawing 270
10.3.5.1 Off-Screen Coordinates 274

11. Arrays and Lists . . . . . . . . . . . . . . . . . . . . 277
11.1 Arrays . . . . . . . . . . . . . . . . . . . . . 277
11.2 Arrays and Algebraic Objects . . . . . . . . . . . . . . 281
11.3 Vectors and Coordinate Systems . . . . . . . . . . . . . 282
11.3.1 Coordinate Systems 284
11.3.2 Example: Coordinate Transformations 288
11.4 Lists . . . . . . . . . . . . . . . . . . . . . . 289
11.4.1 List Operations 290
11.5 List Applications . . . . . . . . . . . . . . . . . . 293
11.5.1 Input Lists 294
11.5.1.1 Index List Arguments 295
11.5.2 Output Lists 296
11.5.3 Lists of Intermediate Results 296
11.6 Composite Objects and Memory . . . . . . . . . . . . 299
11.7 Symbolic Arrays . . . . . . . . . . . . . . . . . . 301
11.7.1 Utilities 301
11.7.2 Symbolic Array Arithmetic 305
11.7.3 Determinants and Characteristic Equations 307

12. Program Development . . . . . . . . . . . . . . . . . . 311
12.1 Program Editing . . . . . . . . . . . . . . . . . . 311
12.1.1 Low Memory Editing Strategies 312
12.2 Starting and Stopping . . . . . . . . . . . . . . . . . 313
12.2.1 ATTN, DOERR and KILL 315
12.2.2 Single-Stepping 316

12.3 Debugging . . . . . . . . . . . . . . . . . . . . 317
12.4 Program Optimization . . . . . . . . . . . . . . . 323
12.5 Memory Use . . . . . . . . . . . . . . . . . . . 326
12.5.1 Using BYTES 327
12.6 Obtaining Input. . . . . . . . . . . . . . . . . . . 329
12.6.1 Halting for Input 329
12.6.1.1 Verbose Prompts 332
12.6.1.2 Prompting with Menus 333
12.6.2 Protected Entry. 335
12.6.3 Using INPUT. 335
12.6.4 Keystroke Input 339
12.6.4.1 KEY 339
12.6.4.2 WAIT 341
12.6.4.3 ATTN 341
12.6.4.4 An Input Programming Example 341
12.7 Displaying Output . . . . . . . . . . . . . . . . . 344
12.7.1 Tagged Objects 345
12.8 Programs as Arguments . . . . . . . . . . . . . . . 346
12.9 Timing Execution . . . . . . . . . . . . . . . . . 349
12.9.1 Erratic Execution 351
12.10 Recursive Programming . . . . . . . . . . . . . . . 352
12.11 Additional Program Examples . . . . . . . . . . . . 354
12.11.1 Random Number Generators 354
12.11.1.1 Poisson Distribution 354
12.11.1.2 Normal Distribution 356
12.11.2 Prime Numbers 359
12.11.3 Simultaneous Equations 361
12.11.4 Infinite Sums 364
12.11.4.1 Sine Integral 365
12.11.4.2 Cosine Integral 366
12.11.4.3 Sum Programs 368

Program Index . . . . . . . . . . . . . . . . . . . . . 371

Subject Index . . . . . . . . . . . . . . . . . . . . . . 373

# 1. Introduction

The HP 48 is a unique calculator. No other handheld device can match its combination of mathematical capability, customizability, and extensibility. It uniqueness, however, means that it contains facilities and uses methods that are new and special to it, making it in many respects a challenge to learn to use effectively. If you are a new user of the HP 48, you may well be a little overwhelmed or even intimidated by the sheer extent of the HP 48's capabilities. You might also imagine that it will take you a long time to master the calculator. Fortunately, this shouldn't be true. Running throughout the HP 48's feature set and methodology are a few common themes and principles; understand those and you will find it easy to assimilate and use each new calculator operation that you study.

There are, of course, many different approaches to teaching the use of a device like the HP 48; no one approach is best for everyone. One method is to teach everything by example, and trust that the underlying principles will become apparent. This is the style of the HP 48 owners' manuals, which works quite well for many people. In this book we will take a different tack and start with the principles, with examples to illustrate the principles. We believe that a clear understanding of those principles helps you understand the examples and extrapolate them more easily to problems for which you don't have explicit examples.

For example, here's how you add two numbers on the HP 48:

1. Key in the first number.
2. Key in the second number.
3. Press $\boxed{+}$.

If you're familiar with traditional HP scientific calculators, you will recognize this as the standard "RPN" keystroke sequence for addition. If you have only used so-called "algebraic" calculators, the sequence may seem a little awkward--but we'll postpone explanation and justification to Chapter 2. The principle involved is the application of a function, in this case +, to arguments that appear on a "stack" of such arguments; the function's result replaces its arguments on that stack. The specific example here shows how two ordinary real numbers are added--once you've learned this sequence, you immediately know also how to add, for example, two complex numbers or two vectors. Just take the above instructions and substitute "complex number," or "vector," everywhere you see "number." You follow the same logical sequence, and press the same $\boxed{+}$ key, for all of the kinds of addition that the HP 48 provides. This consistency and uniformity runs through all HP 48 operations.

When we use the term *HP 48*, we are including the HP 48SX and any other HP 48 calculators that share a common package and operation with the HP 48SX. Successful Hewlett-Packard calculators in the past have often developed into families of several calculators with the same number, such as the HP 41C, HP 41CV, and HP 41CX. For the sake of simplicity and generality, we will generally not use the trailing letters of a calculator's name unless referring to a specific model.

## 1.1 The Evolution of the HP 48

In 1972, Hewlett-Packard introduced the HP 35, an "electronic slide-rule" that revolutionized the world of numerical calculations. It offered high-precision arithmetic, logarithmic, and trigonometric functions at the press of a key, obsoleting slide-rules and thick function tables. The HP 35 was followed by numerous similar products, from HP and from other manufacturers, that expanded on the HP 35 theme by offering more functions and more data storage registers.

A second generation of calculators was started by the HP 65, the first programmable calculator. This calculator allowed you to customize it by creating programs, in effect extending the built-in command set. Like the HP 35, the HP 65 was followed by numerous variations on the programming theme, including handheld computers programmable in BASIC. Perhaps the most successful of these was the HP 41 family, starting with the HP 41C in 1979, which quickly became the standard among engineering calculators. The HP 41's ten-year lifetime, remarkably long in this era of rapid changes in computing technology, resulted from its powerful combination of built-in functions, customizability, and extensibility--the same virtues we extolled above for the HP 48.

The HP 41 and all of the other first- and second-generation calculators share two common limitations. First, they are optimized only for dealing with real floating-point numbers. Some calculators allow you to work with character strings, complex numbers, and/or matrices, but typically each additional data type has its own special commands or working environment, requiring you to learn new calculation methods and making it hard to combine different data types in the same calculation. Second, none of these calculators allow you deal with programs as *unevaluated* mathematical quantities. For example, you can write programs to calculate $a+b$, and $c+d$, but there is no way for you to manipulate the program results to produce a new result like $a+b+c+d$ except by running the programs to produce numerical results, then combining the numbers.

A third generation of calculators was born with the advent of the HP 28C in 1987. The first generation was characterized by the application of built-in functions to real numbers. The second generation added extension of the built-in function set by user programs. The HP 28C made a major leap in calculator technology by making the programs themselves subject to logical and mathematical operations. In short, the HP 28C is the first *symbolic* calculator--on which calculations can be represented as unevaluated expressions and programs, to which you can apply the same operations that you can apply only to numbers on other calculators. Moreover, the HP 28C allows you to work with a variety of data types, including the strings and matrices mentioned above, using exactly the same logic and keystrokes that you use for ordinary numbers. The most important of these new data types is the *algebraic object*. You can, for example, enter algebraic objects that represent $a+b$ and $c+d$ symbolically, then press the $\boxed{+}$ key to return the new symbolic result $a+b+c+d$. The variables do not have to have numeric values *before* you can add them. Most HP 28C mathematical functions, in fact, can accept symbolic inputs and return symbolic results. Not only does this mean that you can perform symbolic algebra, and even calculus, right on the HP 28C, but at a stroke, much of the work of programming disappears. These capabilities represent such a dramatic advance over previous calculator technology that they merit the description "third generation."

The HP 35 introduced a standard "user-interface" called RPN (short for *Reverse Polish Notation*), that has been the hallmark of HP calculators ever since. RPN calculators are organized around a stack of number registers, using a last-in-first-out logic that is optimal for key-per-function operation. Throughout the evolution of HP calculators from the HP 35 up through the HP 41, that standard RPN interface remained virtually unchanged. If you were familiar with one HP calculator, you could pick up any other and use it right away--that is, until the advent of the HP 28C. The HP 28C succeeded in preserving the advantages of RPN while making important changes to generalize the interface to handle the HP 28C's wealth of new data types, most particularly including *variables* and *expressions* for symbolic mathematics.

The HP 28C's advances in calculation ability were so compelling that the calculator was very popular despite a severe handicap--a small memory that made it impractical to use the calculator for anything but modest-sized computations and programs. This deficiency was corrected in a new HP 28 model, the HP 28S, introduced in January, 1988. The first public appearance of HP 28S calculators were special models built to commemorate the 100th anniversary of the American Mathematical Society, delivered at the joint annual meeting of the AMS and the Mathematical Association of America. This was a highly appropriate forum for the introduction, because of the profound impact the HP 28C was having on the mathematics education community. Driven by students and imaginative educators, with whom the HP 28 was an instant hit, the HP 28S is now a standard teaching tool at many universities.

Although the HP 28 was quite successful in engineering and scientific disciplines, it is fair to say that it did not have as dramatic an impact in those fields as in mathematics. This is partly due to the earlier success of the HP 41 with technical users, since they were accustomed to the extensibility provided by the HP 41's plug-in memory ports and consequently less ready to switch to a calculator that lacked that feature. The HP 41's utility was greatly enhanced by the availability of a large amount of professional and amateur software, which could be loaded into the calculator by several automated methods. A similar software base never developed for the HP 28, since its only program entry method is the keyboard.

The HP 48SX, introduced in March, 1990, is a direct descendent of both the HP 41 and the HP 28. Normally, the numbers associated with HP calculators have little significance, but it is hard not to notice that the number *48* itself is a cross between *41* and *28*. From the HP 41, the HP 48SX inherits:

- Plug-in memory ports.
- I/O capability (the HP 41 used HP-IL; the HP 48SX uses a serial communications that is a standard on personal computers).
- A redefinable keyboard.
- The "vertical format" keyboard layout that is convenient for handheld operation.

The HP 28 contributed:

- Extensive real and symbolic mathematical capabilities.
- The operating system and user language.
- Plotting and a graphics display.
- The menu key system.

The HP 48SX also benefited from users' reaction to the HP 28, adding the most-requested features missing from the HP 28:

- A bigger display.
- More graphics and plotting features.
- Bi-directional infrared I/O, especially for importing or saving software.
- Symbolic integration, beyond the Taylor's polynomials method used on the HP 28.
- More "help" from the calculator in using some of its more complicated features.

Some of these features evolved into major HP 48 systems that considerably exceeded the scope of a straightforward evolution from the HP 41 or the HP 28. For example, the HP 48 EquationWriter was an outgrowth of a need to improve the HP 28's mechanism for setting up numerical integration problems. The EquationWriter obviously satisfies that need, but has much broader application that just for integration problems.

Similarly, both the HP 41 (through plug-in programs) and the HP 28 contain some physical unit conversion capability, but the HP 48's unit management system is enormously more flexible, powerful, and usable than that of its predecessors.

In the matter of programming language, no simple convergence of the HP 41 language and the HP 28 language was possible. Although using the HP 41 language in the HP 48 would have made the HP 41 software base available for the new calculator, that language was stretched to its limit already by the HP 41 itself, and it is not capable of supporting the symbolic calculations that are the heart of the HP 28. Consequently, the HP 48 follows the HP 28 design--the HP 48 operating logic and programming language are effectively a superset of those of the HP 28. Computer languages are known for their whimsical names; the HP 28/HP 48 language is no exception, with the name *RPL*, which stands for *Reverse Polish Lisp*. This name suggests its HP 48's derivation from HP calculators (and from FORTH, another computer language that uses reverse Polish logic), and from the computer language LISP, which is frequently used in computer symbolic mathematics systems. Note that the HP 41 language was never given a name, so many people call HP 41 programming "RPN programming," which is unfortunate since, properly speaking, RPN is a mathematical logic that is not specific to any calculator or computer.

## 1.2 About This Book

The HP 48 naturally comes with an *Owner's Manual* that covers most of the calculator's features in varying levels of detail. A *Programmer's Reference Manual* is also available, which presents detailed information on individual commands. *HP 48 Insights* is not intended to supplant those books, but to supplement them. As stated earlier, *Insights* will concentrate on the principles and themes of HP 48 operation, and provide a depth of analysis that is not possible in a comprehensive in-box manual.

We also hope to provide a little more *motivation*, and some more elaborate examples. By motivation, we mean the purpose and use of many of the operations, and the connections between various features of the calculator. The scope of the HP 48 is so broad that we cannot show you how to use it for every imaginable problem, but we can try to help you understand it enough to solve your own problems. We delve quite deeply into the HP 48's principles of operation, with the expectation that if you know the principles, you will learn and remember keystrokes and methods much more easily.

We assume that you have read the HP 48 Owner's Manual enough that you at least know how to perform simple keystroke calculations, enter various object types, and find a command in a menu. In some cases, where there are crucial ideas that we want to communicate, we will show some actual keystroke sequences and certainly repeat some material that is in the HP manuals. But for the most part we will assume that you know the rudiments of HP 48 operation so that we can concentrate on ideas and connections.

*HP 48 Insights Part I* breaks roughly into two main sections. In the first section, Chapters 1 through 7, we discuss primarily the principles and concepts of HP 48 operation, starting with the mathematical ideas that underlie the HP 48's use of Reverse Polish Notation and the object stack, and finishing with a review of keyboard techniques and customization features. The second section, Chapters 8 through 12, is an extended discussion of HP 48 programming, beginning with a review of general problem solving techniques, continuing with a study of the structures and objects central to programming, and concluding with topics in program development.

The following summarizes the chapter topics:

| | **Chapter** | **Topics** |
|---|---|---|
| 1. | Introduction | Introductory material, notation conventions. |
| 2. | Understanding RPN | The theory of RPN, and its electronic implementation. |
| 3. | Objects and Execution | Operations, objects, execution and evaluation, quotes. |
| 4. | The HP 48 Stack | Stack operations, recovering arguments, the interactive stack. |
| 5. | Storing Objects | Creating, storing, recalling, evaluating and purging variables; directories; port variables; libraries; name resolution; calculator resets. |
| 6. | Methods | HP 48 keyboard design and methodology; hidden operations; object entry and editing; the MatrixWriter; the EquationWriter. |
| 7. | Customization | Modes and flags; user key assignments; custom menus; vectored ENTER. |
| 8. | Problem Solving | Introduction to HP 48 problem-solving methods; user-defined functions. |

| | | |
|---|---|---|
| 9. | Programming | The principles of program objects; tests and flags; conditional branches; loops; error handling; local variables. |
| 10. | Display Operations and Graphics | The text and graphics screens; graphics objects; displaying text and graphics; pixel drawing. |
| 11. | Arrays and Lists | Arrays; coordinate systems; lists and their applications; symbolic arrays. |
| 12. | Program Development | The art of program construction; editing and debugging; starting and stopping; optimization; input and output; programs as arguments; recursion. |

You may find the early chapters occasionally to be heavy going, with their emphasis on theory and terminology. Nevertheless, we recommend that you read those chapters through at least enough to insure that you have a grasp of the definitions and terms that we introduce there, which are used throughout the second part of the book. In particular, the concepts of *operations*, *objects*, *execution*, and *evaluation*, described in Chapter 3, are used extensively in all of the material that follows.

The presentation of the book's subject matter is not necessarily linear. That is, we often make use of or refer to concepts or techniques that are not explained until later sections. For example, in Chapter 4 there are listings of some elaborate programs that are relevant to the material under discussion. The programming methods used in the programs are not described until later chapters. Furthermore, wherever possible, examples that illustrate a concept are chosen to have practical uses as well. This often requires combining more techniques into an example than just the one currently being studied. To alleviate this kind of problem, we include many cross-references between the sections, and a subject index. And, of course, you are encouraged to jump around in your reading. When you read about error-trapping in section 9.6, you should go back and look at the program XARCHIVE in section 5.3.4 to see how it deals with errors.

*Part I* of *HP48 Insights* touches only lightly on or omits altogether major HP48 features such as HP Solve, symbolic mathematics, and automated plotting. These topics and the other integrated systems or applications are left for *Part II*. The use of all of those systems is greatly enhanced by programming, so we choose to present programming first, as a foundation from which to explore the rest of the HP48's capabilities.

## 1.3 Notation

In order to help you recognize various calculator commands, keystroke sequences, and results, we use throughout this book certain notation conventions:

- All calculator commands and displayed results that appear in the text are printed in helvetica characters, e.g. DUP 1 2 SWAP. When you see characters like these, you are to understand that they represent specific HP 48 operations rather than any ordinary English-language meanings.

- Italics used within calculator operations sequences indicate varying inputs or results. For example, 123 'REG' STO means that 123 is stored in the specific variable REG, whereas 123 '*name*' STO indicates that the 123 is stored in a variable for which you may choose any name you want. Similarly, « *program* » indicates an unspecified program object; { *numbers* } might represent a list object containing numbers as its elements.

  Italics are also used for emphasis in ordinary text.

- HP 48 keys are displayed in helvetica characters surrounded by rectangular boxes, e.g. $\boxed{\text{ENTER}}$, $\boxed{\text{EVAL}}$, or $\boxed{\text{EEX}}$. The back-arrow key looks like this: $\boxed{⇦}$, and the cursor keys like these: $\boxed{◁}$, $\boxed{▷}$, $\boxed{△}$, and $\boxed{▽}$.

- A shifted key is shown with the key name in a box preceded by a left- or right-shift key picture, $\boxed{↰}$ or $\boxed{↱}$, e.g. $\boxed{↱}\boxed{\text{TIME}}$, or $\boxed{↰}\boxed{\text{PURGE}}$. A shifted key is identified by the orange or blue label above the key, rather than the label on the key itself--$\boxed{↰}$ $\boxed{\text{SOLVE}}$ rather than $\boxed{↰}\boxed{7}$.

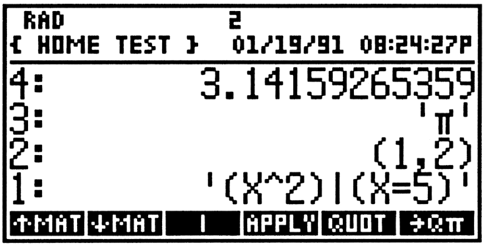
- Menu keys for operations available in the various menus are printed with the key labels surrounded by boxes drawn to suggest the reverse characters you see in the display, like these: ≣SIGN≣ or ≣→LIST≣.

Examples of HP 48 operations take several forms. When appropriate, we will give step-by-step instructions that include specific keystrokes and show the relevant levels of the stack, with comments, as in the following sample:

| **Keystrokes:** | **Results:** | | **Comments:** |
|---|---|---|---|
| 123 $\boxed{\text{ENTER}}$ 456 $\boxed{+}$ | 1: | 579 | Adding 123 and 456 returns 579 to level 1. |

For better legibility, we don't show individual letters and digits in key boxes--we just show 123 rather than $\boxed{1}\boxed{2}\boxed{3}$, and ABC rather than $\boxed{\alpha}\boxed{\alpha}\boxed{A}\boxed{B}\boxed{C}$. Key boxes are used for multi-letter keys on the keyboard and in menus.

In some cases, a printed listing of the stack contents isn't adequate, so we use an actual HP 48-generated picture of the calculator display, such as this picture from Chapter 6:



```
RAD                2
{ HOME TEST }    01/19/91 08:24:27P
4:          3.14159265359
3:                    'π'
2:                  (1,2)
1:          '(X^2)|(X=5)'
```

A large number of the examples, however, are given in a more compact format than the keystroke example shown above. These examples consist of a *sequence* of HP 48 commands and data that you are to execute, together with the stack objects that result from the execution. The "right hand" symbol ☞ is used as a shorthand for "the HP 48 returns..." In the compact format, the addition example is written as

123 456 + ☞ 579

The ☞ means "enter the objects and commands on the left, in left-to-right order, and the HP 48 will give back--*return*--the objects on the right." If there are multiple results, they are listed to the right of the ☞ in the order in which they are returned. For example,

A B C ROT SWAP ☞ B A C

indicates that B is returned to level 3, A to level 2, and C to level 1.

Because of the flexibility of the HP 48, there are usually several ways you can accomplish any given sequence, so we often don't specify precise keystrokes unless there are non-programmable operations in the sequence. If there are no key boxes in the left-side sequence, you can always obtain the right-side results by typing the left side as text into the command line, then pressing $\boxed{\text{ENTER}}$ when you get to the ☞ symbol.

The ☞ symbol is also used in the *stack diagrams* that are part of most program listings. The stack diagrams show how to set up stack objects for execution of the program, where the objects to the left of the ☞ are the "input" objects, and the objects following the ☞ are the program outputs.

The most elaborate "examples" in this book are *programs*. Each program is listed in a box that includes a suggested program variable name, a stack diagram, the actual steps that make up the program, and comments to help you understand the steps. The following sample listing illustrates the various features of the format:

| SAMPLE | *Sample Program Listing* | | | | *checksum* |
|---|---|---|---|---|---|
| | *level 3* | *level 2* | *level 1* | \| | *level 1* |
| | *"string"* | *[matrix]* | *n* | ☞ | *[matrix']* |

| | |
|---|---|
| `<< A B → a b` | Start of program. |
| `  <<` | Start of local variable procedure |
| `    IF C D` | Start of IF structure. |
| `    THEN 1 2 → n m` | |
| `      <<` | Start of local variable procedure. |
| `        START E F` | Start of definite loop. |
| `          DO G UNTIL H END` | DO loop. |
| `        NEXT` | End of definite loop. |
| `      >>` | End of local variable procedure. |
| `    ELSE I J` | |
| `    END` | End of IF structure. |
| `  >>` | End of local variable procedure. |
| `>>` | End of program. |

1. The name of the program (SAMPLE) is listed first, followed by an expanded version of the name that is descriptive of its purpose. When you have entered the listed program, you should store it in a variable with the specified name. If no name is given, the program is just intended to illustrate some point in the text, and there's no need to give it any particular name.

2. The program's checksum is listed at the end of the name line, as a four-digit hexadecimal number. If you enter the program into your HP 48, you can verify that you have entered it correctly by comparing the listed checksum with the value returned by BYTES (section 12.5.1) for your program.

3. Below the program name is a *stack diagram*, that specifies the program's input and output on the stack. The program *arguments* are shown to the left of the ☞, and the *results* to the right. In the example, the stack diagram indicates that the program requires a string in level 3, a matrix in level 2, and a real number *n* in level 1, and returns a new matrix in level 1. The object symbols in the stack diagram are as descriptive as possible, showing not only the required object type but also

the conceptual purpose of the objects. A stack diagram

*length width height* ☞ *volume*

shows that a program takes three real numbers (no object delimiters) representing length, width, and height, and returns another real number that is the volume.

4. The program listing is broken into lines, where each line has one or more program objects listed at the left, and explanatory comments on the right. There may be just one object on a line, or several whenever the collective effect of the objects is easy to follow. You do not have to use the same line breaks (or any at all) when you enter the program.

5. Lists, embedded programs, and program structures start on a new line unless they are short enough to fit entirely on one line. More frequently, each program or list delimiter or structure word starts a new line. The sequences between the structure words are indented, so that the structure words stand out. In the case of nested structures, each structure word of a particular structure is lined up vertically at the same indentation from the left margin. (The structure word → does not start a new line, but the local variable defining procedure that follows the → does start a new line.) Note that when you edit a program on the HP 48, the program display follows these same conventions, within the limitation of the 22-character display or printer width.

6. The comments at the right of the listing describe the purpose or results of the program lines at the left. If you are creating a program using a personal computer text editor, you can include similar comments in your program, setting them off from the program objects using the @ delimiter (section 6.4.3.1). An especially useful "comment" is a description of the contents of the stack that are obtained after the execution of a program line. In our listings, the stack contents are distinguished from ordinary comments by enclosing the stack objects between | | symbols. The leftmost object in the series is in the highest stack level; the rightmost is in level 1. Thus

   | *a* *b* *c* *d* |

   indicates that the object *a* is in level 4, *b* in level 3, *c* in level 2, and *d* in level 1.

We recommend that you use similar conventions when developing and recording your own programs. Whether you write programs out by hand and type them into the HP 48, or use a personal computer to write programs and transfer them to the HP 48 via the serial port, program stack diagrams and comments are invaluable for later understanding and modification of the programs. Of course, there will be many occasions when you create a program directly in the HP 48 command line without benefit of any program listing. In these cases, we still recommend that you afterwards make a listing, or copy the program to a personal computer file, so that you can recover the program if you lose it for any reason.

## 1.4 Terminology

Finding useful terminology to describe a computer system like the HP 48 with new or unusual features can be a substantial problem. We have to use existing English words that are close to the meaning we wish to convey, but the dictionary definitions of the words usually differ from their meanings as applied to the HP 48. Consider the word *object*: for the HP 48, *object* means any of the mathematical or logical elements that constitute the data and procedural building blocks of the RPL language, but you won't find that meaning in a dictionary (although it is close to the definition used in mathematics).

Our solution to this difficulty is to provide precise definitions of any terms that we use that are specific to the HP 48, and then use those definitions consistently throughout. In some cases, the definitions we offer may differ from those used in the HP 48 manuals, usually because we need more careful definitions to get across a particular point. For example, the owners' manuals do not make a distinction between *execute* and *evaluate*. We find that such a distinction is useful (section 3.3) because it simplifies the descriptions of related subjects, such as the nature of global name objects (section 3.6.1).

Two other important terms that arise frequently are *mode* and *environment*. A mode is a calculator setting, often associated with one or more *flags* (section 7.1), that determines how a particular keystroke or command will behave. For example, in *polar mode*, complex numbers and vectors are displayed in polar coordinates rather than the usual rectangular coordinates. An *environment* is a glorified mode, which determines the entire calculator interface, including the display, key actions, and available operations.

The "home base" for the HP 48 is the *standard environment*. In this environment, the display shows the status area, stack, and menu key labels. All keys are active, with their ordinary labeled definitions. If you press ⬅ GRAPH, the HP 48 switches to the *plot environment*. Here the display is devoted to a graph or other picture, the menu keys are restricted to a menu of plotting operations, and the remaining keys are either assigned additional plot actions or are inactive altogether. Pressing ATTN returns to the standard environment. Other environments include the EquationWriter, the MatrixWriter, and the equation and statistics matrix catalogs.

While introducing and using this kind of specialized terminology, at the same time we will be using an informal style that takes some liberties with the language to avoid unnecessarily stilted descriptions. "You are in the program branch menu" is almost a non-sequitor when taken out of context, but it reads more easily than "the current HP 48 menu is the program branch menu," and its meaning is clear.

## 1.5 Easy to Use or Easy to Learn?

It would be nice if you could pick up the HP 48 and use all of its facilities without ever referring to a manual. A common criticism of the HP 48 is that it takes a long time to master, particularly by comparison with other recent HP calculator products such as the HP 17B and the HP 19B, and with some of the simpler function-plotting calculators made by other manufacturers that have become popular in mathematics education at the pre-calculus level. But these calculators obtain their ease of learning by having very limited computational capabilities and flexibility compared to the HP 48. If your problem "fits" on one of these other calculators, then it is easy to use as well as easy to learn. But if you want to do something just a little different, you will find that "easy to learn" translates to "impossible to use."

The HP 48 approach is to provide a broad, very flexible set of computational capabilities, many of which have never before been available on a handheld calculator. Furthermore, it is expressly designed for "linking" calculations together--the results of one calculation are always ready to be used as input for another, even if you didn't know in advance that your work would proceed that way, and even if the calculator designers didn't expect you to make that particular combination of calculations. These ideas are what the HP 48 means by "ease-of-use."

"Ease-of-learning" is a different story. Unfortunately, the HP 48's rich capability set doesn't leave enough built-in memory to provide "no-manual" learning. And there's no doubt that the HP 48 does work differently from other calculators, even from its RPN calculator predecessors like the HP 41. You have no choice but to spend some time reading the manuals and learning new procedures. But learning the basic ideas doesn't take long, and once you master them, a wide range of truly easy-to-use calculating capabilities is available to you.

# 2. Understanding RPN

The HP-48, like most of its Hewlett-Packard calculator predecessors, presents a user interface centered around a logic called "RPN," short for *Reverse Polish Notation*. If you are unfamiliar with this logic, particularly if you are accustomed to so-called "algebraic" calculators, RPN may seem awkward and unfamiliar. In this chapter, we will explain how RPN works, and why its virtues make it the choice for the HP-48.

Many people use a calculator in a style that you might call "fingers in, eyes out." That is, they manually type in all of the data for a calculation and read out the result visually from the display, perhaps writing it down on paper. For this type of use, a calculator that uses "algebraic" entry seems desirable, because in at least simple cases the keystrokes follow more-or-less the order of common written mathematical notation.

The algebraic style, however, is not well suited for exploratory calculation, where you don't necessarily know what to do next until you see the results of previous calculations--and you need those results as part of the next calculation. When you press an algebraic calculator's $\boxed{=}$ key to complete a calculation, you had better be sure that you're finished, because the result you see in the display may vanish at the next keystroke.

The choice and design of an RPN system for a calculator arises from consideration of one central principle:

- *The result of any calculation, no matter how complicated, may be used as an input for a subsequent calculation.*

RPN calculators are designed to embody this principle, by providing a mechanism (the "stack") whereby you can apply mathematical operations to data already entered into the calculator. The results of the operations are also held indefinitely, so that they, in turn, can be the input data for subsequent operations.

In the calculator world, the term *Reverse Polish Notation*, or more specifically, the abbreviation "RPN," has come to mean "the way HP calculators work." RPN actually is a mathematical notation; HP calculators provide an electronic implementation of the notation. In RPN, mathematical functions are written *after* their arguments, not before or between the arguments as in ordinary written expressions. The notation appears strange, because we are not used to visualizing or writing expressions this way. However, when you actually evaluate an expression to a numerical value using pencil and paper, you must revert to an order of operations that exactly corresponds to RPN. We will illustrate this point by examining how mathematical expressions are evaluated.

## 2.1 The Evaluation of Mathematical Expressions

A mathematical *expression* is an abstract representation of the calculation of a single value. An expression combines data (numbers or other explicit quantities), variable names, and functions. When you *evaluate* an expression, you perform all of the calculations represented by the expression. Examples of expressions are:

$$1 + 2$$

$$x + y + 2z$$

$$\sin[\ln(x + 2)]$$

$$x^3 + 4x^2 - 6x + 2$$

We will confine our attention to expressions that can be formed from the mathematical functions included in the HP-48: arithmetic operations, powers, roots, transcendental functions, etc. Expressions like these have the property that they are equivalent to a single value. That is, if you perform the calculations represented by an expression, you end up with a single value as the result.

In our discussions, we will be using the following terms:

- A *function* is a mathematical operation that takes zero, one, or more values as input, and returns one value.
- A value used by a function as "input" is called an *argument*.
- A value returned by a function as "output" is called a *result*.
- A mathematical *variable* is a symbol that stands for a value. Evaluating a variable replaces the symbol with the value.
- *Algebraic syntax* is the set of rules that governs how data, variables, and functions may be combined in an expression.

As an example of these concepts, consider the following expression:

$$\sin[123 + 45\ln(27 - 6)]$$

The expression contains the *functions* sin, ln, $+$, $-$, and $\times$ (implied multiply between the 45 and the ln), and the *numbers* 123, 45, 27, and 6. The expression is written in common mathematical notation, but notice that the order in which you read or write the expression, i.e., left to right, does not correspond very well to the order you would use if you were actually going to evaluate the expression with pencil and paper and function tables. For example, although the ln function *precedes* the quantity $(27 - 6)$, you can't actually compute (or look up) the logarithm until *after* you have computed the difference

27 − 6. Similarly, the sin, which is the *first* function that appears in the expression, is actually the *last* that you will execute. You can not compute the sine until the entire rest of the expression [123 + 45 ln (27 − 6)] is evaluated.

The common mathematical notation that we are using here has been developed over the centuries to present a readable picture of a mathematical expression that takes advantage of a human's ability to view an entire expression at once and draw conclusions from its structure. But the notation is not a very good prescription for actually evaluating an expression--as you step through a calculation, you have to jump back and forth, match parentheses, etc. to find the next step. As we will show now, converting an expression into an orderly procedure for evaluation leads directly to RPN. First we'll adopt a uniform structure that treats all functions alike, then we'll turn it around to match actual calculation order.

Common notation is not uniform because the notation differs for one-argument and two-argument functions. In our sample expression, the one-argument functions sin, ln, and cos, are written *in front* of their arguments ("prefix" notation), whereas the two-argument functions + and − are written *between* their arguments ("infix"). Furthermore, there is an implied multiply between the 45 and the ln that is not explicitly written. Infix notation also leads to ambiguity. For example, does 1+2×3 evaluate to 9 or 7? You either have to introduce extra parentheses, e.g. (1+2)×3 or 1+(2×3), or use so-called *precedence* conventions that specify which functions are executed first in ambiguous situations. One of the drawbacks of non-RPN calculators is that there is no universal standard for precedence, so you have to memorize the precedence rules of each calculator you use.

A general-purpose form for functions is to write each function name followed by its arguments contained in parentheses, as in *f(x)*, *g(x,y)*, etc. You can make expressions more uniform by writing all of its functions in this prefix form:

$$\sin(\ +(123, \times(45, \ln(\ -(27,6)))))$$

In this notation, +(1,2) means "add 1 and 2"; ×(1,2) means multiply 1 by 2; etc.

Writing expressions this way is called *Polish notation*, honoring the Polish logician, Jan Łukasiweicz. Unfortunately, this notation appears practically unintelligible to people accustomed to conventional notation. But it does show explicitly the hierarchical structure of the expression, which we will discuss later (section 3.5.2.1). Also, it is useful because it is a step towards RPN. That is, you can obtain a form that corresponds more closely to the actual order of evaluation of an expression by rewriting the Polish form so that the function names *follow* their arguments' parentheses. For example, rewrite +(1,2) as (1,2)+. The example expression now becomes:

$$((123, (45, ((27,6) - ) \ln) \times ) + ) \sin$$

You have replaced Polish notation with *R*everse *P*olish *N*otation. In this form, the expression represents a step-by-step evaluation prescription for pencil-and-paper or electronic calculation, that follows the left-to-right order of the expression. To see this, consider an orderly pencil-and-paper method for evaluation:

- Start at the left of an RPN expression, and work to the right.
- When you come to a number, write it down below any previous numbers.
- When you come to a function, compute its value using the last number(s) you wrote as its arguments. Erase the argument number(s), and then write the function value.

Apply this procedure to calculate the example expression (keeping two decimal place accuracy):

| **Object** | **What to do** | **What you see** |
|---|---|---|
| 123 | Write 123 | 123 |
| 45 | Write 45 | 123<br>45 |
| 27 | Write 27 | 123<br>45<br>27 |
| 6 | Write 6 | 123<br>45<br>27<br>6 |
| − | Subtract 6 from 27 | 123<br>45<br>21 |
| ln | Find ln(21) | 123<br>45<br>3.04 |
| × | Multiply 45 and 3.04 | 123<br>137.00 |

| | | |
|---|---|---|
| + | Add 123 and 137.00 | 260.00 |
| sin | Take the sine of 260° | −.98 |

There are two things you can notice from this exercise:

- Whenever you encounter a function, you can execute it immediately because you have already calculated its arguments.
- You can ignore parentheses. When you write an expression in RPN form, you don't need parentheses, because there is no ambiguity of precedence--functions are always executed left-to-right.

The latter point means that you can eliminate parentheses from the notation. Doing so, the example becomes:

$$123 \quad 45 \quad 27 \quad 6 \quad - \quad \ln \quad \times \quad + \quad \sin$$

## 2.2 Calculator RPN

An RPN calculator allows you to substitute an electronic medium for paper. The calculator's ENTER key is the equivalent of "write it down" in paper calculations. You "write" a number by pressing the appropriate digit keys, then ENTER, which terminates digit entry and enters the number into the calculator's memory. The memory takes the place of paper.

For cases where you need to have more than one number written down at a time, calculator memory is organized into a "stack." You can visualize the stack as a vertical column of numbers, where the most recently entered numbers are at the bottom of the column, and the oldest numbers at the top. Each new entry "pushes" previous entries to higher stack levels. A function always operates on the latest stack entry or entries, and replaces those entries with its result, where it is ready for use by the next function to come along. If one or more entries are removed from the stack, older entries drop down to fill in the vacant levels. Again, this is quite analogous to the pencil-and-paper technique you used in the example.

To illustrate calculator RPN, redo the previous example on the HP-48. Start by setting the numerical display mode for two decimal places:

| **Keystrokes:** | **Stack:** | |
|---|---|---|
| $\boxed{↰}$ $\boxed{\text{MODE}}$ 2 $\boxed{\text{FIX}}$ | | |
| 123 $\boxed{\text{ENTER}}$ | 1: | 123.00 |
| 45 $\boxed{\text{ENTER}}$ | 2: | 123.00 |
| | 1: | 45.00 |
| 27 $\boxed{\text{ENTER}}$ | 3: | 123.00 |
| | 2: | 45.00 |
| | 1: | 27.00 |
| 6 $\boxed{\text{ENTER}}$ | 4: | 123.00 |
| | 3: | 45.00 |
| | 2: | 27.00 |
| | 1: | 6.00 |
| $\boxed{-}$ | 3: | 123.00 |
| | 2: | 45.00 |
| | 1: | 21.00 |
| $\boxed{↱}$ $\boxed{\text{LN}}$ | 3: | 123.00 |
| | 2: | 45.00 |
| | 1: | 3.04 |
| $\boxed{\times}$ | 2: | 123.00 |
| | 1: | 137.00 |
| $\boxed{+}$ | 1: | 260.00 |
| $\boxed{\text{SIN}}$ | 1: | −0.98 |

Note how

a. each *number* entered goes into level 1, raising the preceding numbers to higher levels;

b. each *function* removes its argument or arguments from the stack, and returns a new result to the stack.

Here you can see how a *stack* provides for the realization of the principle stated at the start of Chapter 2, namely, that every result can be an argument. The stack acts as central exchange, where each function expects to find its arguments. Since each function also returns its results to the stack, those results are automatically ready to be used as arguments for the next function.

## 2.3 RPL RPN

Prior to the introduction of the HP-28C in 1987, RPN calculators provided only a limited form of RPN in which the stack was limited to four levels. This implementation is adequate for many calculations, but has certain shortcomings:

- You can't routinely convert any expression into RPN, then execute it left to right. Instead, you have to study the expression, looking for ways to avoid piling up more than four stack entries at a time.
- Some calculations intrinsically require more than four entries, no matter how clever you are. This means that you have to save one or more intermediate results in storage registers, then recover them later for further stack operations.

A four-level RPN stack is a restriction quite analogous to the limit in most "algebraic" calculators on the number of parentheses that you can nest in a calculation. Such limits are an even greater nuisance than the stack level limit, since algebraic entry does not lend itself well to passing the results of one calculation on to another.

The RPL system employed by the HP-28 and the HP-48 is a thorough implementation of RPN, in which the number of stack levels is not fixed. The stack grows and shrinks as needed. The unlimited stack allows you to concentrate on the results of a calculation without requiring extra mental effort to rearrange it to fit the constraints of a four-level stack. Furthermore, the stack is a stack of general objects, not just of ordinary numbers, so that calculations with extended objects such as matrices can be performed in the same style as simple numerical calculations.

An important example of the multi-object-type stack is RPL's ability to intermix expressions, entered in algebraic form, with RPN operations. This ability is provided through the use of *algebraic objects*, which are representations of expressions that you can enter into the stack as entire units. We discuss algebraic objects in more detail in later sections of this book; for now, you can consider them as the means by which you can calculate with algebraic notation.

In section 2.1 we showed how RPN is derived by considering the manner in which expressions are actually evaluated. However, we do not mean to imply that a completely RPN approach is always the most convenient method of calculation. In fact, to evaluate certain expressions like our example $\sin[123+45\ln(27-6)]$, it is arguably simpler to key in the expression in a manner that corresponds as nearly as possible to the written form, than to figure out the more efficient RPN keystrokes. RPN is most useful for exploratory calculation, when you're not merely evaluating a predetermined expression. RPL allows you to have the best of both worlds, by combining algebraic and RPN logic as follows:

- If you know in advance the complete mathematical form of a calculation, enter it as an algebraic object.
- If you are working out the solution to a problem, and don't know in advance all of the steps, work through the problem with an RPN approach, applying functions to the results as they appear.
- In both cases, the results are held on the stack ready for use in further calculations.

Our sample problem was originally expressed as an expression, so you can enter it as an algebraic object:

```
'SIN(123+45*LN(27-6))' [ENTER]
```

puts the algebraic object representing the expression into stack level 1. (Note that it is the expression itself that is present, not its evaluated value; the ability to handle expressions without first evaluating them is one of the unique and most powerful RPL calculator capabilities.) In this example, you are interested in the numerical value, so press [EVAL] . This replaces the algebraic object with its value −.98. Actually, if this result were all that is of interest, you could omit pressing [ENTER] , and use [EVAL] to take the expression directly from the command line and evaluate it.

Suppose, however, that at the beginning of the calculation you were only interested in the expression $123+45\ln(27-6)$. In that case, you would compute the value by entering

```
'123+45*LN(27-6)'  EVAL  ☞  260.00
```

Then, after obtaining this result, you realize that in addition to the value itself, you also need to know the sine of the value. Because the result of the initial calculation is on the stack, it is ready for further calculation. In this case, you can execute DUP to make a copy of the number for later use, then SIN to compute the sine.

RPL calculators are unique in their ability to hold the results of algebraic expression evaluation in a manner that allows you to apply additional operations to the results after they are calculated. Algebraic entry calculators require that you know the entire course of a calculation before you start; RPN calculators overcome that problem, but you must always mentally rearrange an expression into reverse Polish form as you proceed. The HP-48 allows you to proceed with any mix of the two approaches that is appropriate for the problem at hand.

# 3. Objects and Execution

In Chapter 2, we demonstrated how you perform calculations on the HP 48 by applying functions to numbers that are present on a stack, which acts as the electronic equivalent of a sheet of scratch paper. This RPN system is very uniform and flexible, and there is no particular reason to restrict its use to real numbers and ordinary mathematical functions. The HP 48 generalizes the RPN approach to problem solving in two ways:

- Real numbers are just one of several types of *objects* that the HP 48 can manipulate on the stack and store in memory. (Several other English words might be substituted for *object*; item, unit, element, etc. The use of *object* for this purpose is common in mathematical jargon, and so that word is adopted for HP 48 terminology.)
- Mathematical functions are just one of several classes of HP 48 *operations* that can be applied to numbers and other types of objects.

The terms *object* and *operation* are key terms for any discussion of the HP 48, and we will study them in detail in this chapter. In addition, we will introduce the concept of object *execution*, and the closely related term *evaluation*. In rough terms, *operations* are "what things the HP 48 can do," and *objects* are "what the HP 48 can do things *to*." *Execution* and *evaluation* are the actual "doing."

We will use these four words extensively throughout this book to make general statements about HP 48 principles, so it is important that you understand the meanings of each. If you find occasionally that the statements are too abstract, you can relate them to more familiar ideas by substituting concrete examples for the general terms. For example, when we refer to an object, you can think of a number as an example; for an operation, think of an ordinary math function like + or sine. Execution is the "activation" of an object--think of running a program. Evaluation differs from execution only for algebraic and list objects: execution treats these types of objects as data and merely returns them to the stack; evaluation actually performs sequences of calculations defined by the objects.

## 3.1 Operations

"What things the HP 48 can do" make up a very long list, and constitute the subject matter of most of this book. Here we will concentrate on defining the different types of operations, to facilitate later discussions.

We use the term *operation* to mean any of the built-in capabilities of the calculator. Most calculator manuals use the term *function* for this purpose. In describing the HP 48, the term *operation* is preferable, reserving *functions* to mean a specific group of

HP 48 operations that correspond to the mathematical meaning of *function*.

There are two basic methods by which you can make the HP 48 "do" something; that is, perform an operation.

- Find the key that is labeled with the name or symbol for an operation, and press it. Many important operations, such as the arithmetic operators, or STO and EVAL, are permanently available on the keyboard. The remaining operations are available as menu keys.
- Spell out the operation's name in the command line, then press [ENTER]. ENTER on the HP 48 plays a role that combines its original RPN calculator purpose of ending number entry with a more sophisticated meaning of "do these commands." ENTER is explored in detail in section 6.4.3.

HP 48 operations are classified as follows:

1. An operation can be a *command* or a *manual operation*, according to whether it is programmable or non-programmable, respectively. A command has a specific name, so that you can
   - execute the command by typing its name into the command line.
   - include the command in a program that you write.

   Manual operations don't have names that you can spell out or include in a program; you can only execute a manual operation by pressing a key. Examples are [ENTER], [◁] [REVIEW], and ≡SOLVR≡.
2. Programmable operations--commands--are sorted into two classes. If a command can be included in the definition of an algebraic object, it is called a *function*. Examples of functions are +, SIN, LOG, and NOT. Commands that are not allowed in algebraics are called *RPN commands*. These commands, such as DUP, STO, or RDZ (randomize), are typically stack or memory operations that make no sense in the context of an algebraic object, which is the HP 48 calculator representation of a mathematical expression or equation. The logic of expressions demands that every part of an expression (including the entire expression itself) can be evaluated to a single value. So for an HP 48 command to be included in an algebraic object, it must act like a mathematical function--use zero or more values as input, and always return exactly one result.
3. The final classification of HP 48 operations is the division of functions into two categories: *analytic* and *non-analytic*. Analytic functions are those for which the HP 48 knows the derivative and inverse. "Knowing" the inverse of a function $f$ means the HP 48 can automatically solve the equation $f(x) = y$ for $x$. (In mathematics, an analytic function is continuous and differentiable, which

corresponds more-or-less to the HP 48 meaning of analytic function. For various reasons, the HP 48 does not provide derivatives and/or inverses for every function that is analytic mathematically. % is an example of a well-behaved function for which no built-in derivative is provided. On the other hand, the function ABS can be differentiated on the HP 48, even though it is not properly differentiable at zero.)

The main reasons for sorting HP 48 operations into these categories is to make possible general statements about various classes of operations, and to provide information about individual operations without unnecessary repetition. Thus when we refer to DUP as an RPN command, we are reminding you that DUP is programmable, but not allowed in an algebraic expression.

## 3.2 Objects

The HP 48 provides 18 distinct types of objects that can be created and manipulated with ordinary built-in operations. These object types are listed by their type numbers (as returned by the commands TYPE and VTYPE) in Table 3.1. In addition, there are twelve *system object* types, including seven that are actually used by the HP 48 in internal calculations, and five provided for future extensions. You won't normally see any of these while using only built-in operations, but external software may bring them to light.

The word *object* is the collective term for all of the different items listed in the table. This list does not contain all imaginable object types; these are just the types that you can create and use on the HP 48. In the abstract, an object is a collection of data or procedures that can be treated as a single logical entity. In practical HP 48 terms, this means that an object is something that you can put on the stack.

Most objects are identified in the HP 48 by their characteristic *delimiters*, which are just the symbols #, ", ', etc., which you enter to tell the calculator what type of object you are entering, and where it starts and stops. (If you enter a string of characters without any delimiters, the HP 48 attempts to interpret it as a real number, or failing that, as a name or command.) Similarly, the calculator uses the same delimiters when it displays an already entered object so that you can recognize its type.

An individual object is characterized by its type and its value. The *type* (number, array, etc.) indicates the general nature and behavior of the object. The *value* distinguishes one object from another of the same type. For a real number object, the value is its simple numerical value. For a string, the value is the text characters in the string. For a program, the "value" is the sequence of objects and commands that make up the program. For lists, programs, and algebraic objects, which are made up of other objects, we will use the term *definition* rather than *value*.

**Table 3.1. HP48 Objects**

| TYPE Number | Object Type | Identification |
|---|---|---|
| 0 | Real number | *digits* |
| 1 | Complex number | *(real number, real number)* |
| 2 | String (text) | "*characters*" (stack or command line)<br>C$ *n characters* (command line) |
| 3 | Real array (vector or matrix) | [ *real numbers* ] |
| 4 | Complex array (vector or matrix) | [ *complex numbers* ] |
| 5 | List | { *objects* } |
| 6 | Global name | *characters*† |
| 7 | Local name | *characters*† |
| 8 | Program | << *objects* >> |
| 9 | Algebraic object | '*objects*' |
| 10 | Binary integer number | *#digits* |
| 11 | Graphics object | Graphic *n* × *m* (stack)<br>GROB *n m data* (command line) |
| 12 | Tagged object | *characters*: *object* (stack)<br>:*characters*: *object* (stack) |
| 13 | Unit object | *number_units* |
| 14 | XLIB name | *characters* (library present)<br>XLIB *n*, *m* (library missing) |
| 15 | Directory | DIR *name object* ... END |
| 16 | Library object | Library *n*: *Title* |
| 17 | Backup object | Backup *characters* |

† Names can be entered with or without ' ' delimiters. See section 3.7.

A central theme of the HP 48 is its uniform treatment of different object types. This means that the basic calculation process--applying operations to objects on the stack--is the same for every object type:

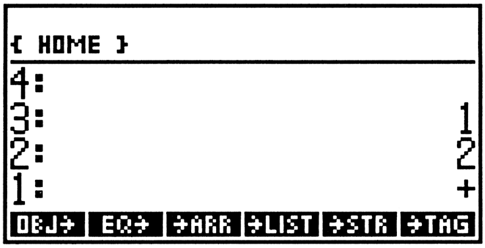
- Each stack level holds one object, regardless of type.
- The stack commands to copy, reorder, and discard objects are the same for all object types.
- The processes of storing (naming), recalling, and executing are the same for all object types.
- The same operation can be applied to as many different object types as make sense for the operation.

These points have the very practical consequence of simplifying the learning and use of the HP 48, for once you learn how an operation works for one object type, you automatically know how to use it for any other object types to which it might apply. For example, if you learn RPN arithmetic for real numbers, you don't have to learn anything new to do arithmetic with complex numbers or arrays--the steps and logic are the same. There is no such thing as "complex mode" or "matrix mode" on the HP 48.

## 3.2.1 Operations as Objects

You might ordinarily think of operations as actions, and objects as the targets or results of the actions. However, the existence of object types that are not simple data--names, algebraic objects, and programs--blurs this distinction. As a matter of fact, all HP 48 commands are just built-in program objects. To demonstrate that a command is an object, you can put it on the stack. Try this:

1 2 { + } ENTER PRG OBJ OBJ→ ⇦



In level 1, you see the *object* +. (You have to enter the + originally in a list to prevent its execution when you press OBJ→ .) If you now press EVAL , the + is executed, adding the 1 and 2 you entered previously and leaving the result 3. This technique works for any command.

This brings us to the subject of *execution*: when is an object "passive"--like the + just waiting on the stack, for example--and when is it "active"--like the + actually performing the addition?

## 3.3 Execution and Evaluation

We have generalized the concept of an object to include not only data objects but also user-defined programs and expressions, and built-in operations. We now similarly define *execution* as the general term for the *activation* of an object: to execute an object means to perform the "action" associated with that object. In the next sections, we will look at the various actions associated with the different object types.

Most object types are considered as data, for which execution simply means "put the object on the stack." Five object types have a more energetic definition of execution:

- Executing a *local name* means to recall an object stored in a local variable (section 9.7) to the stack.
- Executing a *global name* means to execute an object stored in a global variable (section 5.1).
- Executing an *XLIB name* means to execute an object stored in a library--an extension to the calculator's built-in operation set (section 5.3.2).
- Executing a *program* means to execute the objects that make up the program's definition.
- Executing a system *code object* executes the assembly language program that defines the object.

*Lists* and *algebraic objects* are defined, like programs, by a sequence of other objects (in fact, the internal structures of lists, programs, and algebraic objects are identical). Collectively, the three types of objects are called *composite* objects. The HP48 provides a second form of execution, called *evaluation*, in which composite objects of any type are executed like programs--the objects that make up a composite object are executed sequentially. For non-composite objects, evaluation and execution are synonymous.

The primary means of evaluating an object is the EVAL command, which evaluates the object in level 1, e.g.

| | | | |
|---|---|---|---|
| 3 | EVAL | ☞ | 3 |
| '1+2' | EVAL | ☞ | 3 |
| { 1 2 + } | EVAL | ☞ | 3 |
| << 1 2 + >> | EVAL | ☞ | 3 |

The use of the term *evaluation* arises from its meaning of performing the calculations represented symbolically by an algebraic expression to obtain the value of expression. In addition to EVAL, algebraic objects are evaluated by →NUM, plus several other commands that deal with expressions' values, such as ∫, DRAW, and ROOT. EVAL is the only means of evaluating a list.

## 3.3.1 When are Objects Executed?

Before studying the execution actions of the various object types, it is helpful to review the circumstances under which objects are executed or evaluated. It is not unreasonable to say that object execution takes place all the time while the HP48 is on, since virtually any HP48 activity--interpreting keystrokes, displaying objects, printing, etc.--can be viewed as the automatic execution of built-in program objects. However, of most interest are the times when objects are executed under *your* direction, particularly objects that you have created. These times are as follows:

1. *Execution*
   - When you execute ENTER (section 6.4.3), each object specified in the command line is executed, in the order in which it appears in the command line. You can *prevent* execution of names or programs in the command line by enclosing them in their respective delimiters ' ' or << >>, as discussed in sections 3.8.
   - When a program is executed, the objects that make up the program are executed, following the same rules as command line execution.
   - When a global name (section 3.6.1) is executed, the object stored in the corresponding variable is executed. (Execution of a local name merely recalls the stored object.)
   - When an XLIB name is executed, the named object in a library is executed.
2. *Evaluation*
   - EVAL removes the object in level 1 from the stack and evaluates it. This is the most common means for evaluating an object *after* it is placed on the stack.

- →NUM is similar to EVAL, except that it invokes numerical execution mode (section 3.5.5.2), and does not evaluate lists.
- QUAD, ROOT, SHOW, TAYLR, ∂, and ∫ also evaluate their stack arguments.
- HP Solve and DRAW cause evaluation of the current equation specified in the variable EQ.
- Commands such as PUT or ∫ that use a list containing real numbers as an argument numerically evaluate (→NUM) the objects in the list to convert them to real numbers.
- Program structure words such as THEN, that take a flag value from the stack, evaluate algebraic object arguments to obtain a numeric flag value.
- The *conditionals* IFT and IFTE evaluate the stack object selected by the value of the stack flag (section 7.1).

It is useful to sort HP 48 objects into three classes of objects: *data*, *name*, and *procedure*. This classification is made according to an object's behavior when it is executed or evaluated. Most types of objects are data class objects, which just put themselves on the stack when executed. The execution of name class objects (global, local, and XLIB names) causes the recall or execution of stored objects. Procedures are composite objects; their evaluation causes the sequential execution of the objects contained in the procedures.

Lists and algebraic objects classify differently depending on whether they are executed or evaluated. Because lists are primarily used as data (the contents of lists are usually not appropriate for sequential execution), we shall consider them as data class objects, which occasionally are made to act as procedures by EVAL. Algebraic objects are always suitable for evaluation, so we will consider them as procedures while keeping in mind that they act as data objects when executed.

## 3.4 Data Objects

The idea of a *data object* should be quite familiar to you, since data objects are the only quantities that can be manipulated as objects by other calculators (except for the HP 28) and BASIC computers. The archetype data object is a *floating-point real number*. More generally, an HP 48 data object is the calculator's representation of a mathematical or logical data entity such as a number, a vector, or a character string.

You would not expect a data object to be able to *do* anything; rather, it exists to have things done to it. Nevertheless, data objects do have an execution action: they just enter themselves onto the stack. When you type in a number, for example, and press $\boxed{\text{ENTER}}$, the number object is executed and so ends up in level 1. When a data object is already on the stack and you execute EVAL, nothing apparently happens. Actually, EVAL removes the object and executes it, which puts it right back on the stack. Note classifying an object as "data" does not imply that the object is small or simple--a directory is a data class object, but it can occupy any amount of memory and have a very complex structure.

The HP 48 data object class includes the following types: real number, complex number, string, real array, complex array, list, binary integer, graphics object, tagged object, unit object, directory, library, and backup object, plus all of the system object types except the code object.

## 3.4.1 Real Numbers

A real number object is the HP 48's version of an ordinary real decimal number. The number value of the object is stored in *floating-point* representation, as a combination of a 12-digit *mantissa* $(x/10^{\mathrm{IP}(\log|x|)})$ between 1 and 9.99999999999, and a 3-digit *exponent* $(\mathrm{IP}(\log|x|))$ between $-499$ and $+499$. That is, a number is represented as

$$mantissa \times 10^{exponent}.$$

When the HP 48 is in scientific number display mode (execute 12 [◁] [MODES] ≡SCI≡ ), you can see the mantissa and exponent explicitly; for example, the number $1.234\times10^{23}$ is displayed as 1.23400000000E23. The E is a one-character symbol for "×10 to the power..."

When the HP 48 performs internal calculations during the execution of mathematical functions, real numbers are expanded to fifteen-digit mantissas and five-digit exponents, and all of the calculations are carried out to that accuracy. Functions' results are rounded back to twelve-digit mantissas and three-digit exponents when they are returned to the stack. Note that this does not imply that calculations involving multiple functions are always accurate to twelve digits. The error derived from rounding intermediate results to twelve digits accumulates as each new function executes on the result of the previous one.

Real numbers are entered and displayed without any delimiters. In the command line, a real number is a consecutive sequence of decimal digits, optionally including a leading + or −, a fraction mark (decimal point), and/or an "E" followed by an optional + or − to mark the start of the exponent field.

- If you enter more than 12 digits in the mantissa, the resulting exponent will take the extra digits into account, but the mantissa is rounded to 12 digits:

9999999999999 ☞ 1.00000000000E13

- Entering more than three digits in the exponent causes a syntax error.
- In FIX display mode, real numbers displayed on the stack are shown with digit-group commas (periods when flag −51 is set). However, you can not include such commas when you enter numbers in the command line, since the commas are interpreted as object separators:

123,456,789 ☞ 123 456 789.

### 3.4.2 Complex Numbers

Complex number objects consist of two real numbers combined as an ordered pair $(x,y)$. They have two primary uses:

- To represent complex numbers, where the first number in each ordered pair is the real part of a complex number, and the second number is the imaginary part. A complex number object $(x,y)$ corresponds to the complex number $z = x + iy$, where $x = \text{Re}\,z$ and $y = \text{Im}\,z$. The object (3,2) represents the complex number $3+2i$. Complex number objects obey the rules of complex number arithmetic; for example,

  (1,2) (3,4) + ☞ (4,6).
- To represent the coordinates of points in two dimensions, such as points used in conjunction with HP48 plotting (10.3). The real part (the first number of the pair) of the complex number is the horizontal coordinate of the point, and the imaginary part (the second number) is the vertical coordinate. In this context, complex numbers act as two-dimensional vectors, and are suitable for vector addition and subtraction. However, other common vector operations, such as dot and cross products, are not defined for the complex number object type; for those purposes, you must use vector objects.

The standard entry form for a complex number is matched parentheses surrounding two real numbers $x$ and $y$: $(x,y)$. separated by spaces or commas (periods). The numbers can also be interpreted as the absolute value $r$ and phase $\theta$, by separating them with an angle sign $\measuredangle$, i.e. $(r \measuredangle\theta)$. Similarly, complex numbers by default are displayed on the stack in rectangular format, but you can obtain a polar form display by selecting polar coordinate mode (section 11.3.1).

When you enter a complex number within an algebraic object, you must separate the real and imaginary parts (or the absolute value and phase) with a comma or a semicolon. However, you can enter a complex number as an ordered pair of real numbers or symbolic quantities. In the latter case, $\boxed{\text{ENTER}}$ automatically converts the ordered pair to an explicit complex number using the symbolic constant i (section 3.5.6.1):

```
'(A,B)'      ☞   'A+B*i'
'(R, ∡θ)'    ☞   'R*COS(θ)+R*SIN(θ)*i'
```

When you are performing manual calculations with complex numbers, the 2D (*2-dimensions*) operation is a convenient tool for entering or obtaining parts of complex numbers (2D by default works with two-dimensional vectors, but you can direct it to work with complex numbers by setting flag −19). For example, if you find that you frequently forget to press $\boxed{◁}\boxed{(\,)}$ to start complex number entry, try entering the two real numbers without the parentheses and then pressing $\boxed{◁}\boxed{\text{2D}}$. 2D combines the two real numbers into a complex number, interpreting the first number as the real part (or the absolute value, in polar mode), and the second as the imaginary part (or the phase angle). Using 2D also makes it straightforward to *compute* the entries; e.g. you can enter (1,√3/2) with:

```
1  3  √  2  /  [◁][2D]   ☞   (1, .866025403785)
```

(assuming rectangular coordinate mode).

2D is its own inverse: if there is a complex number in level 1, 2D takes it apart into two real numbers, according to the current coordinate mode. There is no single programmable form of 2D; instead, the command →V2 (flag −19 set) serves to combine two real numbers into a complex number, and V→ takes a complex number apart. Both commands respect the coordinate mode, unlike R→C (*Real-to-Complex*) and C→R (*Complex-to-Real*) for which the real number arguments and results are always the real and imaginary parts of the complex number:

```
(1,2)  C→R             ☞   1  2
DEG  (1,∡45)  C→R      ☞   .707106781187  .707106781187
3  4  R→C              ☞   (3,4)
```

You can also decompose a complex number with OBJ→, which is equivalent to C→R for complex numbers.

HP48 mathematical functions treat real number and complex number objects in a very uniform manner. That is, you can intermix the two object types in almost any calculation involving arithmetic, trigonometric, logarithmic, or exponential functions. Two-argument functions return complex results if either argument is complex:

3 (2,3) * ☞ (6,9).

The result of a single-argument function may be real or complex, according to the argument type and the appropriate mathematics. The functions RE (real part), IM (imaginary part), ARG, and ABS always return real number objects. A trigonometric, logarithmic, exponential, power or root function applied to a complex argument always returns a complex results, e.g.:

(0,2) √ ☞ (1,1).

Such functions applied to real arguments may return either a real or a complex result. For example,

DEG .5 ASIN ☞ 30,

but

2 ASIN ☞ (1.57079632679,-1.31695789692).

On most other calculators, the last example would cause an error. The HP 48's integrated treatment of real and complex numbers means that you can write programs that work equally well for real and complex inputs and outputs. However, it also means that you may have to include explicit range testing in a program that you *want* to stop when a calculation strays out of the real number domain.

You should note that the last example gives the same result regardless of whether the HP 48 is in degrees mode or radians mode. Trigonometric functions consider all complex arguments and results to be expressed in radians.

### 3.4.3 Strings

*String objects* (object type 2) are character sequences that are interpreted as simple text. Strings are identified by the double quote delimiters " ". The characters within the quotes can be any HP 48 characters, including the other delimiter characters, which have no special meaning in a string. You can use string objects to prompt for input or label output, or as data to be processed logically, such as names to be alphabetized by a sorting routine (section 11.6.3). The sequence "*text*" DROP can act as a program "comment" that has no computational significance but helps you to document a portion of a program. If you write or keep programs (or any object types) on a personal computer, the comment delimiter "@" provides a better commenting method.

Strings are normally entered and edited by surrounding a sequence of characters with double quotes, e.g. "ABCDEF". However, if you want to enter a string object in which one or more of the characters are double quotes, you can use the alternate command line forms

C$ *n* *characters*
or
C$ $ *characters*

The first of these "counted string" forms makes a string object using the first *n* characters in the command line after the number *n* (not counting the first space or other non-numeric character after the *n*):

```
C$ 10 ABCD"EFGHI   ☞   "ABCD"EFGHI"
C$ 2ABC123         ☞   "BC"  123
```

When you edit a string object that contains double quote characters, it always appears in the command line in this counted string form.

The second counted string form uses all of the remaining characters in the command line following the C$ $:

```
C$ $ ABCDEFG ☞ "ABCDEFG"
```

In this case, there must be a space after the second $.

#### 3.4.3.1 Concatenation

One of the most common string operations is *concatenation*, the appending of one string to another. This is achieved on the HP 48 by the + command, which appends a string object in level 1 to the end of a string in level 2:

```
"ABC" "DEF" + ☞ "ABCDEF"
```

String concatenation does not require that both arguments of + be strings; if either argument is a string, the non-string object (unless it is a list--see section 11.5.1) is automatically converted to a string (as by →STR) and then concatenated to the other argument:

```
STD "Result=" 10 + ☞ "Result=10"
```

### 3.4.3.2 String Comparisons

String objects can be compared (ordered) by using any of the six comparison operators ==, ≠, <, >, ≤, and ≥ (section 9.3.1). Comparisons are made on a character-by-character basis, where pairs of characters are compared according to their *character codes*. The character code is a number from 0 through 255, that represents the number of a character in the ISO 8859 Latin 1 character set used by the HP48. Two strings are equal if they contain the same characters in the same order. $string_1$ is "less than" $string_2$ if the first character from the left that is not the same in both strings has a smaller character code in $string_1$ than in $string_2$. The following sequence orders two strings so that the "smaller" is returned to level 2:

```
DUP2  IF  >  THEN  SWAP  END
```

Since lower-case letters have different character codes (97–122) than upper-case letters (65-90), the ordering produced by > or < is not a case-insensitive alphabetizing.

### 3.4.3.3 Other String Manipulation Commands

The program object menu ( MTH ≣OBJ≣ ) provides commands for performing simple string manipulations.

- OBJ→ with a string argument (same as STR→) is a programmable form of ENTER, that "executes" the string object as if the string characters were entered in the command line:

  ```
  "123 456 +"  OBJ→  ☞  579
  ```

  OBJ→ is useful in programs for creating objects (like other programs) by concatenating strings representing parts of the objects.

- →STR converts any object to a string object, where the string characters represent the display form of the object:

  ```
  (1,2)  →STR  ☞  "(1,2)"
  ```

  (If the object is already a string, →STR has no effect.) Note that since STR→ respects the current number display modes, the combination →STR OBJ→ does not necessarily leave an object unchanged unless the current number display mode is STD, and the binary integer wordsize is 64 bits.

- SIZE returns the number of characters in a string:

  ```
  "ABCDEFG"  SIZE  ☞  7.
  ```

- POS (*POSition*) finds the position of one string (level 1) within another (level 2):

  "ABCDEF" "CDE" POS ☞ 3.

  The position is counted from the left, starting with the first character as position 1. POS returns 0 if the second string is not contained within the first.

- REPL (*replace*) overwrites a portion of a string (level 3) with another string (level 1), starting at a specified position (level 2). Call the target string *string*$_1$ (length $l_1$), the replacement string *string*$_2$ (length $l_2$), and the position $n$. Then for

  $n > l_1$, *string*$_2$ is concatenated to *string*$_1$:

  "ABCDE" 10 "FG" REPL ☞ "ABCDEFG"

  $n + l_2 - 1 \leq l_1$, characters $n$ through $n + l_2 - 1$ are replaced; the remaining $l_1 - l_2$ characters in *string*$_1$ are unchanged:

  "ABCDE" 2 "FG" REPL ☞ "AFGDE"

  $n + l_2 - 1 > l_1$, characters $n$ through $l_1$ are replaced, and the leftover $l_2 - (l_1 - n)$ characters from the end of *string*$_2$ are concatenated, so that the result string has $n + l_2 - 1$ characters:

  "ABCDE" 5 "FG" REPL ☞ "ABCDFG"

  $n = 0$, the Bad Argument Value error is reported.

- SUB extracts a substring from a string (level 3), where the start and end character positions are specified in level 2 and level 1:

  "ABCDEFG" 3 7 SUB ☞ "CDEFG"

  A character position argument less than 1 is treated the same as 1; a position greater than the string length is treated as that length. A null string is returned if the specified end position is less than the start position.

- NUM returns the *character code* of the first character in a string:

  "ABCDEF" NUM ☞ 65

- CHR produces a one-character string, where the character is specified by its character code:

  189 CHR ☞ "½"

CHR provides the only means of entering certain seldom-used characters, such as the ½ shown in the example, that are not available on the keyboard.

### 3.4.4 Arrays

Array objects (object types 3 and 4) are the HP48 representation of real or complex vectors (one-dimensional arrays) and matrices (two-dimensional). Arrays are identified in the command line and in the stack display by the square-bracket delimiters [ ]. A sequence of numbers surrounded by a single pair of brackets is a *vector*. A sequence of vectors surrounded by an additional pair of brackets is a *matrix*, where each vector is one row of the matrix.

Arrays can be either real (type 3) or complex (type 4). In a real array all of the elements are real numbers; in a complex array the elements are complex numbers. As in the case of number (scalar) objects, you can intermix real and complex arrays in calculations. You can also combine numbers and arrays for many operations, where it makes mathematical sense. For example,

```
2 [1 2] * ☞ [2 4].
```

However, you can't add a number to an array, since that is not a mathematically defined operation.

Arrays are discussed at more length in Chapter 11.

### 3.4.5 Lists

A *list* object (type 5) consists of a series of any types of objects entered between { } delimiters. The primary purpose of lists is to allow two or more objects to be manipulated together as a single data object. Lists are described in detail in Chapter 11, and are used in numerous program examples throughout this book.

### 3.4.6 Binary Integers

*Binary integer* objects represent unsigned integer numbers, stored as sequences of binary bits (rather than decimal digits as for floating-point numbers). The maximum value of a binary integer is the hexadecimal number FFFFFFFFFFFFFFFF, corresponding to 64 binary 1's.

In addition to their immediate use for performing integer arithmetic, binary integers are used in the HP48 for

- a modest set of bit-shifting and logic commands common to computer science applications, provided in in the base menu ( MTH ≣BASE≣ );

- encoding the user and system flags (section 7.1);
- representing graphic object pixel numbers (section 10.3);
- computing object checksums (section 12.5.1).

For the four arithmetic operations, you can intermix binary integer and real number arguments--the results will be binary integers.

You can control the entry and display of binary integers by executing one of the base mode commands BIN (binary, base 2), OCT (octal, base 8), DEC (decimal, base 10) or HEX (hexadecimal, base 16). To enter a binary integer, type the # delimiter followed by the number digits. The digits are interpreted according to the current base; in hexadecimal mode, for example, you can use digits 0-9 and A-F. You can override the current base by adding a lower-case letter b, o, d, or h immediately after the number digits. The objects are always displayed in the current base, including the trailing letter that identifies the base, regardless of how they were entered.

When a binary integer is entered, it is always created with 64-bit precision. However, integer operations and display are limited by the current *wordsize*, a number from 1 through 64 (the default is 64). STWS sets the wordsize from a real number argument; RCWS returns the current wordsize as a real number. The stack display of binary integers shows only the least significant *wordsize* bits, e.g.

```
HEX  10  STWS  #FFFFh  ☞  #3FFh.
```

At this point, the number has not actually been truncates to 10 bits--if you execute 64 STWS you will see #FFFF. However, all arithmetic and logical commands that work with binary integers truncate their arguments to the current wordsize before performing their operations, and return results truncated to the wordsize. If you multiply the #FFFh above by 1, then set the wordsize to 64, you will see #3FF, since the multiplication truncated the arguments and results. The truncation actually shortens the binary integer to the specified number of bits, rather than just setting the most significant bits to zero:

```
12  STWS  #FFFh  DUP  1  *  ☞  #FFFh  #FFFh.
```

Here we have two binary integers with the same numerical value. However, BYTES (section 12.5.1) applied to those two arguments returns memory sizes differing by 6.5 bytes (and different checksums), showing that one is 52 bits (6.5×8) longer than the other.

## 3.4.7 Graphics Objects

A *graphics object* (object type 11), or *grob* for short, encodes a display picture. It is defined by its *dimensions*--width × height--and the picture data. The data consists of one binary bit for each pixel, where 1 is "on" and 0 is "off", plus some additional bits that pad the data so that each pixel row is an integer number of bytes. Grobs are not restricted to the 131×64 pixels display size--they can range from 1×1 (actually, you can make a 0×0 grob, but it has no particular use).

Graphics objects are most frequently created by an operation such as DRAW, but you can create them in the command line. The command line format is

GROB *width height* · · · *data* · · ·

| | |
|---|---|
| GROB | is the "delimiter" that identifies the start of a graphics object. |
| *width* | is a real number indicating the horizontal width of the grob, in pixels. |
| *height* | is a real number indicating the vertical height of the grob, in pixels. |
| · · · *data* · · · | is a sequence of hexadecimal digits 0-F that represent the pixel data in a "readable" form. |

The readable data consists of the data for each pixel row concatenated together into one long sequence, in top-to-bottom order. Each hexadecimal digit represents four pixels; if you consider a digit as a four-bit binary number, you can translate its value to a left-to-right pixel pattern by reversing the order of the bits. The digit C, for example, represents the pixel pattern 0101, where 0 is an "off" pixel, and 1 is "on." The last one or two digits in each row may be "padded" with zeros, in order to make each row an integer number of bytes. Thus the smallest grobs are GROB 1 1 00 and GROB 1 1 10, which are 1×1 grobs--the first has its one pixel off, and the second has its one pixel on.

There are a wealth of operations related to the creation and manipulation of graphics objects. These are described in section 10.3.

## 3.4.8 Tagged Objects

*Tagged objects* (object type 12) are objects used for putting visible labels on stack objects. That is, a tagged object contains a single object of any type together with a character string that labels the object. In our discussions of tagged objects, we'll use the following terms:

- A *tag* is any character string. To *tag* an object is to combine it with a tag into a tagged object.

- An *untagged object* refers to the object inside a tagged object, when it is thought of as a separate object.
- A *tagged object* is then an object that contains a tag and an untagged object.

Thus for :ABC:12345, the tag is ABC, the untagged object is the real number 12345, and the combination :ABC:12345 is the tagged object. This terminology may be confusing, but fortunately the design of tagged objects is such that you can generally use an object in calculations with or without a tag, disregarding the distinctions.

When a tagged object appears on the stack, it is displayed as *tag: object*, where *tag* is the label string, and *object* is the usual display of the object. You can create tagged objects in the command line by typing the tag string, surrounded by : : delimiters, followed by the tagged object in its ordinary syntax:

```
:Result: 1.234
```

(there can be any number of spaces or other separators between the tag and the object). The colons act as start and end delimiters for the tag string; between the colons you can include any other characters including spaces. When a tagged object is displayed by itself in a stack level, the leading : is not shown to make a more visually pleasing label, but both colons are required in command line entry to mark the start and end of the tag. Note that tags longer than 17 characters are not particularly useful, since 17 characters is the longest tag that can be displayed including the final :.

You will find that direct command line creation of tagged objects is less common than their automated construction in programs using →TAG. The HP48 creates tags itself in some cases; HP Solve tags its results, as does LR and certain plot environment operations. Similarly, programs you write can attach identifying tags to results (see also section 12.7.1). For example, this simple program computes and labels the volume of a box from three numbers on the stack:

```
<< * * "Volume" →TAG >>
```

→TAG tags an object in level 2 with a tag formed from a string or global or local name in level 1.

You can remove the tag(s) from an object either with OBJ→, which splits a tagged object into the untagged object (to level 2) and the tag string (level 1), or DTAG, which strips any and all tags from an object. [Since a tagged object is an object, it can be tagged itself, so that a tagged object can effectively have multiple tags. OBJ→ splits off one tag at a time; DTAG strips all tags, returning only the innermost untagged object.]

The beauty of tagged objects is that normally you don't have to worry about stripping tags: *a tagged object can be used as an argument for any operation that works with its untagged object*. Most operations apply directly to the untagged object, automatically stripping its tags (including multiple tags). The only exceptions to this rule are:

- Stack operations that just move or duplicate objects on the stack treat tagged objects like any other object, leaving the tags intact.
- STO of a tagged object into a local variable or a backup object does not strip tags (STO into a global variable does strip tags).
- SAME (section 9.3.2) includes tags when comparing two objects.
- OBJ→ and DTAG remove tags.
- TYPE returns type 12 for tagged objects.

These properties mean that you can use a tagged object interchangeably with an untagged object of the same type as the tagged object. For example,

```
:Length:10_m  :Area:100_m^2  *  "Volume"  →TAG  ☞  :Volume:1000_m^3
```

Here the * automatically strips the tags from the the length and area values before multiplying them to obtain the volume.

Further illustrations of the uses of tagged objects are given in section 12.7.1.

### 3.4.9 Unit Objects

*Unit objects* (object type 14) are the basic components of HP 48 *unit management*--its ability to perform mathematical operations on quantities that include physical dimensions. Unit management is discussed in *Part II.*

A unit object consists of a *magnitude* and a *unit expression* joined by the delimiter _ in the format *magnitude_expression*. The magnitude is a real number; the unit expression is an algebraic expression consisting of products of unit names raised to various powers. If any of the powers are negative, the expression is defined as a single numerator that is the product of names with positive powers, divided by a denominator that is the product of the names with negative powers, expressed then with positive exponents. For example, $1m^2s^{-2}K^{-1}$ is represented by the unit object 1_m^2/(s^2*K). Because there is no closing delimiter on the unit expression, you must enter the expression immediately after the _, and it may contain no spaces (there can be spaces between the magnitude and the _).

Unlike other object delimiters, the underscore _ is also a *function*. This allows the straightforward use of the EquationWriter for unit object entry. _ takes two arguments, which may be real numbers, names, or algebraic expressions. For most argument combinations, _ is equivalent to multiplication (*). But when the second argument is a name or an algebraic, it is converted to a unit object before multiplication by the first argument. Thus

```
2  3  [ENTER]  _   ☞  6
6  1_cm  _         ☞  6_cm
12  'X'  _         ☞  12_X
1  'm-cm'  _       ☞  99_cm
```

In the first example, the extra [ENTER] (other than that implied by the ☞) is necessary because if _ is preceded by a real number in the command line, it is taken as a delimiter and must be immediately followed by a unit expression. The last example illustrates the conversion of an algebraic object into a unit expression: all names in the object are converted to unit objects of magnitude 1, then the expression is evaluated. The result is multiplied by the first argument.

## 3.4.10 Directories

A *directory* (object type 15) is an object that contains a sequence of *global variables*-- name/object pairs. A full explanation of the nature and properties of directories is given in section 5.2; here we just note that a directory is a data-class object, meaning that it can be recalled to the stack, copied, and stored. As data-class objects, their execution action is just to return themselves to the stack. (These are enhancements over the HP28, where directories were also objects, but no provision was made for manipulating them as objects.)

The command line and display form of a directory is

DIR $name_1$ $object_1$ · · · $name_n$ $object_n$ END

where DIR and END act as start and end delimiters. Each $name_1$ $object_1$ pair specifies a variable. The order of the variables is the same as they appear in the VAR menu.

## 3.4.11 Libraries

A *library* object (object type 15) is similar to a directory object, in that it contains a sequence of named objects (*library commands*). However, unlike a directory, a library has a fixed internal structure, so that you can not edit it.

- In a library, the object names are separated from the objects into a table, providing faster access by name to the objects than in a directory.
- Depending on the origin of a library, it may contain nameless or other special system objects. There is no provision on the HP48 for displaying the contents of a library, other than the LIBRARY menu (section 5.3.1.1), which displays a library's commands.
- All objects in a library are uniformly accessible--there is no sub-library structure analogous to subdirectories in a directory.

The named objects or *library commands* within a library are extensions to the built-in command set, and can be used in the same manner. A library is an object so that it can be transferred from calculator to calculator or between calculator and personal computer, moved between the ports (section 5.3), or stored in an inactive form in a variable. When a library is displayed as an object, it appears as Library *n*: *title*, where *n* is a decimal number that identifies the library, and *title* is a descriptive text string. You can't see more than a few characters of a library title when the library is on the stack, but you can use ⬅ EDIT or ▽ to view all of the title in the command line (you should cancel the edit with ATTN rather than using ENTER, since you can't actually edit a library).

A library's commands are executed by means of *XLIB name* objects, which are described in section 3.6.3. The methods of *attaching* libraries to directories so that their XLIB names are usable is described in section 5.3.2.1.

[As a matter of fact, built-in commands are also contained in libraries. Because they are permanently located at fixed memory addresses, the commands can be represented on the stack by pointers to the objects rather than by XLIB names.]

### 3.4.12 Backup Objects

A *backup object* is the object form of a *variable* (section 5.1), in that it contains a single object of any type plus a name. As an object it is mobile and can be copied or stored, unlike a variable, which is not an object but is a part of a directory. A backup object also contains a checksum that is used by the HP48 to verify its memory integrity when it is transferred between main memory and plug-in memory.

If a backup object is stored in a port (section 5.3.1), the object it contains can be accessed in a manner similar to an object stored in a global variable. Such backup objects are addressed by means of global names tagged with a port number. Normally, a backup object is created directly in port memory, so that you will seldom see backup objects on the stack--the primary focus is on the object stored within the backup object. Backup objects on the stack appear as Backup *name*, where *name* is the backup object's name. You can not create or edit a backup object in the command line.

## 3.5 Procedure Objects

In the preceding review of data class objects, the concept of object execution is straightforward but not very interesting. Indeed, there is little point in executing a data object (with EVAL, for instance) once it is on the stack; the main point of executing such objects derives from their behavior when executed indirectly during the execution of a name or a procedure.

In most calculators, a program is a series of numbered steps that are executed in numerical order, with occasional breaks in the sequence caused by GOTO instructions or subroutine calls. Each step in such programs either enters data, or performs a built-in command. The step numbers indicate the order of execution, but they really have no meaning other than for visual reference, or in some cases as labels for GOTO. The HP 28/HP 48 replacements for the conventional calculator program are *procedure class objects*. A *procedure* is an object defined to be a series of other objects intended for sequential execution. The procedure class of objects includes program objects, algebraic objects and code objects (lists can also act as procedures-- see section 3.5.3).

### 3.5.1 Program Objects

An HP 48 *program object* (object type 8) is similar to a conventional program in that it contains a sequence of "steps". The steps are either objects themselves, or combinations of objects called *program structures*; together, the steps are called the *program definition* or *program contents*. Execution of a program object causes execution in turn of each object in its definition. Program structures such as *branches* and *loops* (section 9.2) can alter the order of execution beyond simple linear sequences.

A program object is identified by its start- and end-delimiters << >>. Objects entered between the delimiters make up the program's definition. Note that a program, like any other object, has no intrinsic name. You name a program by storing it in a named variable.

### 3.5.2 Algebraic Objects

An *algebraic object* (object type 9) is also a procedure-class object, but it resembles a conventional program even less than a program object does, since it is displayed as an algebraic formula. The delimiters for algebraic objects are the single quotes (usually called "ticks", for short) ' '; the objects that make up the algebraic object's definition are entered between the quotes.

Algebraic objects have internal structures indentical to programs, but they differ in these respects:

- Programs can contain any HP 48 objects; algebraics can contain only numbers, unit objects, names, and the subset of HP 48 commands identified as *functions*.
- The objects in a program may appear in any combination, and may be grouped into structures (section 9.2). In an algebraic object, the objects are always organized according to specific rules, called *algebraic syntax*, that insure that the object looks and behaves like a mathematical formula.
- For programs, execution and evaluation are synonymous. The execution action of a program is to execute the contents of the program sequentially. For algebraic objects, *execution* treats the objects as data objects, returning the unchanged object to the stack. *Evaluation* of an algebraic object treats the object as a program, and executes the objects that define the algebraic object.
- Evaluation of a program may take any number of other objects from the stack, and return any number of arguments, depending on the program definition. Evaluation of an algebraic object normally takes no arguments from the stack, and returns one result. (This general rule can be broken if any of the names within the algebraic object correspond to program variables; execution of those names causes *execution* of the programs, which may have arbitrary stack effects.)

The ability of algebraic objects to act as data when executed, or as programs when evaluated, is one of the foundations of the HP 48's ability to perform symbolic mathematics. When you rearrange a formula using mathematical rules, you are treating it as data; when you perform substitution of variables' values for their names, you are evaluating the formula.

In section 2.1, we showed how RPN logic is derived from the desire to convert a mathematical expression into a series of steps by which you can evaluate the expression by hand or using a machine. Looking at this from a different point of view, you can note that since any expression can be translated to RPN, any expression can be represented in a calculator by an RPN program. In fact, this is what the HP 48 does--an algebraic object is stored in calculator memory in an RPN program form just like that of an actual program object. The HP 48 saves you from having to do the conversion yourself by providing the algebraic object type.

The only difference between algebraic objects and program objects is that the two are "marked" differently, so that the HP 48 knows which to display in algebraic form and which to display in RPN. Also, functions that accept symbolic arguments can only accept algebraic objects, not programs, since algebraics are by definition valid mathematical expressions, whereas program objects are completely unrestricted in their content and may not be suitable arguments for a mathematical function.

To illustrate the program nature of algebraic objects, create this program B:

```
« DUP 20 » 'B' STO
```

Next, enter the algebraic object '5+5+B', and press $\boxed{\text{EVAL}}$. The algebraic object disappears, and the numbers 10 and 30 appear on the stack. You can understand this result by following the execution of the equivalent RPN sequence 5 5 + B +. When this sequence is executed, two 5's are entered, then summed to 10 by the first +. B executes next, which duplicates the 10 and enters 20. Then the final + executes, returning 30. You can break down any algebraic object execution into RPN steps this way. Knowing how algebraic evaluation works is the key to understanding some of the subtleties of symbolic operations on the HP48 in general.

Picturing an algebraic object as a program will also help you understand why evaluation of the object causes variable substitution "one level at a time." Consider the object 'A+B', where A has the value 10, B has the value 'C+D', C has the value 20, and D has the value 30. Evaluating 'A+B' once does one level of substitution, returning '10+(C+D)', not the numerical result 60. To see why, remember that 'A+B' is represented by the sequence A B +. Evaluating 'A+B' therefore executes A, B, and + in sequence: A returns 10, then B returns 'C+D', so that + returns '10+(C+D)'. [Note that the latter in RPN is 10 C D + +, which is obtained from the original A B + by substituting the RPN sequence C D + for B.]

These considerations also explain why you might get unexpected objects on the stack when an error occurs during evaluation of an algebraic object. For example, if you execute EVAL on an algebraic object and an error occurs, you might expect that the original object would be returned to the stack. But evaluating an algebraic object is the same as executing a program, so that an error returns the arguments of whatever function (within the algebraic) caused the error, along with anything else that was on the stack at the time of the error. Again, you can predict the contents of the stack from the RPN sequence that is equivalent to the algebraic object.

For example, suppose you execute 'A+(B+C)' EVAL, where A and B are undefined, but C has a vector value [1 2]. The HP48 will halt and show the Bad Argument Type error message, with the stack containing

```
3:       'A'
2:       'B'
1:     [ 1 2 ]
```

This configuration results because the RPN sequence A B C + + errored at the first +. A, B, and C had already executed, leaving their values on the stack as shown; the +

errored because the combination of a name ('B') and a vector ( [ 1 2 ] ) is not valid for addition. These arguments of +, not the original argument of EVAL, are returned to the stack. (Note that if you execute EVAL by using the EVAL key, you can restore the original algebraic object by pressing ⬅ LAST STACK .)

### 3.5.2.1 Expression Structure

One advantage of writing a mathematical expression in Polish notation (section 2.1) is that it makes explicit the organization of the expression into a hierarchy of *subexpressions* (section 3.5.2.1). For example, consider the expression $a + \sin(b-c)$. Rewriting this in Polish form, you obtain $+(a, \sin(-(b,c)))$. The "outermost" subexpression is the entire expression, consisting of the function + and its arguments $a$ and $\sin(-(b,c))$. Each of the two arguments is a subexpression--the first is just the name $a$, the second is the function sin and its argument $-(b,c)$. The latter in turn is a subexpression consisting of $-$ and its arguments $b$ and $c$, and so on as you peel off the layers of parentheses. The *level* of a subexpression is a measure of how deep it is in the hierarchy. The level is defined as the number of pairs of parentheses that surround the subexpression. In the example, the full expression is level 0; the $a$ and $\sin(-(b,c))$ are level 1 subexpressions, $-(b,c)$ is level 2, etc.

There are two reasons for you to keep these ideas of expression structure in mind as you work with the HP 48:

1. The structure of an expression determines the order of evaluation of its subexpressions. For example, in the evaluation of 'A+B+C', the A and B are added first, then the sum is added to C. You can alter this order by changing the expression to 'A+(B+C)', in which case the B and C are added first. This distinction is important in a floating-point calculator, even though the two forms are formally the same. To see this, assign the values $10^{50}$ to A, $-10^{50}$ to B, and 1 to C. If you evaluate 'A+B+C', you obtain 1, whereas if you evaluate 'A+(B+C)', you obtain 0.
2. Understanding the structure of an expression can help you follow the behavior of HP 48 symbolic manipulation commands. For example, EXPAN is defined to work at one level of a subexpression at a time. 'A*(B+C+D)' EXPAN returns 'A*(B+C)+A*D' rather than 'A*B+A*C+A*D' as you might expect. This is more obvious if you think of the original expression as *(A,+(+(B,C),D)). When one of the arguments of * is a sum, EXPAN multiplies the other argument by each of the two arguments of +, then adds the products. The fact that in this case the first argument of the (first) + is also a sum is not considered--EXPAN only works one level at a time.

We can use these ideas to re-express the basic RPN calculator principle ("*any result can* *be an argument*") in "algebraic" terms by saying "*any expression can be a subexpression.*" A subexpression is self-contained; it may or may not be embedded in a larger expression. The shortcoming of algebraic calculators is that they don't recognize this principle. They are designed for evaluating an expression as a whole--"from the outside in," so to speak. On the other hand, in a purely RPN calculator like the HP 41, you can only calculate an expression "from the inside out," since you can only enter one number or function at a time. The HP 48 merges both approaches, by allowing you to enter any subexpression in its algebraic form. You can evaluate an entire expression at once, or you can divide it into subexpressions of any size, or you can work only with one object at a time.

As with most of the principles of HP 48 operation, the concept of algebraic object evaluation is derived from a mathematical model. In ordinary terms, to "evaluate" means "to find the value." For a mathematical expression, this translates to "perform the operations represented by the expression, to find its value." Evaluation means to "activate" an expression, which in turn means to execute sequentially the objects that make up the expression.

As an example, consider the simple expression 1+2. We showed in section 2.1 that an expression can be translated into an RPN form that represents a prescription for actually performing the operations of the expression--evaluating it. Thus the expression 1+2 is the sequence 1 2 + in RPN. This is a sequence of *objects*--remember (see section 3.2.1) that the +, as well as the 1 and the 2, can be considered as an object. When you write the expression, the objects are passive; but if you execute each object in succession--"enter the 1, enter the 2, do the +"--you obtain the value of the expression.

### 3.5.3 Lists as Procedures

As mentioned in section 3.5.3, lists are composite objects with internal structures like programs and algebraic objects. As such, they can be evaluated as programs. The only commands on the HP 48 that treat lists as procedures are EVAL (section 3.9), IFT and IFTE (section 9.4.2). The principal reasons for providing list procedure evaluation in this manner is to permit the construction of new procedures by programs, and to facilitate changing directories by executing path lists (section 5.5.3). This form of list evaluation is not available on the HP 28, where lists are strictly treated as data-class objects.

### 3.5.4 Commands and Functions

As illustrated in section 3.2.1, HP 48 commands are objects. Because there is a permanent binding between command objects and their command names, command objects are always entered and displayed using their names only--you never see the actual built-in SIN program, for instance, only the name SIN. Actually, all commands are program objects, but to help a program distinguish between commands and user-created programs, the TYPE and VTYPE commands return type 18 or 19 for commands rather than type 8 (program). Type 18 indicates that the command is a *function*; type 19 indicates an RPN command.

In most calculators, there is a distinction between user-written programs and built-in commands:

- *Programs* are written in the user programming language, and are executed by means of a command like RUN, XEQ, GOSUB, etc., combined with a program name or label number. Programs can call other programs (subroutines), but there may be a restriction on the number of pending returns of which the calculator can keep track (six in the HP-41, for example).
- *Commands*, on the other hand, are executed or entered into a user program by name, with no prefix command. In most calculators, executing a command by name consists of pressing the key that has the command name on it. This either executes the command, or enters a function code or the name itself into a program. Some calculators have an alphabetic keyboard that allows you also to specify a command by spelling out its name.

The fact that calculator commands themselves are just internal programs is not readily apparent. The programs can't be viewed or edited, and they are written in the calculator's assembly language, which would require much more information for the typical user to understand and apply than can be provided in owners' manuals.

The HP-28/48 philosophy is that the distinction between user programs and built-in commands is artificial and unnecessary, at least as regards their use from the keyboard and as subroutines. That is, when you write a program and name it, you should be able to use it exactly as if it were a built-in command. When you enter a program name into the command line and press $\boxed{\text{ENTER}}$, or include a program name in another program definition and execute the latter program, or just press a menu key labeled with the program name--the program should execute. The central idea underlying the execution of HP48 name objects follows from these ideas (section 3.6).

## 3.5.5 Function Execution

HP48 functions have two important execution properties that are not shared by RPN commands. These are *automatic simplification*, and a choice of *symbolic* and *numerical* execution modes.

### 3.5.5.1 Automatic Simplification

When certain functions execute, they check their arguments for special cases in which ordinary calculation can be replaced by a mathematical simplification. For example, if you execute the sequence 1 'X' *, you obtain 'X', not '1*X'. You can observe the same effect by executing '1*X' EVAL. This simplification is a property of the * function; when it is executed, * explicitly looks for cases where one of its arguments is 1. In such cases, the subexpression consisting of the * and its two arguments is automatically replaced by the non-1 argument. Other examples are the replacement of SIN(ASIN(X)) by X, and EXP(LN(X+1)) by X+1. Again, these simplifications are built into the functions SIN and EXP. Table 3.2 is a complete list of automatic simplifications built into the HP 48.

Note that not all cases of a function applied to its own inverse are simplified. For example, ASIN(SIN(X)) does not automatically simplify to X, since there are infinitely many angles with the same sine as X. Similarly, since the HP 48 treats complex numbers uniformly with real numbers, LN(EXP(X)) does not reduce to X.

Automatic simplification is not the same as the simplification that results when a numerical expression is evaluated by COLCT. For example, although '2/2' automatically simplifies to 1 when you evaluate it, '2*X/2' does not automatically simplify to X. In order for the simplification to take place, the two 2's must be the arguments of the /, as in '(2/2)*X'. To simplify '2*X/2', you can either use RULES to rearrange it to '(2/2)*X', or use COLCT.

#### 3.5.5.2 Symbolic and Numerical Execution; →NUM

The key to the HP 48's ability to perform symbolic calculations is the fact that HP 48 functions used with symbolic arguments (names or algebraics) return symbolic results. Each time you evaluate an algebraic object, the names in the expression or equation are executed, so that those corresponding to existing variables are replaced by the objects stored in the variables. But the replacement objects are *not* evaluated, so that the final result may still be symbolic. If you want to evaluate a symbolic object all the way to a numerical value, you may have to use EVAL repeatedly until all of the names have been replaced by numbers.

In some circumstances, it is desirable to evaluate a symbolic object to its final numerical value in a single operation. For example, in the course of their execution, DRAW and HP Solve both evaluate the current equation to numerical values. To deal with such cases, as well as the symbolic evaluation described already, the HP 48 provides you with the choice of *symbolic execution mode* or *numerical execution mode*. In symbolic execution mode, a function evaluated with symbolic arguments returns a symbolic result. In numerical execution mode, a function of symbolic arguments evaluates its arguments, repeatedly if necessary, until they are data objects (usually numbers). Then the function returns a numerical result. If any name is encountered during the evaluations that has no corresponding variable, the Undefined Name error is returned.

**Table 3.2. Automatic Simplification**

| *Addition and Subtraction* | | |
|---|---|---|
| X−X | ☞ | 0 |
| 0+X | ☞ | X |
| (0,0)+X | ☞ | X |
| 0−X | ☞ | CHS(X) |
| (0,0)−X | ☞ | CHS(X) |
| X+0 | ☞ | X |
| X+(0,0) | ☞ | X |
| X+ −$p$ | ☞ | X−$p$ |
| X−0 | ☞ | X |
| X−(0,0) | ☞ | X |
| X− −$p$ | ☞ | X+$p$ |

| *Multiplication and Division* | | |
|---|---|---|
| INV(i) | ☞ | −i |
| Y*INV(X) | ☞ | Y/X |
| Y/INV(X) | ☞ | Y*X |
| 0*X | ☞ | 0 |
| (0,0)*X | ☞ | (0,0) |
| i*i | ☞ | −1 |
| 1*X | ☞ | X |
| (1,0)*X | ☞ | X |
| (−1)*X | ☞ | CHS(X) |
| (−1,0)*X | ☞ | CHS(X) |
| X*0 | ☞ | 0 |
| X*(0,0) | ☞ | (0,0) |
| X*1 | ☞ | X |
| X*(1,0) | ☞ | X |
| X*(−1) | ☞ | CHS(X) |
| X*(−1,0) | ☞ | CHS(X) |
| X/1 | ☞ | X |
| X/(1,0) | ☞ | X |
| X/(−1) | ☞ | CHS(X) |
| X/(−1,0) | ☞ | CHS(X) |
| 0/X | ☞ | 0 |
| (0,0)/X | ☞ | (0,0) |
| 1/X | ☞ | INV(X) |
| (1,0)/X | ☞ | INV(X) |
| (−1)/X | ☞ | −INV(X) |
| (−1,0)/X | ☞ | −INV(X) |

| *Powers* | | |
|---|---|---|
| 1^X | ☞ | 1 |
| (1,0)^X | ☞ | (1,0) |
| SQ(√(X)) | ☞ | X |
| SQ(Y^X) | ☞ | Y^(2*X) |
| SQ(i) | ☞ | −1 |
| X^0 | ☞ | 1 |
| X^(0,0) | ☞ | (1,0) |
| X^1 | ☞ | X |
| X^(1,0) | ☞ | X |
| X^(−1) | ☞ | INV(X) |
| X^(−1,0) | ☞ | INV(X) |
| | | (√X)^2 |
| (√X)^(2,0) | ☞ | X |
| i^2 | ☞ | −1 |
| i^(2,0) | ☞ | (−1,0) |

| *Parts* | | |
|---|---|---|
| ABS(ABS(X)) | ☞ | ABS(X) |
| ABS(CHS(X)) | ☞ | ABS(X) |
| CONJ(CONJ(X)) | ☞ | X |
| CONJ(IM(X)) | ☞ | IM(X) |
| CONJ(RE(X)) | ☞ | RE(X) |
| CONJ(i) | ☞ | −i |
| IM(CONJ(X)) | ☞ | −IM(X) |
| IM(IM(X)) | ☞ | 0 |
| IM(RE(X)) | ☞ | 0 |
| IM(*p) | ☞ | 0 |
| IM(i) | ☞ | 1 |
| MAX(X,X) | ☞ | X |
| MIN(X,X) | ☞ | X |
| MOD(0,X) | ☞ | 0 |
| MOD(X,X) | ☞ | 0 |
| MOD(X,0) | ☞ | X |
| X MOD Y MOD Y | ☞ | X MOD Y |
| RE(CONJ(X)) | ☞ | RE(X) |
| RE(IM(X)) | ☞ | IM(X) |
| RE(RE(X)) | ☞ | RE(X) |
| RE(π) | ☞ | π |
| RE(i) | ☞ | 0 |
| SIGN(SIGN(X)) | ☞ | SIGN(X) |

X, Y are any subexpressions.
$p$ is any positive real number.

You can select numerical execution mode temporarily, for a single evaluation of a symbolic object, or for an indefinite period:

- To evaluate numerically a single object containing functions, use →NUM instead of EVAL. →NUM enables numerical execution mode, evaluates its argument in the same manner as EVAL, then restores the original execution mode.
- To select numerical execution mode "permanently," set flag −3. The [◁] [MODES] menu key ≡SYM≡ is handy for this purpose; pressing that key toggles between symbolic and numerical execution modes. If the key label shows a white box ( ≡SYM□≡ ), then flag −3 is clear and symbolic execution is active; the absence of the white box indicates that numerical execution is in effect. You can also set and clear the flag with SF and CF (in the second page of the [▷] [MODES] menu). While flag −3 is set, the execution of any function returns a numerical result, or an error message if numerical execution fails. In this mode, EVAL and →NUM produce the same results. To restore symbolic execution mode, press ≡SYM□≡ or clear flag −3. Symbolic execution mode is the default mode following a memory reset (section 5.8).

To illustrate these ideas, execute

30 'X' STO 'X'

to create a variable X with the value 30, and leave its name on the stack. Next select degrees mode by executing DEG ( [◁] [MODES] ≡DEG≡ ) if necessary. Now,

1. In symbolic execution mode, compute the sine:

   [SIN] ☞ 'SIN(X)'.

   At this point, you still have a symbolic result. Find the numerical value:

   [EVAL] ☞ .5.

   When 'SIN(X)' is evaluated, X is replaced by its value 30; then, since SIN has a numerical argument, a numerical result is returned.

2. Now try the calculation in numerical mode:

   [◁] [MODES] ≡SYM□≡ 'X' [SIN] ☞ .5

   This time, you immediately obtain the numerical result .5. This is because in numerical execution mode, SIN evaluates the symbolic argument 'X' to its value 30, then returns the numerical $\sin 30°$.

### 3.5.6 Symbolic Constants

A frequently asked questions about HP calculators is "why does the sequence π SIN (in radians mode) *not* return 0, when everybody knows that $\sin\pi = 0$?" On the HP-41, for example, π SIN returns -4.1E-10. The answer is that the π key does not return *mathematical* π, but an approximation accurate to the numerical precision of the calculator, which is the 10 digit number 3.141592654 on the HP-41. When SIN uses this approximation as an argument, it treats it like any other floating-point number and computes its sine, again accurate to the calculator's precision. To understand the approximate value, consider that for small $x$, $\sin(\pi + x) = -x$. In this case, $x$ is the difference between π and the calculator approximation: $\pi + x = 3.141592654$. Thus

$$x = 3.141592654 - 3.14159265359^{+} \approx 4.1\times 10^{-10},$$

and

$$\sin(\pi + x) = -4.1\times 10^{-10},$$

which is just what the HP-41 returns. SIN is evidently returning an accurate result for its argument, but the argument is not π.

Could a calculator be designed to recognize the approximation as its best numerical representation of π and return zero for the sine of that number? Certainly it could, but HP calculators generally don't do this sort of thing, following the guideline that the limitations of fixed-precision calculations make it unwise to try to guess when a numerical value is supposed to be some special number. This sort of problem shows up in lots of cases: for example, should 1/.142857142857 evaluate to 7.00000000001, which is the most accurate 12-digit reciprocal of that argument, or 7.00000000000, on the chance that .142857142857 was obtained originally by computing the reciprocal of 7? This problem is a fundamental limitation of trying to represent arbitrary numbers with a finite number of digits.

The HP 28 and HP 48 provide a different approach to the problem of π other calculators. Assuming for the moment that flags -2 and -3 are clear, executing π returns the *expression* 'π' (note that this is an algebraic object, not a name--e.g. TYPE returns 9). If you execute π 2 *, you obtain '2*π'. As long as you don't force numerical execution by executing →NUM, π retains its symbolic form through any number of operations. This has two immediate benefits:

- An expression containing the symbol π gives you more information about the nature and derivation of the expression. Once you convert it to a numerical form, no matter how accurate, the presence of π in the expression becomes obscured. The expression 'π/4' is more informative than the number 0.785398163398.
- Using symbolic π prevents errors arising from a finite precision numerical representation of π from accumulating in chained calculations. By delaying the substitution

of a numerical value for π until a calculation is complete, you obtain maximum accuracy.

A symbolic π also permits a new resolution of the sin π issue. On the HP 48, if you execute π SIN (with flags −2 and −3 clear, and radians mode active), you obtain 0. This is an automatic simplification (section 3.5.5.1), not a numerical computation--when SIN is executed, it checks its argument to see if it is symbolic π. If so, the subexpression SIN(π) is replaced by 0. The following additional simplifications are also made, in the same spirit:

- SIN(π/2) is replaced by 1 (note: SIN(1.5707963268) also returns 1);
- COS(π) is replaced by −1 (COS(3.14159265359) also returns −1);
- COS(π/2) is replaced by 0.
- TAN(π) is replaced by 0.

Only these four specific subexpressions are simplified. SIN(2*π), for example, is not simplified, and returns 4.13523074713E−13 when evaluated numerically.

### 3.5.6.1 Other Symbolic Constants

In addition to π, the HP 48 provides four other symbolic constants: e (numerical value 2.71828182846), i (value (0,1)), MAXR (value 9.9999999999E499), and MINR (value 1E−499). There are no special simplifications associated with e, MINR or MAXR, but the symbolic forms allow you to track the associated constants through calculations. i has these simplifications:

| **Subexpression** | **Replacement** |
|---|---|
| SQ(i) | −1 |
| i*i | −1 |
| i^2 | −1 |
| i^(2,0) | −1 |
| RE(i) | 0 |
| IM(i) | 1 |
| CONJ(i) | −i |

You can use i to enter complex numbers in the form *a* + *bi* rather than the standard object format (*a*,*b*). For example, 1+2i can be entered as '1+2*i'. You can perform arithmetic with such expressions, using EXPAN and COLCT where appropriate to simplify a multi-term expression into the form *a* + *bi*.

### 3.5.6.2 Evaluation of Symbolic Constants

Symbolic and numerical execution modes affect the way all built-in HP 48 functions evaluate symbolic arguments. The five symbolic constants π, e, i, MAXR and MINR behave as functions of zero arguments--and as functions they are sensitive to the execution mode. When flag −3 is clear, execution of any of these constants returns a symbolic result, which is just the constant itself unchanged. When flag −3 is set, execution of a symbolic constant replaces it with its numerical value.

It is possible by means of flag −2 to select a restricted form of numerical execution mode that affects only these constants. When symbolic execution mode is active (flag −3 clear), setting flag −2 causes symbolic constants to evaluate numerically, without affecting the execution of other functions. This permits, for example, replacement of symbolic constants with numerical values in expressions that contain formal variables (undefined names). To see this, enter 'X' PURGE, then enter the expression '2*π*X' into level 1. Then,

```
-2  CF  -3  CF  EVAL  ☞  '2*π*X'
```

and

```
→NUM  ☞  Undefined Name error.
```

But if you set flag −2:

```
'2*π*X'  -2  SF  EVAL  ☞  '6.28318530718*X'.
```

π evaluates to its numerical value, while with flag −3 clear * still returns a symbolic product.

## 3.6 Name Objects

The center of the action in the HP 48 is the stack, where objects can be manipulated and executed. However, it is impractical to keep all objects on the stack; in particular built-in objects and those in libraries are most convenient if they can be executed without ever putting them on the stack. To this end, the HP 48 provides several types of *name* objects, that let you access objects indirectly. Executing a name object either recalls or executes another object so that in many cases you can perform operations on objects entirely by means of their associated name objects.

Since objects are intrinsically nameless, to name an object requires storing it in memory in such a way as to preserve the association between an object and a name. In the HP 48, to *name* an object means to *store* it; a named object is a stored object, and vice-versa. Stored/named objects appear in several forms:

- *Built-in objects*--operations--are permanently stored in the HP48's read-only memory. A subset of operations called *commands* have names, and thus may be included in procedures or entered on the stack by means of their names. It is generally not necessary to distinguish between command objects and their command names, since they are not separable, and only the names are ever "visible."
- *Library objects* (section 3.4.11) contain extensions to the built-in command set in the form of stored objects that are accessed using *XLIB name objects*.
- *Global variables* (section 5.1) are the most visible form of storage of user-created objects, corresponding to numbered or lettered registers on other calculators. *Global name objects* (object type 6) are used to access the contents of global variables. These variables exist in the so-called *user memory* , also called *VAR memory* because of its association with the VAR key. The structure of user memory is explained in section 5.2.
- *Local variables* are created by programs for their own use, and only exist while the associated programs are running. Their contents are accessed by means of *local name objects*. See section 9.7.
- *Port variables* (section 5.3) are like global variables in port memory. Access to their contents is provided by means of *path-names*, which are specially tagged global names or lists.

The execution actions of all three types of name objects, including the path-name variation of global names, are designed to enable use of the stored objects associated with the names. Whether "use" of a stored object means execution of the object or merely a recall to the stack depends on where the object is stored, i.e. which type of name object is used.

You can view name objects as the HP48 version of the storage register numbers or letters used on ordinary calculators, but this simple picture doesn't really do justice to their power. Register numbers are purely passive labels, of the most primitive sort--they don't tell you anything about what is stored in the register. Names, on the other hand, label their variable contents with text that can help you remember what each variable does, and which make programs more legible. Furthermore, HP48 names are active instead of passive: when you execute them, they cause automatic recall or execution of another object.

## 3.6.1 Global Names

Global variables are intended for storing data for general access, and for containing named programs that act to extend the HP48 command set. With this in mind, global name objects are designed to work like commands:

*Execution of a global name causes execution of the object stored in the global variable with that name.*

The net result of the execution of a global name follows directly from the execution action of the object stored in the corresponding variable--data objects and algebraics return to the stack, programs (or commands) run, and name objects execute or recall their stored objects in turn. There is one extension to this general rule: if the stored object is a *directory*, execution of the associated name object does not leave the directory on the stack but instead makes the directory the *current directory* (section 5.2).

The properties of global name execution listed here explain why RCL is relegated to a shifted key position on the HP48 keyboard. Used with the names of variables containing all object types except programs and names, EVAL (which is on an unshifted key) and RCL are equivalent. The primary purpose of RCL, therefore, is to recall a stored program or name to the stack without evaluating it, a relatively infrequent need.

Unquoted global names act just like built-in commands, so that you can define your own command set by storing programs in global variables. You can execute a global name by:

- Typing the name into the command line and pressing ENTER ; or
- Pressing the VAR or CST menu key labeled with that name; or
- Including the name in a procedure (a program or an algebraic), and evaluating the procedure.

These three methods are identical for global name objects and HP48 commands.

[As it happens, HP48 commands are also programs written in a language that is a superset of the HP48 user RPL, so there really is no structural difference between user programs and commands. The practical difference is that since a built-in command is fixed in read-only memory, it can be encoded in a program by its memory address and hence executed more quickly than an object stored in a global variable. The latter are referenced by name, and must be searched for in user memory whenever their names are executed.]

The fact that executing the name of a stored algebraic object returns the object to the stack without evaluation makes possible "step-wise" algebraic substitution. For example, consider evaluating 'A+B', where A has the value 'C+D', B is 5, C is 10, and D is 20. The HP48 will return 'C+D+5' at the first use of EVAL, and −2 at the next. If an algebraic stored in a variable was automatically *evaluated* when the variable's name was executed, you would lose the intermediate step and obtain only the final result −2 at the first EVAL.

In the case where a global name is executed for which no variable currently exists, the action is simple--the name itself is just returned to the stack as if it were a data object. This behavior is necessary for symbolic operations; it means the HP48 can deal with symbols (names) even when no value has yet been established. Thus 'A+B', where A is undefined and B is 10, evaluates to 'A+10'. Execution of the A returns 'A', B returns 10, and + combines the symbolic 'A' and the number 10 into a new symbolic 'A+10'. We call A a *formal* variable, meaning you can work formally with the name in calculations just as if there were an existing variable named A.

If a variable contains a global name, the stored name is executed when the variable's name is executed. Thus if the number 8 is stored in the variable A, and 'A' is stored in B, evaluating B returns 8. This property of names leads to the possibility of "endless loops"--if 'A' is stored in B, and 'B' is stored in A, evaluating either A or B will start an unending circle of executions, so that the HP48 will be busy indefinitely without any apparent sign except that the hourglass annunciator stays on. You can just press [ATTN] to stop execution.

## 3.6.2 Local Names

*Local* variables are intended primarily for temporary storing and naming of stack objects, in order to simplify argument manipulations in programs. This implies that generally the objects stored in local variables are to be recalled to the stack unchanged (i.e. not executed). Hence local name execution is intentionally simpler than that of global names:

> *Execution of a local name recalls the object stored in the corresponding local variable,* without *executing the object.*

The creation and use of local variables is described in section 9.7.

## 3.6.3 XLIB Names

*XLIB names* provide access to objects stored within library objects. The abbreviation "XLIB" is short for "eXternal LIBrary", the "external" referring to a library that is not built into the HP48's permanent memory. In most respects, you use XLIB names in the same manner as commands--executing an XLIB name executes the associated object in the library. As long as its library is available (i.e. present in the current name resolution path--see section 5.5), an XLIB name is entered and displayed as (unquoted) text, again like a command. However, when its library is absent, a previously entered XLIB name is displayed in the form

XLIB *library-number, object-number,*

where *library-number* is the library identification number of the library, and *object-*

*number* is the number of the specified object within the library. Executing an XLIB name when its library is absent returns the Undefined XLIB Name error.

## 3.7 Quoted Names

We have shown that global and local names automatically replace themselves with their associated variable values when executed. But there are many cases where you need the name object itself on the stack, so that you can use it as an argument for a command like STO or GET. You can accomplish this by enclosing the name within single quote delimiters, e.g. *'name'*. The quotes around a name instruct the HP 48 to return the literal name itself, and not to execute it.

To store the value 10 into a variable X, the correct sequence is 10 'X' STO. If you omit the quotes, as in 10 X STO, you may very well get an error, since the *value* of X is returned before the STO executes, rather than the name X. You *can* use 10 X STO if the variable X does not yet exist, since that case executing X just returns to the stack the name 'X', which is a suitable argument for STO. In general, to avoid uncertainty you should keep the habit of entering the quotes around the name when you want to store. However, if you're primarily performing symbolic calculations, you may want to take the trouble to purge all of the variables you want to work with, just so you can put the names on the stack without bothering with the quotes.

## 3.8 Quotes in General

There are three sets of quotation marks that are used as HP 48 delimiters:

- Single quotes ' ', (called "ticks," for short) which identify algebraic objects, and also create name objects on the stack;
- Double quotes " ", which create strings; and
- Program quotes « » (*guillemets*), which create programs.

All three types of quotation marks have a common theme in the HP 48. They mean "put this object on the stack--don't execute it yet." Preventing execution of a string object is not particularly meaningful, since strings are data objects, but we include the double quotes " " in this discussion for completeness. The double quotes primarily distinguish text strings from names.

We stated in section 3.7 that placing single quotes around a global or local name enters the name as an object on the stack. The quotes play the same role for algebraic objects--the same symbol is used for the two different object types (name and algebraic) because it makes sense in many contexts to treat a name object as an algebraic expression consisting of just one variable name. As we mentioned in section 3.3, an algebraic object is a composite object and thus can be evaluated like a program--it happens to be displayed in algebraic form rather than RPN. Again, the quotes mean "don't execute this program, just put it on the stack." The HP 48 doesn't allow you to specify an immediate-execute algebraic object (i.e., without quotes)--if you want the expression to be executed immediately, you have to enter it in RPN form.

Although the same delimiters are used for algebraics and names, and for many cases you can treat them the same, they are still different object types. The distinction is maintained for the sake of commands like PUT and RCL, which would make no sense with an expression or an equation as an argument. The HP 48 insures a smooth interaction between names and algebraics by treating them uniformly (as a general *symbolic* object type) as arguments for functions, and by automatically converting algebraics containing only a variable name into name objects. Thus TYPE returns type 9 for the expression 'A+0', but if you evaluate the expression (assuming A has no value) to eliminate the 0, TYPE then returns 6, indicating that the object is a global name.

Understanding the meaning of quoted and unquoted *programs* starts with the recognition that the contents of the command line constitute a program--an arbitrary series of objects intended for sequential execution. When you're carrying out keyboard calculations, the execution is immediate as soon as you execute ENTER (section 6.4.3). The command line program is created, then executed right away. However, you can postpone execution of the command line by inserting a << delimiter at the start. ENTER then creates a program object containing the command line objects.

Because the command line is a program, and programs are deferred-execution command lines, it follows that whatever you can do in the command line, you can also do in a program (and vice-versa). Thus programs can contain quoted objects: names, algebraics, and even other programs. For example, here is a program named TEST that creates a global variable containing yet another program:

```
<< ... << 10 * >> 'X10' STO ... >> 'TEST' STO
```

Executing TEST executes its stored program, which in turn creates a variable X10 containing the program << 10 * >>. Because of the surrounding << >>, the sequence 10 * is not executed, but is put on the stack as a program, where it and the (quoted) name X10 are the arguments for STO.

Adding a tag (section 3.4.8) is another method of entering an object without execution. This point is particularly relevant for port names (section 5.3.1), since you can not add single quotes to a port name--but you don't have to because the tag prevents its execution anyway.

## 3.9 EVAL

As you saw in the preceding section, the various types of quote delimiters cause objects to be placed on the stack without being evaluated. The EVAL command is provided so that you can later evaluate these "pending" objects, particularly programs, names, and algebraics. Applying EVAL to a data object does in fact evaluate the object, but that just returns the same object.

Perhaps the most common use of EVAL arises in symbolic calculation, where you have entered an algebraic object and want to substitute values for the variable names that appear in the object's definition. The [EVAL] key also provides a handy way of making a keyboard calculation in algebraic syntax. Just press ['] to start algebraic entry, enter an expression, then press [EVAL], which here acts like an algebraic calculator's [=]. For example,

['] 1+2*3 [EVAL] ☞ 7.

## 3.10 System Objects

In addition to the object types described in the preceding sections, the RPL system uses several additional object types. Although these objects do not appear in normal use of the HP 48, you may see them in these circumstances:

- A defect in a system program may leave one or more such objects on the stack.
- Future libraries may provide for intentional user-manipulation of the system object types.

Table 3.3 on the next page summarizes the system object types.

Most built-in assembly language objects are also displayed as External when they are on the stack, because their structure does not conform to any of the object types listed in the table. You should not normally see such objects; if you do, it is due to a defect in the HP 48's built-in programming. We recommend that you immediately perform a system halt ( [ON] - [C] ) to remove the object and reset the system to a safe condition. Do *not* try to evaluate the object.

### 3.10.1 SYSEVAL

Built-in HP 48 program objects--commands--are permanently stored in the calculator. These objects are always in the same place in memory; any such object could in principle be executed by specifying its memory *address* rather than its name. In fact, this execution-by-address is the most common form of execution within HP 48 system

**Table 3.3. System Objects**

| Object Type | TYPE Number | Stack Display | Class | Definition |
|---|---|---|---|---|
| System binary | 20 | <*nnnnn*> | Data | 20-bit unsigned integer |
| Extended Real | 21 | Ext. Real | Data | Extended precision (15-digit mantissa, 5-digit exponent) real number |
| Extended Complex | 22 | Ext. Complex | Data | Extended precision complex number |
| Linked Array | 23 | Linked Array | Data | Like ordinary array, but all elements do not have to be present. |
| Character | 24 | Character | Data | One text character. |
| Code Object | 25 | Code | Procedure | A program written in assembly language. |
| Library Data Object | 26 | Library Data | Data | Data-class object used by libraries to save data specific to each library. |
| External Object | 27-31 | External | Data | Data-class objects not specifically defined in the HP 48 (may be used by external software). |

programs. Furthermore, the HP 48 contains many hundreds of objects that are not named, and which are consequently not directly executable from the keyboard. The majority of these objects are not useful for common HP 48 operations--those that are most useful have names to make them commands. However, some unnamed objects do have practical uses.

The SYSEVAL command provides for execution of any system object by means of its address. That is, you enter the object address as a binary integer object, then execute SYSEVAL, which in turn executes the specified system object. From time to time, in response to customers' requests, Hewlett-Packard has published the addresses of a few system objects that help solve certain common programming problems.

For example, if a program creates a temporary display by means of DISP or other commands, that display will persist until the end of the program. You can cause the calculator to restore the normal stack display while a program is running by executing #39BADh SYSEVAL.

You must use extreme care when using SYSEVAL, for execution with an incorrect address may cause a system halt or a memory reset (section 5.8). When you execute SYSEVAL from the command line, or enter it in a program, you should do the following:

- Be sure that the address you are using is correct.
- Be sure you enter the address correctly. This means not only getting all digits right, but also making sure that the number is correct for the current binary integer base. All of the SYSEVAL addresses listed in this book are given in hexadecimal, so you should execute HEX before entering the binary integer address. (Remember that including HEX in the command line does *not* affect the interpretation of binary integers entered in that same command line).
- Do not attempt to single-step (section 12.2.2) programs containing SYSEVAL. If you need to do this, replace the sequences *#address* SYSEVAL with global names, where each name corresponds to a variable containing a program

  << *#address* SYSEVAL >>.

# 4. The HP48 Stack

The HP48 *stack* is the center of all calculator operations. It is the place where the great majority of commands find their arguments and return their results. It's also the primary and most efficient means for commands and programs to transfer data and instructions so that a series of calculations can be linked together. In this chapter, we'll describe the fundamental stack operations by which you can manipulate the objects on the stack. We will use real numbers and names as example objects, but all of the stack operations described here apply uniformly to any of the various RPL object types. There are numerous practical examples of stack manipulations in the program examples in later chapters.

The stack consists of series of numbered *levels*, each of which contains one object of any type. The stack is always filled from the lowest level up, so that there are never any empty levels between full ones. ENTER always moves new objects from the command line into level 1, pushing previous stack objects up to higher levels. Most commands remove their argument objects from the lowest levels, whereupon the objects in higher levels drop down. The only exceptions are some of the stack manipulation commands, which can move objects to or from arbitrary stack levels. There is no limit on the number of objects or levels of the stack; you can enter as many objects as available memory will permit.

The HP48 provides an extensive set of stack manipulation commands, some permanently assigned to keys, and the rest contained in the stack menu ( PRG ≡STK≡ ). All of the stack menu operations are programmable commands, which means that you can execute them by pressing the appropriate keys or by spelling their names into the command line. Most stack operations can also be executed by using the *interactive stack*, described in section 4.5.

If you have no previous experience with RPN calculators, a good way to get used to the RPN stack is to view it at first as a "history" stack, which keeps a record of your calculations. That is, you can calculate in "algebraic" style by entering algebraic expressions surrounded by ' ' delimiters (see section 3.8) and pressing EVAL to perform the calculations. The successive results pile up on the stack, where you can then start to get the "feel" of RPN by executing RPN commands to combine the results into new values.

Table 4.1 lists the stack operations found on the keyboard and in the STACK menu. The individual operations are explained in subsequent sections. [Most of the HP48 stack commands are adapted from the FORTH computer language. Indeed, many key HP48 features are based on FORTH, with its unlimited data and return stacks, RPN logic, and structured programming.]

**Table 4.1. HP 48 Stack Manipulations**

| | **Command** | **Action** |
|---|---|---|
| *Stack Clearing* | DROP | Discard the level 1 object |
| | DROP2 | Discard the objects in levels 1 and 2 |
| | DROPN | Discard the first *n* objects |
| | CLEAR | Discard all stack objects |
| *Reordering Arguments* | SWAP | Exchange the objects in levels 1 and 2 |
| | ROT | Rotate the level 3 object to level 1 |
| | ROLL | Rotate the level *n* object to level 1 |
| | ROLLD | Rotate the level 1 object to level *n* |
| *Copying Objects* | DUP | Copy the level 1 object |
| | OVER | Copy the level 2 object |
| | PICK | Copy the level *n* object |
| | DUPN | Copy the first *n* objects |
| *Counting Objects* | DEPTH | Count the number of objects on the stack |
| *Object Recovery* | LASTARG | Return the arguments used by the last command |
| | LAST STACK | Restore the stack to its state prior to ENTER |

## 4.1 Clearing the Stack

Perhaps the most common stack operation is "clearing" one or more objects, either to discard unnecessary objects so that others are moved to lower levels, or just to clear the decks for a new calculation. The latter is accomplished by CLEAR, which removes the entire contents of the stack in a single operation. CLEAR is usually executed from the keyboard (⮣ CLR ); a well-designed HP 48 program does not execute clear because that might destroy stack objects needed by a second program that called it.

There are three commands for removing a specific number of objects from the lowest-numbered stack levels: DROP, DROP2, and DROPN. The basic command is DROP, which removes the object in level 1, and "drops" the remaining stack objects one level to fill in the empty level. Each DROP discards another object, and the stack drops one level.

DROP2 and DROPN, are equivalent to repeated execution of DROP. DROP2 does just what its name implies: it removes two objects, from level 2 and level 1, then drops the remaining objects down two levels to fill in. DROPN drops *n* objects in addition to the number *n* in level 1 (so actually $n+1$ objects are dropped--see section 4.2.4 for a discussion of stack depth parameters). Notice that although DROPN appears abbreviated as DRPN in menus, its correct name in a program is DROPN.

The need to drop objects arises when extraneous or no-longer-necessary objects occupy the lowest stack levels. For example, if you take a vector apart with OBJ→, level 1 will contain a list { *n* } specifying the number of elements in the vector. But if you are working with vectors of a particular size, the size list may be redundant information, in which case you can drop the list and continue with operations on the elements.

## 4.2 Rearranging the Stack

Dropping objects from the stack is not always the appropriate action when you need access to objects in higher-numbered stack levels--you may also need to preserve the low-numbered objects. In such cases, you must employ stack rearrangement commands to change the order of the objects.

### 4.2.1 Exchanging Two Arguments

The simplest form of stack rearrangement is the exchange of the positions of the objects in levels 1 and 2, which is accomplished by SWAP. SWAP is used for switching the arguments for a two-argument command, or more generally for changing the order in which the level 1 and 2 objects may be used. SWAP is easy to illustrate:

A B SWAP ☞ B A.

### 4.2.2 Rolling the Stack

A stack "roll" is an exchange of stack positions involving objects in two or more stack levels. One object is moved to or from level 1, and other objects move up or down together to make room for it. The commands ROLL (roll up) and ROLLD (roll down) provide for stack rolls in both directions, where "up" and "down" refer to the apparent motion of the stack objects other than the level 1 object. You must specify the number of stack levels you want to roll by placing a number *n* in level 1. Either command drops the number from the stack, then rolls the first *n* of the remaining stack objects. For example, if $n=4$:

| **Level** | | **Stack Contents** | |
|---|---|---|---|
| | *Before* | *After* 4 ROLL | *After* 4 ROLLD |
| 4: | *t* | *z* | *x* |
| 3: | *z* | *y* | *t* |
| 2: | *y* | *x* | *z* |
| 1: | *x* | *t* | *y* |

Although ROLL and ROLLD move several objects at once, the primary purpose of these commands is still focused on level 1:

- *n* ROLL means "bring the *n*th level object to level 1." That is, ROLL retrieves a previously entered or computed object that has been pushed to higher stack stack levels by subsequent entries.
- *n* ROLLD means "move the level 1 object to level *n*." ROLLD moves the level 1 object "behind" other objects that you want to use first.

SWAP and ROT (rotate) are one-step versions of ROLL. SWAP is equivalent to 2 ROLL; ROT is the same as 3 ROLL. 0 ROLL and 1 ROLL do nothing, but the latter is still useful in program loops that use objects from successive stack levels including level 1.

### 4.2.3 Copying Stack Objects

One of the strengths of RPN calculators is their ability to make copies of an object on the stack, so that you can reuse it without having to stop and create a variable. The simplest example of this facility is the HP48 command DUP, which makes a second copy of the object in level 1, pushing the original copy to level 2, and all other stack objects up one level. The HP48 also lets you copy a block of stack objects with DUPN. The sequence *n* DUPN, where *n* is a real integer, makes copies of the first *n* objects on the stack. The order of the objects is preserved; for example

X Y Z 3 DUPN ☞ X Y Z X Y Z.

DUP2 is a one-command version of 2 DUPN:

X Y DUP2 ☞ X Y X Y.

In some cases it is desirable to copy an object that is not in level 1, by bringing a copy to level 1 while leaving the object in its original position relative to other objects. In the HP48, this combination of ROLL, DUP, and ROLLD is provided by PICK, the general purpose stack copy command. PICK works like ROLL, returning the *n*th level object to level 1, but it leaves the original copy behind. The original therefore ends up in level *n+1*:

W X Y Z 4 PICK ☞ W X Y Z W.

DUP is the same as 1 PICK, and OVER is a one-step version of 2 PICK:

X Y OVER ☞ X Y X.

Generally, you use PICK and ROLL when you are carrying out a complicated calculation entirely with stack objects. When you need to use a certain object repeatedly, you use PICK to get each new copy of the object. For the *final* use of the object, use ROLL instead of PICK; then you won't leave an unneeded copy around after the calculation is complete.

### 4.2.4 How Many Stack Objects?

Several HP48 stack commands require you to supply an argument that specifies how many stack levels the command will affect. Because this argument is always taken from level 1, you might be uncertain about what the argument should be--should you count level 1, which contains the argument? The answer is no--always count the stack levels you need before the count is entered into level 1.

For example, suppose the stack looks like this:

| | |
|---|---|
| 4: | D |
| 3: | C |
| 2: | B |
| 1: | A |

To roll D to level 1, execute 4 ROLL. But notice that at the point when ROLL actually executes, the stack is:

| | |
|---|---|
| 5: | D |
| 4: | C |
| 3: | B |
| 2: | A |
| 1: | 4 |

Here D is actually in level 5. But don't try to compensate for this by using 5 as the argument to ROLL. ROLL removes its argument from the stack *before* it counts levels for the roll. All other similar commands, such as DUPN, PICK, ROLLD, →LIST, etc., work the same way.

DEPTH, which returns the number of objects currently on the stack, works in conjunction with this class of commands. The count returned by DEPTH does not include itself--it counts the objects before the new count object is pushed onto the stack. (Every time you execute DEPTH, the depth increases by one.) Thus DEPTH ROLL rolls the entire stack, DEPTH →LIST packs up all the stack objects into a list, etc.

## 4.3 Recovering Arguments

HP 48 commands characteristically remove their arguments from the stack. Occasionally, it is useful to recover a copy of one or more of a command's arguments:

- To allow you to re-use the same argument(s) for a new command.
- To help you reverse the effect of an incorrect command, by applying the inverse of the command to some combination of the result and the original arguments.

Traditional HP four-level RPN calculators have a LASTX command that combines these two purposes. On the HP 48, there are two separate operations:

1. The capability of recovering an argument for reuse is provided by the *last arguments recovery* system, whereby each command that uses stack arguments saves copies of all of its arguments--up to five--in a reserved area of memory. No built-in HP 48 command uses more than four arguments, but the last arguments system provides for up to five for the sake of library commands. Commands like DUPN or →ARRY, which appear to use an indefinite number of arguments, are considered for this purpose to use only one argument, which is the number or list in level 1 that specifies the number of stack levels that are involved.

   The arguments saved by the most recent command can be retrieved by the command LASTARG (also called LAST, for compatibility with the HP 28), which re-enters all of the arguments onto the stack in their original order. Note that since most HP 48 commands use arguments, the last arguments objects change frequently. Even simple stack rearrangements such as DROP and SWAP save their arguments. Only commands like STD or HEX, that use no arguments at all, leave the last arguments unchanged.

2. Manual recovery from incorrect commands is provided by the *stack recovery* system. At the start of each ENTER, a copy of the entire stack is saved (see section 6.4.3) in a local memory (section 5.4). When all of the objects processed by ENTER have completed execution, you can cancel their stack effects by pressing [◁] [LAST STACK] . This discards the new stack and replaces it with the stack contents saved by ENTER.

The objects saved for stack recovery and last argument recovery can consume a substantial amount of memory if the objects are numerous or large. When you are working with objects that are comparable in size to available memory, such as adding large arrays, the memory needed to save copies of objects for recovery can actually prevent you from carrying out various operations. For this reason, the HP 48 gives you the option of disabling either or both of these features (and also the command stack), by means of the appropriate keys in the ⬅ MODES menu. You can also disable argument recovery by setting flag −55.

Two notes:

- Disabling last arguments prevents commands that error from returning their arguments to the stack. This makes it harder to recover from an error, and also affects the design of error traps (section 9.6).
- If there is insufficient memory available to save the current stack as the recovery stack, the HP 48 shows the error message No Room for LAST STACK, and *automatically* disables stack recovery. This last step is necessary, since you would otherwise be unable to do anything--including trying to free some memory. Any command would fail, since the HP 48 tries to save the stack before executing the command.

LASTARG can also be used to recover accidentally purged or replaced variables. See section 5.1.3.

## 4.4 Stack Manipulations and Local Variables

The following example illustrates the use of several of the HP 48 stack commands. If you execute the commands one at a time, you can observe how to copy, move, and combine stack objects.

■ *Example.* Write a program that computes the three values

$$P + A + B$$
$$P + B \cdot F + A/F$$
$$P + B/F + A \cdot F,$$

leaving the results on the stack. Assume that P is in level 4, A in level 3, B in level 2, and F in level 1.

■ *Solution*:

```
<<  4  ROLLD  3  DUPN  3  DUPN  +  +
  8  ROLLD  7  PICK  *  SWAP  7  PICK
  /  +  +  5  ROLLD  4  PICK
  /  SWAP  4  ROLL  *  +  +
>>
```

This example illustrates the use of stack manipulation commands, but it does not necessarily represent the best way to solve the problem. Keeping track of numerous objects on the stack takes considerable care when you are writing or editing a program. In general, manipulating objects on the stack in a purely RPN manner yields the most efficient programs (see section 12.4). However, there are other programming techniques that are easier and produce more legible programs. For example, you can store the initial and intermediate values in global variables, then recall each to the stack by name as it is needed in the calculations. Better yet, you can avoid cluttering up user memory with a lot of variables (which you may or may not need after the program is finished) by using *local* variables.

With local variables, the solution to the example problem is

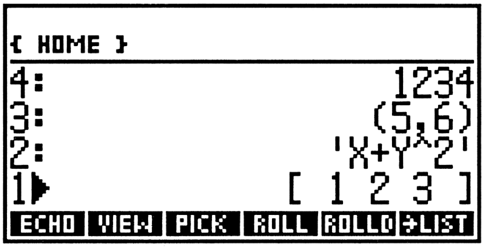
```
<<  →  p  a  b  f
  <<  'p+a+b'  EVAL
    'p+b*f+a/f'  EVAL
    'p+b/f+a*f'  EVAL
  >>
>>
```

→ p a b f takes the four initial values off the stack and assigns them to local variables p, a, b, and f (here we are using the convention of lower-case characters for local names). The rest of the program computes the three results, then discards the local variables. The obvious advantage of this method is that you can write the program "instantly," since the program so closely resembles the written form of the expressions you are trying to compute. The use of local variables is explored in detail in sections 8.5 and 9.7.

## 4.5 The Interactive Stack

HP 48 stack commands are available either on the keyboard (DROP, SWAP, DUP, and CLEAR) or in the stack menu ( PRG STK ). However, the HP 48 also provides the *interactive stack environment*, in which you can apply stack commands to objects in various levels by selecting the objects with a pointer rather than a stack level argument. The interactive stack also lets you view or edit any stack object, copy objects to the command line, combine objects into a list, and discard objects from the stack.

The interactive stack is activated by pressing △ when there is no command line active. The interactive stack menu appears, and the colon in the level 1 indicator 1: changes to a triangle pointer, to show that the level 1 object is currently selected:


```
{ HOME }
4:                  1234
3:                 (5,6)
2:              'X+Y^2'
1▶            [ 1 2 3 ]
```

Note that the stack is redisplayed in single-line format, so that four stack levels can appear in the display. Pressing △ moves the selector to level 2; pressing the key repeatedly moves the arrow to the top of the stack display and then begins scrolling objects from higher levels into the window. ⬑ △ moves up four levels; ⬏ △ moves the arrow to the highest stack level. You can also move the arrow down using ▽ , ⬑ ▽ , and ⬏ ▽ .

"Selecting" an object consists of moving the arrow to point at it; the stack level number of the selected object is then an implicit argument for the stack operations that appear in the menu. For example, to move the object in level 5 to level 1, you press △ five times (or △ ⬑ △ ), then press ≡ROLL≡ . This is equivalent to executing 5 ROLL, but it is easier because the very act of moving the pointer up to level 5 to see where the object is not only automatically activates a menu containing ROLL, but also saves you from having to enter the 5.

The interactive stack menu operations ≡PICK≡ , ≡ROLL≡ , ≡ROLLD≡ , ≡→LIST≡ , ≡DUPN≡ , and ≡DRPN≡ (DROPN) are self-explanatory, since they derive from the corresponding stack commands (section 4.2), using a stack level argument provided implicitly by the stack pointer. The remaining four operations in the interactive stack menu do not have command equivalents:

- ≡ECHO≡ is for copying an object to the command line when you want a new copy of the object, either to modify to make a new object, or to embed in some command line sequence. It differs from EDIT or VISIT in that the new command line object does not replace the original stack object.

- VIEW activates the appropriate viewer (section 6.5.1) for the selected object.
- KEEP discards all stack objects in levels above the selected object. It is intended for manual stack cleanup, and has no programmable equivalent since generally it is not a good idea for a program to discard objects that might have been on the stack before it began execution. It is, however, easy to write a program to replicate KEEP --see section 4.6.1.
- LEVEL returns the selected level number to level 1 (pushing current stack objects to the next higher stack level).

In addition to the interactive menu keys, these other keys are active:

- ⇦ removes the selected object from the stack. It is equivalent to *n* ROLL DROP.
- ◁ EDIT (you can omit the ◁) edits the selected object in the command line (in program entry mode), and returns it to its original level when you press ENTER . Thus the interactive stack EDIT is equivalent to ordinary ▷ VISIT with a numerical argument.
- ▷ VISIT takes the indirection of object editing one step further: it uses the selected stack object as an argument for VISIT (section 6.5). If the selected stack object is a number, the object in the stack level indicated by that number is copied to the command line; if the selected object is a global name, the object stored in the corresponding global variable is edited. (If the selected object is not a global name or a valid stack level number, an error is returned and the interactive stack environment is terminated with the invalid argument copied to level 1.)

## 4.6 Managing the Unlimited Stack

If you have not previously used an RPN calculator, you should find that the HP48's unlimited stack of objects is a straightforward implementation of RPN principles. However, if you are used to a four-level HP41 style stack, there are several general aspects of the use of the HP48 stack that will require some adjustment. The hardest part, perhaps, may be changing keystroke and programming practices that you have developed to use the advantages and to overcome the disadvantages of a four-level stack. In the following sections, we will outline some suggestions for optimum use of the unlimited stack.

### 4.6.1 Stack Housekeeping

An important advantage of an unlimited stack is that objects are never lost by being pushed off the end of the stack when a new object is entered. This is also a mild disadvantage--if you don't clear objects from the stack when you're through with them, more and more objects will pile up. This not only wastes memory, but causes the HP48 to pause more frequently for memory packing (section 12.9.1). It can also be distracting to see old objects appear in the display when you've long since forgotten their purpose.

A general recommendation for HP48 stack management is to clean up the stack after a calculation is complete. By all means pile up as much as you want on the stack while you are working through a problem--that is its purpose. But when you're finished, empty the stack. You can do this either at the beginning or the end of each calculation. We recommend the latter, since at that point you will best remember what each object is, and whether it's all right to throw it away.

"Clean up the stack" doesn't always mean to empty the stack with CLEAR ([⮫] [CLR] . You may very well want to keep certain objects, either leaving them on the stack or storing them in variables. Notice that STO removes the object being stored from the stack, reducing the number of objects on the stack.

The interactive stack is particularly useful for selective stack cleanup:

- To discard a single object, select it and press [⇐].
- To discard a block of objects at the low-numbered end of the stack, select the highest-numbered object to discard and press ≡DRPN≡ .
- To discard a block of objects at the high-numbered end of the stack, select the highest-numbered object that you want to keep, and press ≡KEEP≡ .
- To discard a block of object in the middle of the stack, select the lowest-numbered object to discard, and press [⇐] repeatedly.

You can write programs that perform the different stack removal operations, although their practical use is properly structured programs is limited. KEEP is a program form of the interactive stack ≡KEEP≡ operation; it discards all objects after the first *n*, where *n* is specified in level 1. For example,

A B C D E 2 KEEP ☞ D E

(This is our first example of a named program; you may wish to refer to the description of the program listing format in section 1.3)

| KEEP | *Keep N Objects* | | | A24D |
|---|---|---|---|---|
| ... | *level 1* | \| | ... | |
| *objects* | *n* | ☞ | *n objects* | |

| Program | Comments |
|---|---|
| `«  →LIST` | Combine *n* objects in a list. |
| `  →  keep` | Save the list in a local variable. |
| `  «  CLEAR` | Clear the stack. |
| `    keep  OBJ→  DROP` | Put the saved objects back on the stack. |
| `  »` | |
| `»` | |

MNDROP discards all objects from levels *m* through *n*. For example,

A B C D E 2 4 MNDROP ☞ A E

| MNDROP | | *DROP* m *through* n | | | | | 13BF |
|---|---|---|---|---|---|---|---|
| *level n+2* | ... | *level 2* | *level 1* | \| | level m−n | ... | level 1 |
| $object_n$ | ... | *m* | *n* | ☞ | $object_{m-1}$ | ... | $object_1$ |

| Program | Comments |
|---|---|
| `«  SWAP  DUP  →  n` | Save *n*. |
| `  «  -  1  +  1  SWAP` | Set up to repeat *m−n+1* times. |
| `    START  n  ROLL  DROP` | Drop one object. |
| `    NEXT` | Repeat. |
| `  »` | |
| `»` | |

Occasionally you may need to interrupt one ongoing keyboard calculation in order to perform another, and wish to resume the suspended work later. In this case it is not appropriate to clear the stack with CLEAR to provide an empty stack for the new calculation. You could take the trouble to save each object in a variable, but this is tedious, and makes it hard to reconstruct the stack order of the objects. A better approach is to preserve the entire stack in a single variable by combining the stack objects into a list. From the keyboard, you can use the interactive stack; the keystrokes are

[△] [↱][△] ≣→LIST≣ [ENTER].

Then you can store the list into a variable named OLDST (for example) by typing

'OLDST' STO . The stack is now cleared for another calculation. After completing any number of subsequent operations, you can restore the old stack by pressing

VAR ≡OLDST≡ PRG ≡OBJ≡ ≡OBJ→≡ ⇦ .

(The DROP removes the object count returned by OBJ→.)

In a program, a local variable (section 9.7) is ideal to save the stack contents:

| | |
|---|---|
| DEPTH →LIST → keep | Save the stack in local variable keep. |
| << ... | Any program steps here... |
| keep OBJ→ DROP | Restore the old stack. |
| >> | |

## 4.6.2 A Really Empty Stack

An important property of the HP 48 stack not shared by an HP 41-style stack is its ability to be *empty*. That is, when you clear the stack with DROP or CLEAR, there's *nothing* left. If you try to execute a command that requires arguments, you'll get an outright error--Too Few Arguments. The HP 48 makes no attempt to supply default arguments.

You can turn this property to advantage. The following sequence adds a series of numbers on the stack, no matter how many there are:

```
WHILE DEPTH 1 > REPEAT + END "TOTAL" →TAG
```

The sequence is an indefinite loop (section 9.5.2) that keeps adding (REPEAT +) as long as there is more than one object on the stack (WHILE DEPTH 1 >), then quits, leaving the labeled total in level 1. This routine is useful when you must add a column of numbers--you can enter all of the numbers onto the stack, use the interactive stack to review the entries, then perform all of the additions at once. Notice that if an empty stack were treated as if it were filled with zeros, there would be no way for the program to know when to stop adding.

## 4.6.3 Disappearing Arguments

The HP 48 itself takes some steps to insure that unnecessary objects don't pile up on the stack. In particular, most commands that use stack arguments remove those arguments from the stack. You shouldn't find this surprising; for example, you wouldn't expect the sequence 1 2 + to leave the 1 and the 2 on the stack as well as the answer 3. But it may be a little disconcerting the first time you use STO on the HP 48, to see that the object you just stored disappears from the stack.

If commands did not remove their arguments from the stack, then you would have to take the trouble to drop them when you no longer need them. On the other hand, since HP 48 commands do remove their arguments, you must remember to duplicate them before executing the commands on those occasions when you want to reuse the arguments. The HP 48 chooses this approach for these reasons:

- Consistency with mathematical functions. You *never* want math functions to leave their arguments on the stack--otherwise, the whole RPN calculation sequence would be disrupted.
- Stack "discipline." The fewer objects that are on the stack, the easier it is to keep track of what they are.
- Efficiency. It's easier to duplicate or retrieve a lost argument than it is to get rid of an unwanted one.

To illustrate the last point, consider obtaining a substring from a string:

```
"ABCDEFG" 3 4 SUB ☞ "CD".
```

This sequence returns only the result string "CD"; the original string "ABCDEFG", and the 3 and 4 that specify the substring are discarded. If you want to keep the original string, add a DUP after the original string object:

```
"ABCDEFG" DUP 3 4 SUB ☞ "ABCDEFG" "CD".
```

If SUB left its arguments on the stack, the original sequence would yield a final stack like this:

```
4:        "ABCDEFG"
3:                3
2:                4
1:             "CD"
```

In that case, to leave only the result on the stack, you would have to add 4 ROLLD 3 DROPN to the sequence. If you only want the two strings, you would have to add ROT ROT DROP2. As we stated, either of these is more complicated than adding a DUP to the start of the sequence.

When you use STO to preserve an intermediate result in the middle of a calculation, you may prefer to keep the result on the stack so that you can continue the calculation. In this case, just execute DUP (press $\boxed{\text{ENTER}}$ if you're performing manual calculations) before you enter the variable name for the STO. If you forget, the stored object is always available by name in the VAR menu.

## 4.7 Design Insights

An alternative (and more accurate) picture of the HP 48 stack is that the stack consists of the stack objects themselves, rather than a set of levels that may or may not contain objects. The picture conveyed by the HP 48 display is slightly misleading in that it suggests that the stack levels with their numbers actually exist in memory, including the empty levels that are just waiting to have objects put in them. (This picture is literally correct in the four-level RPN calculators that preceded the HP 48.) In fact, the stack consists of the stack objects placed adjacent to each other in memory, a starting memory location, and a memory pointer. The pointer points to the location where the next stack object will be placed. If the pointer points to the start, the stack is empty. When an object is placed on an empty stack, it is stored at the starting location, and the stack pointer is adjusted to point just past the object. As additional objects are added, they are placed next to the last-entered object, and the pointer is adjusted. You can picture the stack as growing like this:

```
Empty        One          Two          Three
             Object       Objects      Objects

▷             ONE          ONE          ONE
             ▷             TWO          TWO
                          ▷             THREE
                                       ▷
```

The key idea here is that when objects are added to and deleted from the stack, the remainder of the stack does not move (as you might think from the 48 display, since entering an object shows the initial objects moving up the display, and dropping objects shows objects moving down). Thus it takes no more time to add an object to a stack of 1000 objects than it does to an empty stack. Similarly, when you execute any stack rearrangement, the only movement takes place among the objects involved in the rearrangement.

In the diagram we show the stack growing downwards, as in the HP 48 display. Descriptions of stack-oriented computer languages usually show the opposite picture, with the "top" of a stack being the most-recently entered object. HP RPN calculator manuals have always shown level 1 (the $x$-register) at the visual "bottom" of any stack pictures. The HP 28C was the first calculator in which more than one level was visible at a time; it displays level 1 at the bottom of its display. The HP 48 continues this model, which is sensible since when you perform a simple operation like addition, the numbers appear on the stack the same way they would appear on paper, with the first-entered number above the second. To avoid confusion, however, we will not refer to the "top" or the "bottom" of the stack, referring instead to specific stack object/level numbers.

The stack-of-objects model needs further modification to correspond exactly to the HP48 internal design. The real HP48 stack is a stack of the *memory addresses* of the visible stack objects rather than the object themselves. The objects may be in any of a number of places--in user memory, in the built-in ROM, in plug-in RAM or ROM, or, if not in one of these places, in a temporary object memory. All HP48 operations that deal with the stack "know" that the objects are only present indirectly on the stack. Because of this consistent system design, you can deal with stack objects as if they were literally in a stack without any concern about the indirection.

An understanding of the internal stack design can, however, provide some insights into using the system efficiently, such as why stack manipulations are very fast. The addresses on the stack are all the same size--2.5 bytes--so that copying them, counting through them, etc. involves very simple operations that can be encoded in very efficient assembly language. For example, DUP has only to duplicate the 2.5 byte address of the level 1 object--it does not have to copy the object itself--and add 2.5 to the stack end pointer. (This also means that copying an object with DUP only uses 2.5 bytes of memory.) Also, finding an object on the stack is fast; to find the level one object, the HP48 just reads the address indicated by the stack pointer. By contrast, to find an object stored in a variable from the variable's name, the calculator must search through user memory until it finds a variable with the right name, which can require many memory reads and comparisons.

The stack-of-addresses model implies that you can make any number of copies of an object at a memory cost of only 2.5 bytes per copy. When you execute a program that contains an explicit object that goes onto the stack, it still only costs 2.5 bytes for the object, because the program literally contains the object. The resulting stack address points inside the program, to the point in the program where the object is defined. There is a catch here: if you purge the program while the object it entered is still on the stack, the HP48 copies the entire program to temporary object memory where it remains until you finally drop the stack object. The memory occupied by the program is only reclaimed when the object is dropped, not when the program is purged.

Other consequences of the RPL stack design are discussed in sections 11.6 (composite objects and memory) and 12.9.1 (erratic execution). The complete logical description of the internal design of RPL would constitute a book by itself. Fortunately, you can generally use the HP48 and write quite elaborate programs without concern about the details of its internal design.

# 5. Storing Objects

The methods and organization of object storage on the HP 48 are quite straightforward in practice, but can be a little convoluted to explain in the abstract. Therefore we will develop this theme by means of a continuous example, where we will start with a hypothetical "empty" calculator and start to fill it with stored objects, explaining the principles as they are introduced in the example.

## 5.1 Global Variables

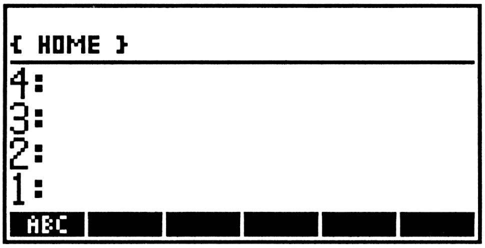
Imagine now that you want to enter the real number 123 and store it away for future use. This is accomplished by using the command STO to create a *global variable* that both stores the number and gives it a name. STO evidently requires two arguments: the object to be stored (level 2), and a name (level 1). The name in this case is represented by a *global name object* (section 3.6.1):

```
123  'ABC'  STO
```

This sequence enters the number 123 and stores it with the name ABC. The quotes ' ' surrounding the name ensure that the name object itself is entered on the stack, rather than executing the name (section 3.7). You should notice that both 123 and 'ABC' are removed from the stack by STO; to leave a copy of the 123 on the stack, you should copy it first:

```
123  DUP  'ABC'  STO
```

To see where the 123 has gone, press the VAR key:



```
{ HOME }
4:
3:
2:
1:
```

You are seeing the VAR menu, which is an automatic catalog of all global variables that currently exist. In this example, there is only one variable, ABC, which appears as the label of the leftmost menu key. If you press that menu key, the number 123 is returned to the stack, which demonstrates the fundamental behavior of VAR menu keys:

- Pressing an unshifted VAR menu key executes the global name displayed on the key label.

According to the principles of global name execution described in section 3.6.1, executing a global name executes the object stored with that name, which in this example is the number 123. When the stored object is a program, a name or a directory, you may want to recall the object without executing it, which leads to a second property of the VAR menu:

- Pressing a right-shifted VAR menu key recalls the object stored in the corresponding global variable.

Thus, pressing [↱] [ABC] is equivalent to executing 'ABC' RCL. For objects other than programs, names, and directories, executing the object is the same as recalling it to the stack, so the right-shifted and unshifted VAR menu keys have the same effect. When you are unsure of a stored object's type, and want to recall it without executing it, you should use the right-shifted menu key.

For symmetry, the left-shifted VAR menu keys are also active:

- Pressing a left-shifted VAR menu key stores the object in level 1 in the corresponding global variable.

456 [↰] [ABC] is equivalent to 456 'ABC' STO. When STO is executed with the name of an already existing variable, the existing contents of the variable are replaced with the object in level 1.

The action of the left-shifted menu keys as a shortcut for STO has the obvious disadvantage that it is easy to overwrite the contents of a variable accidentally, when you press the left shift instead of the right or forget that the left shift was left active from some previous incomplete operation. To help you remember which shift is which, observe that shifted menu key operations roughly match those of the shifted [STO] key: [↱] [STO] performs RCL, like the right-shifted menu key; and [↰] [STO] executes DEFINE, which is a type of storing. Also, if you do perform an unwanted store by pressing a left-shifted menu key, you can undo the operation by immediately pressing

[↱] [LASTARG] [STO] [↱] [LASTARG] [⇐].

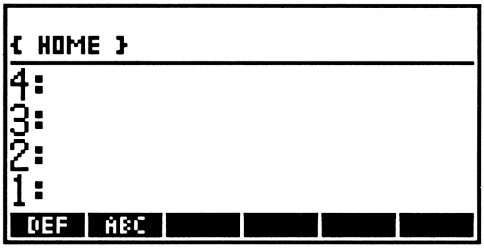
(see section 5.1.3).

The properties of the VAR menu keys described above apply only to immediate-execute entry mode; in algebraic (ALG annunciator) or program (PRG) entry modes, an unshifted menu key merely echoes the key label name to the command line, and the shifted menu keys are inactive.

Now create a second variable DEF:

456 'DEF' STO

The VAR menu now looks like this:

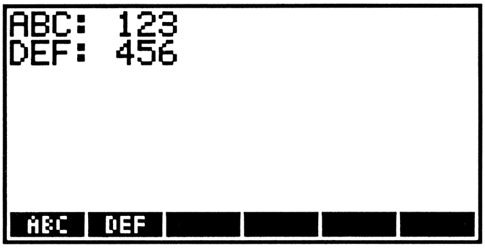


The newer variable DEF appears on the left-most menu key, with ABC moved one position to the right. In general, as each new variable is created, its menu entry takes the first menu position. This ensures that the most recently created entries are the most accessible in the menu, but it also means that menu entries move around as variables are created or deleted (which can trip you up if you are pressing keys quickly, since the display showing the menu positions is not updated until any type-ahead keystrokes are processed). The command ORDER gives you control of the order of menu keys in the VAR menu. ORDER rearranges the menu to match the order of names in a list. For example, to put ABC on the first key label in our example menu, execute

{ ABC DEF } ORDER.

Actually, the DEF entry in the list is superfluous in this case. ORDER moves the variables named in the list to the start of the VAR menu in the order specified, leaving any other variables in their current order, following the final entry in the list.

[⬅] [REVIEW] is a handy way to make a quick check of the contents of the variables listed on one page of the VAR menu. Each variable name is displayed on one line, followed by a colon plus as much of the corresponding stored object as will fit on the line:



```
ABC: 123
DEF: 456
```

You can also catalog global variables using VARS and TVARS, described in section 5.8.3.

### 5.1.1 DEFINE

When the object to be stored in a global variable is an object that is permitted within an algebraic expression, DEFINE ( ⬑ DEF ) provides a convenient alternative to STO. DEFINE takes an algebraic equation of the form '*name*=*expression*' as its single argument, and stores the object *expression* in a global variable *name*. If *expression* consists of a single real or complex number, name or unit object, the stored object will be of that type. For more complicated expressions, the stored object depends on numeric execution mode (section 3.5.5.2):

- With flag −3 clear (symbolic execution), *expression* is stored as an unevaluated algebraic expression. 'A=1+2' DEFINE stores '1+2' in the global variable A.
- With flag −3 set (numeric execution), *expression* is evaluated numerically (as by →NUM), and the result object is stored. 'A=1+2' DEFINE stores 3 in the global variable A.

Notice that numeric-mode DEFINE resembles a postfix form of the BASIC language LET, providing a simple way of redefining a variable in terms of its current value. For example, 'X=X+1' DEFINE adds 1 to the current value of X, which would be accomplished in BASIC with LET X=X+1 (or usually just X=X+1, with implied LET). (Don't do this with flag −3 clear, since that leads to a circular definition--see section 3.6.1).

DEFINE can also be used to create *user-defined functions*, which are described in section 8.5.

### 5.1.2 Deleting and Renaming a Variable

The command PURGE removes from memory the variable that is specified by a global name argument. It does not error if the variable does not exist, so that you can delete a variable without bothering to check to see if it is present. When a variable is removed, its position in the VAR menu is filled in from the right by the remaining labels in the menu.

There is no built-in command for renaming a variable, but you can use the following sequence, with the original name in level 2, and the new name in level 1:

```
SWAP  DUP  RCL  SWAP  PURGE  SWAP  STO
```

A program to perform more elaborate variable moves is listed in section 5.7.4.

### 5.1.3 Cancelling STO and PURGE

The HP-48 uses its argument recovery facility (section 4.3) in a non-standard way to provide a method for recovering from an accidental overwrite of the contents of a global variable. After the STO command itself is executed, LASTARG returns the stack arguments: the variable name to level 1, and the stored object to level 2. However, if the [STO] key is used in immediate-execute mode (section 6.4.1), the resulting store differs from the normal STO command in two ways:

- The Circular Reference error is returned if the two stack arguments are both the same (global name). This prevents simple endless execution loops (section 3.6.1).
- If the named variable already existed, LASTARG returns the object that was previously stored in the variable to level 2, rather than the newly stored object. Thus you can use [⬑] [LAST ARG] [STO] [⬑] [LAST ARG] to cancel the effect of an incorrect store, restoring the stack *and* the variable to their states prior to the incorrect store.

The variable protection of the [STO] key also applies to the other keyboard store operations--pressing unshifted HP Solve variables menu keys, or left-shifted VAR or CST menu keys. It does not apply to the programmable command STO, or to [STO] when the name argument is other than an untagged global name.

A similar recovery facility works with the [⬑] [PURGE] key, when the argument is an untagged global name. In this case, LASTARG returns the purged object to level 2 as well as the name argument to level 1, so that you can undo an accidental purge by pressing [LAST ARG] [STO] . Again, PURGE executed from the command line or in a program retains the normal last argument action.

You should realize that this non-standard but useful behavior of the [STO] and [⬑] [PURGE] keys means that replacing or deleting a stored object does not immediately recover the memory associated with the object, since the object is kept in the last argument memory until replaced by the arguments of a subsequent command. You can use the command form of the operations when you want to be sure to discard the old object immediately; e.g. use ⮫ ENTRY STO ENTER instead of STO . Or you can execute another command (or a system halt) to remove the old object from last argument memory after the store or purge.

## 5.2 Directories

The HP 48 allows you to create any number of variables like ABC and DEF. When you have more than six variables, the VAR menu shows a *page* of six at time; NXT (*next page*) and ⮨ PREV (*previous page*) allow you to page forward and backward through the menu (⮫ PREV moves to the first page of the menu). However, the menu becomes cumbersome once you have more than a few pages of six variables. For this reason the HP 48 provides *directories*, which allow you to organize logical groups of variables.

The variables ABC and DEF in our example so far together constitute the *home directory*, a permanent directory that serves as the "root" of the HP 48's global variable organization. Like any directory, the home directory can be empty, as it is following a memory reset, or it can contain any number of variables. You can picture the current home directory like this:

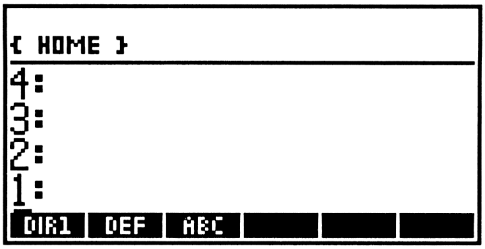
| ABC | DEF |
|---|---|
| 123 | 456 |

Each box represents a variable, showing its name and contents. The variables are shown in the same order that they are presented in the VAR menu.

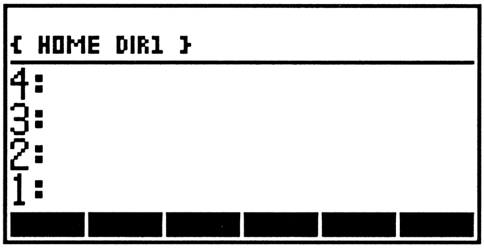
The HP 48 allows you to create variables within the home directory that themselves contain directories--groups of additional variables. This process, which can be repeated indefinitely within the new directories and their variables, allows you to organize *user memory*--the complete collection of global variables--into a hierarchical structure. To see how this works, create a directory variable DIR1:

'DIR1' ⮨ MEMORY ≡CRDIR≡

Press VAR to show the VAR menu again:



A menu label has appeared for DIR1, indicating that CRDIR (*CReate DIRectory*) has added a variable DIR1 in the home directory. The little "tab" above the label, which makes it resemble a file folder, indicates that the corresponding variable is a directory. Initially, the directory contains no variables. Now press DIR1 :

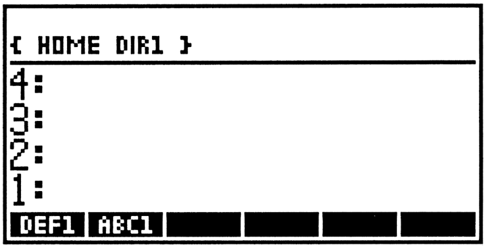


Executing a directory by name causes that directory to become the *current directory*, which by definition is the directory whose variables are displayed in the VAR menu. Since the directory DIR1 is empty, the VAR menu shows only blank menu keys at this point. Notice also that the list { HOME DIR1 } is now displayed in the second line of the status display. This list, called the *current path*, is the sequence of directories that leads to the current directory. You can return this list as a stack object by executing PATH (in the ⬅ MEMORY menu); if you later change current directories, you can evaluate the list (EVAL) to return to the directory specified by the list. (In subsequent discussions, we will simplify descriptions by using expressions like "switch to" or "go to" rather than "make current." Thus "switch to the DIR1 directory" means "make DIR1 the current directory.")

Any variables that you create while a particular directory is current become part of that directory. For example, create two new variables:

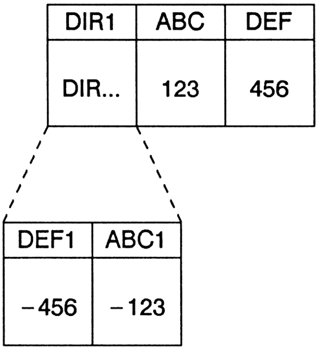
−123 'ABC1' STO −456 'DEF1' STO .

The new variables appear in the VAR menu:



Meanwhile, what has become of the variables ABC and DEF created at the start of this exercise? They are still available for execution or recall, but are not visible in the menu. For example, if you execute ABC ENTER, the value 123 is returned. This illustrates the essential property of HP48 *name resolution*: when the HP48 searches for ("resolves") a global name, it first searches the current directory. If it can not find a variable with that name there, it proceeds to search the *parent* of the current directory--the directory that contains the current directory as a variable. The search continues through the parent of the parent, and so on to the home directory if necessary.

The following diagram represents the current structure of user memory:



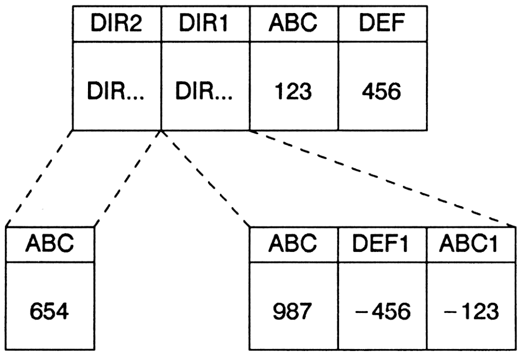
The figure shows the contents of the DIR1 directory *below* the home directory. This matches the HP48 terminology in which DIR1 is considered as a *subdirectory* of the home directory, and where the command UPDIR (⬅ UP) is named to suggest moving upwards through the user memory structure. UPDIR goes to (makes current) the parent of the current directory; HOME is equivalent to executing UPDIR repeatedly until the home directory is reached.

A key principle of HP 48 name resolution (see section 5.5) is that global variable searches always proceed *upwards* through the directory tree, but never downwards. In the current example, if you execute HOME or UPDIR to return to the home directory, then executing 'ABC1' RCL returns the Undefined Name error since the search for ABC1 does not include the DIR1 subdirectory. (The error message is somewhat inaccurate, since it is the *variable* that is not "defined", rather than the name.)

The HP 48 does permit you to have any number of variables with the same name, as long as there is only one such variable in any directory. For example, execute

```
⮣ HOME   DIR1   987  'ABC'   STO
⮣ HOME   'DIR2'  ⮢ MEMORY   ≣CRDIR≣
VAR   ≣DIR2≣   654  'ABC'   STO
```

Now user memory looks like this:



Executing ABC returns a different result when each directory is current:

```
ABC              ☞  654
⮢ UP     ABC     ☞  123
≣DIR1≣   ABC     ☞  987.
```

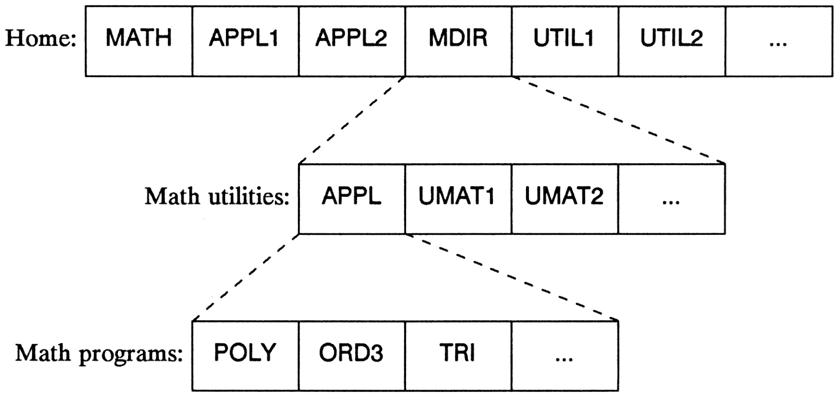
The variable searches performed by commands that change the contents of variables, such as STO, PURGE, etc., are limited to the current directory. This provides a measure of protection against the accidental destruction of variables you can't see in the VAR menu.

### 5.2.1 Organizing User Memory

The properties of directories outlined in the preceding sections suggest the following guidelines for organizing user memory:

- The home directory should contain utility variables that are needed in a variety of applications, plus directories that contain groups of variables associated with individual applications.
- Make a separate directory for each application program or set of programs, to avoid variable name conflicts and to keep the individual directories short.
- Use ORDER to arrange each directory so that the variables you need most frequently are at the start of the directory and appear at the beginning of the VAR menu. Better yet, use a custom menu (section 7.3) to show a subset of a directory's variables, in an order that won't change as you create or delete variables in the directory.
- If a program uses variables that have no use in manual operations, put those variables in a directory that is a parent of the directory containing the program. This keeps the variables from cluttering up the VAR menu that includes the program, and helps prevent the program's users from altering or deleting the variables.

The last guideline indicates a structure like the following:

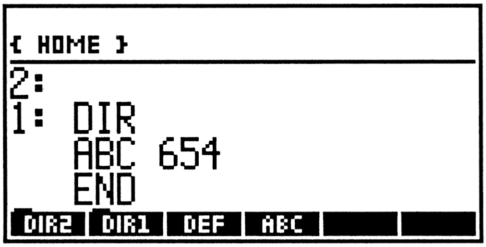


The example home directory "application" variable in the figure is MATH. The variable MATH contains the program << MDIR APPL >>, which first makes MDIR the current directory, then APPL. When you press ≡MATH≡ , therefore, you bypass the math utility subdirectory MDIR containing the programs and activate the APPL subdirectory. This subdirectory contains the directly usable application programs, POLY, ORD3, TRI, etc., that are associated with the ≡MATH≡ key. These programs use subroutines named UMAT1, UMAT2, etc., which are stored in the directory MDIR that is the parent directory for APPL.

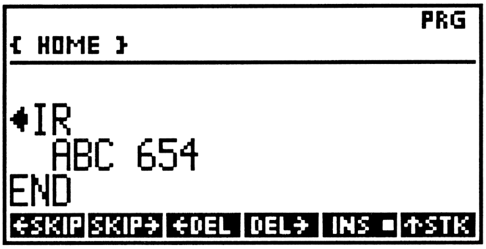
The programs in the APPL directory are those you are likely to use from the keyboard. These programs can use any of the utility programs in MDIR or in the home directory. But while the APPL directory is current, the VAR menu contains only keyboard-useful programs--you're not distracted by seeing utility programs in the menu. Also, while APPL is current, you don't have to worry about unwittingly overwriting one of the utility programs.

## 5.2.2 Directory Objects

In the discussion so far we have described a directory only as a collection of variables. However, it is important to note that a directory is itself an object, with all of the properties of a regular HP 48 object--a directory can be recalled, edited, copied, executed, etc. In the current example, you can recall the directory DIR2 by pressing [↱] [HOME] [↱] ≡DIR2≡ :

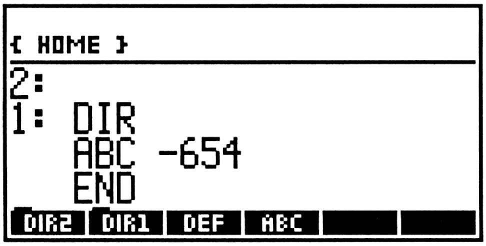

```
{ HOME }
2:
1: DIR
     ABC 654
   END
 DIR2 | DIR1 | DEF | ABC |  |
```

The directory object is now in level 1, displayed using the DIR...END syntax described in section 3.4.10. You can now create a new directory variable by storing the stack object in a new variable. This ability is convenient when you want to move the contents of a directory--you can use the same move strategy as for any other variable, as described in section 5.7.4. You can also edit a directory object: here, press [↰] [EDIT] or [▽] :



```
                          PRG
{ HOME }
◆IR
  ABC 654
END
←SKIP SKIP→ ←DEL DEL→ INS ■ ↑STK
```

Press ≣SKIP→≣ twice to put the cursor on the 654, then press [+/-] to negate the 654, and then [ENTER] :



```
{ HOME }
2:
1: DIR
   ABC -654
   END
DIR2 DIR1 DEF ABC
```

Now you have a new directory object, different from the original still stored in the variable DIR2. However, if you try to replace the contents of DIR2 with the new object by pressing [⬑] ≣DIR2≣ , the HP48 returns the Directory Not Allowed error. Because directories can contain major portions of user memory, variables that contain directories are given a special protection. You can not apply STO, or any other operation that changes the contents of an existing variable, to a directory variable. To replace the old DIR2 with a new one, then, you must first delete the old version. But there is one more level of protection that you must defeat: you can not apply PURGE to a directory variable unless the stored directory is empty--contains no variables itself (Non-Empty Directory error). Try this:

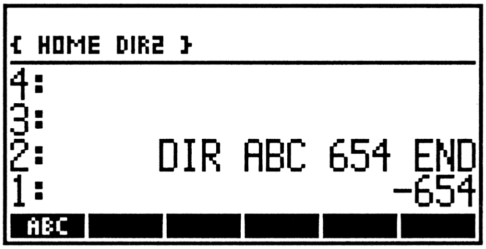
≣DIR2≣ 'ABC' [⬑][PURGE] [⬑][UP] 'DIR2' [⬑][PURGE] .

This succeeds is purging the old DIR2, so you can proceed to store the new copy by entering 'DIR2' [STO] . Now recall DIR2 (press [⬏]≣DIR2≣ ), and you can see that it contains the new directory, with the value −654 in the variable ABC.

The rule against storing into a non-empty directory variable also extends to VISIT. Executing VISIT on the name of a directory variable proceeds normally until you press [ENTER] to store the modified copy. Then the HP48 returns the Directory Not Allowed error. However, in this case, the modified directory and the variable name are left on the stack, so you can delete the original directory if you want then store the new version from the stack.

In addition to the special variable protection, directories exhibit two other peculiarities that are not shared by any other object type. To demonstrate the first, execute the following, with the copy of DIR2 still on the stack (if you have changed the stack, execute [⮣] [HOME] [⮣] DIR2 first):

DIR2 ABC [ENTER] [+/-] [⮢] ABC :



```
{ HOME DIR2 }
4:
3:
2:     DIR ABC 654 END
1:                -654
```

Here you can see the surprising effect that changing the stored directory (by changing one of the variables within it) also changed the recalled copy of the directory, now in level 2. For other types of objects, changing an object in variable has no effect on a previously recalled copy--notice that the −654 in level 1 is unchanged. If you want to modify a stored directory without changing a copy, you must first execute NEWOB (section 11.6) on the copy. [The reason for this behavior derives from HP48 memory management, which does not permit recalling objects from within unstored directories.]

The other idiosyncrasy of directories is that you can't store a directory within itself (which is not unreasonable, if you think about it). Try this, with DIR2 as the current directory:

'DIR2' [⮣] [RCL] 'XYZ' [STO] ☞ Directory Recursion *error.*

This message means that you tried to define a directory in terms of itself, which is something too hard even for the HP48 to do.

### 5.2.3 Purging Multiple Variables and Directories

In the previous section, we succeeded in removing a directory by purging its only variable, then purging the directory itself. The HP 48 provides three methods for deleting several variables simultaneously, or entire directories:

- PURGE works with a list of names as well as with a single name. Each variable named in the list is purged, starting with the first and proceeding to the end of the list. If a non-empty directory's name is encountered in the list, the Non-Empty Directory error is returned, and the variables named following the directory are not purged. The list may also contain port names (section 5.3).
- CLVAR (*CLear Variables*) deletes all of the variables in the current directory. It is equivalent to VARS PURGE, including stopping after partial completion if it encounters a non-empty directory. [For sake of compatibility with the HP 28, CLVAR can also be entered as CLUSR.]
- PGDIR (*PurGe DIRectory*) removes a directory specified by name. It does this by recursively executing CLVAR and PURGE recursively on each subdirectory until the original directory is empty. (This process can take a relatively long time if the directory is large.) Because it is such a dangerous command, PGDIR is deliberately buried in the third page of the ⬅ MEMORY menu.

## 5.3 Ports

A *port* is a independent portion of memory that is established to contain *libraries* and *port* variables. The HP 48 system defines three such ports: ports 1 and 2, which correspond to memory plugged into the HP 48SX's two memory card slots, and port 0, which is permanently defined in main RAM.

There are two methods for determining the contents of a port: the LIBRARY menu, which is explained in subsequent sections, and the command PVARS. With a argument of 0, 1, or 2 to specify a particular port, PVARS (*Port VARiableS*) returns a list analogous to that of VARS (section 5.7.3), containing the number of each library and the name of each variable in the corresponding port, with each object in the list tagged with the port number. PVARS also returns (to level 1) one of the following objects:

| **Object** | **Meaning** |
|---|---|
| *real number* | Amount of free memory left in the port (RAM). |
| "ROM" | The port contains ROM or write-protected RAM. |
| "SYSRAM" | The port memory is merged (the contents list will be empty). |

The free memory reported by PVARS for port 0 is the same amount returned by MEM.

PVARS provides a convenient means for deleting all of the objects in a port. For example,

```
0 PVARS DROP PURGE
```

removes all of the objects from port 0.

## 5.3.1 Port Variables

One advantage of storing an object in the home directory is that it is always accessible by name, no matter which directory is current. This makes the home directory the logical place to store general purpose variables, such as the program FIND described in section 5.7.3. However, if you store too many variables in the home directory, the associated VAR menu becomes unwieldy.

*Port 0* is another portion of memory that plays a role similar to that of the home directory. You can create one or more variables there; the variables are universally accessible by name; and there is an automatic menu associated with the port. Port 0 is always available, but in an HP 48SX you can also create two more similar memory regions, port 1 and port 2, by inserting RAM cards into one or both card slots. We will use port 0 as an example here, but the properties of port 0 also apply to port 1 and port 2.

A named object stored in a memory port constitutes a *port variable*. Like a global variable, it is a combination of a text name with any object, providing access to the stored object by means of the name. The HP 48 does not define a unique name object type for port variables as it does for global variables, local variables and commands. Instead, the commands that access port variables recognize global or local names tagged with a port number 0, 1, or 2, as designating port variables. For example, :0:ABC RCL recalls the object stored in a port variable ABC in port 1. We will refer to such tagged names as *port names*.

There are six commands that accept port name arguments of the form :*n*:*name*:

| | |
|---|---|
| STO | creates a new port variable *name* in port *n*. The object to be stored is taken from level 2, and the port name from level 1. |
| RCL | recalls the object stored in the specified port variable. |
| EVAL | executes the object stored in the specified port variable. |
| PURGE | purges the specified port variable. PURGE will also accept a list of global names and port names, and purge all of the variables named in the list. |
| PRVAR | prints the object stored in the specified port variable. Like PURGE, PRVAR will operate on a list of global names and port names. |

ARCHIVE makes an archival copy of user memory and stores it in the specified port variable.

The port number *n* used with these commands can be 0, 1, or 2, or, except for STO and ARCHIVE, the character &. When the latter is used, the HP 48 searches for a variable with the specified name in port 2, port 1, and port 0 in that order; if no match is found, the tag is ignored and the name is treated as an ordinary untagged global or local name. This feature allows programs to use objects that may move around among the ports and main memory. ARCHIVE also accepts a name tagged with :IO:, for which the archived user memory is transmitted to the serial port as a Kermit file.

Only the six commands listed above recognize port names. Other commands ignore the tag on a port name and operate on the untagged name. This can lead to some surprises: for example, :0:ABC STO+ always attempts to add (see section 5.7.2.1 ) to the contents of a global or local variable ABC, even when there is a port variable ABC in port 0.

#### 5.3.1.1 The LIBRARY Menu

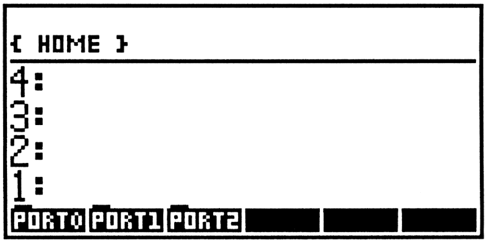
As an example of creating port variables, enter the following:

234 :0:ABC [STO] 567 :0:DEF [STO] .

Assuming that the home directory still contains the variable ABC created earlier in the ongoing example, executing ABC still returns the value 123 stored in that global variable. In order to return the value just stored in the port variable ABC, you must include the port-number tag:

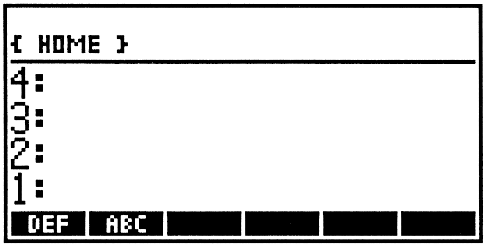
:0:ABC [EVAL] ☞ 234.

The LIBRARY menu ([◁] [LIBRARY] ) is an automatic operational catalog of port variables, analogous to the VAR menu for global variables. When you press [◁] [LIBRARY] , you see a display similar to this:



```
{ HOME }
4:
3:
2:
1:
PORT0 PORT1 PORT2
```

You will always see at least the PORT0 menu key label. If you have memory cards plugged into card slots, you will also see labels for PORT1 or PORT2, or both. Furthermore, if you have libraries attached to the current directory (see section 5.2), there will be an additional menu label for each library. In the ongoing example, press ≣PORT0≣ :



```
{ HOME }
4:
3:
2:
1:
```

This activates the *port 0 menu*, where you see menu keys for the port variables ABC and DEF created earlier. Pressing either of the labeled menu keys returns the number stored in the corresponding variable:

≣ABC≣ ☞ 234.

As for VAR menu keys, when you press a port variable menu key, the object stored in the specified port variable is executed. The right-shifted menu keys also perform a RCL like a VAR menu key; the left-shifted menu key *attempts* to execute STO, but like STO itself, this fails because you can't store into an existing port variable (section 5.3.1.2).

Because a port name is a tagged object, it is effectively already quoted and therefore you do not quote a port name to enter it on the stack for use as an argument. If you do attempt to enter a port name within single quotes ' ', you will obtain an Invalid Syntax error message, because tagged names are not allowed in algebraic expressions. You can nevertheless use LIBRARY menu keys to enter port names: press [↱] [ENTRY] first to activate program entry mode, then press the appropriate menu key. This enters the port variable name preceded by the appropriate port number tag.

### 5.3.1.2 Altering Port Variables

Port variables are intended for object storage that is somewhat more permanent than that offered by global variables. For this reason, the contents of port variables can not be changed once they are created, short of deleting them with PURGE. STO returns the Object In Use error if you attempt to overwrite the contents of an existing port variable. Furthermore, you can't delete a port variable if the stored object is *referenced*, in which case PURGE returns the same error message as STO. *Referenced* means that a stored object (or part of it) has been recalled by one means or another, and the recalled copy is still present--on the stack, in argument recovery or stack recovery memory, on the program return stack, or in a local variable. (Specifically, this means that there is a pointer to the port variable object in any of these areas--see section 4.7). To succeed in purging a port variable, you must first remove all such references to the object, either individually, or collectively by executing a system halt $\boxed{\text{ON}}$ - $\boxed{\text{C}}$ ). Some references may be very subtle; for example, if you execute a program in a port variable which uses DOERR with a string argument defined in the program, the program will be referenced for the sake of the ERRM command until some subsequent error generates a new error message.

If you want to delete a port variable but still keep a copy of its stored object, you must recall the object and either store it in a global variable or another port variable, or execute NEWOB (section 11.6) with the object in level 1. This creates a new copy of the object and unreferences the port variable. Then you can use PURGE to delete the variable.

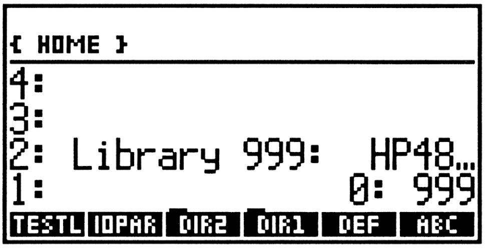
### 5.3.2 Libraries

*Commands* are named objects that are stored for execution only, and which are not available for recall or modification. A collection of commands is called a *library*. Libraries are objects (section 3.4.11), which allows you to move them around within the HP48, primarily to transfer them from a personal computer or from HP48 to HP48. When you are dealing with a library as an object, the commands that comprise the library are not visible or accessible. To activate the contents of a library, it must be stored in a port and *attached* to a directory. On the stack, a library object is displayed simply as Library *n*, *title*, where *n* is the unique *library ID* number assigned to the library, and *title* is its text title.

All of the HP48's built-in commands and program structure words (section 9.2) are contained in libraries permanently stored in built-in ROM. The details of their organization into libraries is not important; the only place where the division manifests itself is in the various error numbers, the leading digits of which identify the library in which the error occurred.

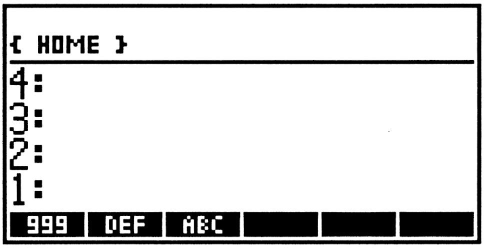
Externally created libraries (at this writing, there are no facilities for creating libraries on the HP48 itself, although it is possible in principle) must be stored in a port in order for its commands to be available. In a plug-in ROM card, the libraries are permanently stored; in a port containing RAM, including port 0, you must store a library there using STO. The "name" required by STO in this case is just the (real) port number. You may also use any number tagged with the port number, such as :0:123; this allows STO to use the same port-tagged library ID as used by RCL to recall a library from a port to the stack, or by PURGE to delete the library.

Imagine that you have a library on the stack, ID number 999. (If you have the *HP 48 Insights Program Disk*, you can transfer this sample library from your personal computer to the HP 48, and follow along with the example.) To store the library in port 0, enter :0:999



```
{ HOME }
4:
3:
2: Library 999:  HP48...
1:               0: 999
```

Press STO, then ⬅ LIBRARY PORT 0 :

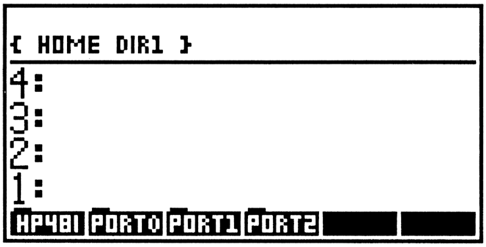


```
{ HOME }
4:
3:
2:
1:
```

The ABC and DEF labels correspond to the port variables created in the ongoing example, in section 5.3.1.1. The new entry 999 indicates that the library has been stored in port 0. However, at this point you still don't have access to the commands in the library. First, you must turn the HP 48 off, then on. You will observe that this causes a system halt (section 5.8), so you should store any objects on the stack that you want to keep before turning the HP 48 off. The system halt occurs when the HP 48 detects that a library has been added to a port; during the halt operation, the HP 48 builds a table of all of its current libraries that it uses to find the libraries later.

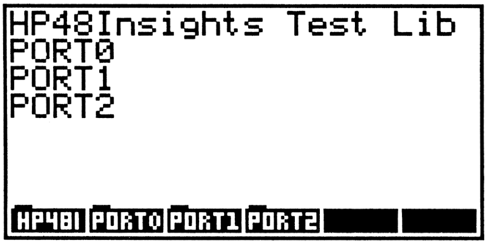
The last step in making library commands accessible is to *attach* the library to a directory. When a library is attached to a directory, the libraries commands are accessible for execution or entry into a composite object whenever that directory is in the current path (section 5.2)--just like the global variables in that directory. Each directory may have one library attached to it, except for the home directory, which may have any number of attached libraries (including the built-in libraries, which are permanently attached there). To attach a library to a directory, you make that directory the current directory, then execute *id* ATTACH, where *id* is the library's ID (expressed as a real number). The library does not need to be present when ATTACH is executed, but the attachment will have no consequence unless the library is installed in a port.

In the current example, execute HOME DIR1 to make DIR1 the current directory, then execute 999 ATTACH (the menu key for ATTACH is in the second page of the ⬅ MEMORY menu). Now press ⬅ LIBRARY :



```
{ HOME DIR1 }
4:
3:
2:
1:
HP48I PORT0 PORT1 PORT2
```

There is a new entry in the menu: HP48I , arising from the newly attached library. In addition to the library ID, a library contains a library *title*, a text string that describes the library. The first four or five characters of the title is used to label the library menu key. To see the full text of the title (up to 23 characters), use ⬅ REVIEW :

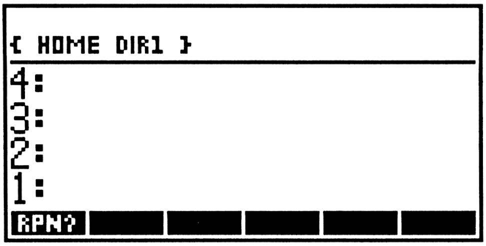


```
HP48Insights Test Lib
PORT0
PORT1
PORT2



HP48I PORT0 PORT1 PORT2
```

The tab on a library's menu key label indicates that the key activates yet another menu. Press HP48I :



This menu is created from the objects defined in the library. Pressing ≡RPN?≡ returns the string "I love RPN!". If you press [⬅] [{ }] ≡RPN?≡, the list { RPN? } is placed on the stack. The object RPN? within the list (you can take it out of the list with OBJ→) is an *XLIB name object* (section 3.6.3). It is similar to a global name, in that executing it executes an object stored with the name. However, you can not recall or view the stored object itself.

An XLIB name object does not actually contain the text of the name, which is stored in the library. You can see this by purging the library (:0:999 PURGE); if you do so, the list on the stack becomes { XLIB 999 0 }. Since the library is unavailable, the HP 48 does not know the XLIB name text, and reverts to the two number codes that constitute the XLIB name object. The two numbers show that the name corresponds to command 0 in library 999. The fact that XLIB names contain number codes rather than text makes them more compact and speeds up execution of library commands.

For a library intended to be attached to the home directory (which makes their commands universally available), it is common for the library to attach itself automatically to the home directory. This is useful, since unlike attachments to subdirectories, home directory attachments are cleared by a system halt. During a system halt, each library is given a chance to execute its *configuration program*, which can prepare any special HP 48 resources needed by the library, including attaching the library to a particular directory. The HP 82211A Solve Equation Library Application Card contains several libraries, each of which automatically attaches itself to the home directory. To access the card's programs, therefore, you have only to insert the card in a port and turn the calculator on.

[When you transfer a library from a personal computer to the HP 48, you can not transfer the library directly to a port, but must transfer it first to a global variable. Then you can recall the library to the stack and store it in a port. It is generally a good idea not to leave a copy of a library in a global variable after you have copied it to a port, not only to conserve memory, but because the Recover RAM process associated with an accidental or deliberate memory reset ( [ON] - [A] - [F] ) does not work well when there are libraries stored in global variables.]

#### 5.3.2.1 Other Library Commands

In addition to STO and ATTACH, the following commands are associated with libraries:

- PURGE. To remove a library from a port, execute *:n:ID* PURGE. As in the case of port variables, you will be unable to purge a library if it is referenced in any way (Object in Use). For a library, a reference can be obvious, such as recalling the library to the stack. But references can also be more subtle--for example, if a library program is executed and creates an object that is left on the stack or in a local variable, the library will be referenced until the object is removed. An extreme case of this situation is a library that uses DOERR (section 12.2.1) with a string argument; the library will remain referenced for the sake of ERRM (section 9.6) until a subsequent use of DOERR changes the error message string. (In HP 48SX versions 1A - 1D, even a system halt does not clear the ERRM reference.) A library attached to the home directory is "permanently" referenced; you must detach it (see below) before purging.

  When you purge a library, you may see the display jump briefly. This is caused by the movement of display memory arising from the removal of an entry in the HP 48's internal table of libraries; it is quite harmless.

- DETACH. The inverse of ATTACH is DETACH, which detaches a library specified by number from the current directory. A common reason to detach a library is to disable the commands in that library. For example, a library might define a new meaning for SIN; to use the built-in version you must either detach the library or change to a directory in which the library is not in the current path.

- LIBS. This command catalogs the libraries attached to the current directory, returning for each library the full library title, library number, and the number of the port containing the library. In any directory except the home directory, there can only be one attached library, so LIBS returns a list of three elements:

  { *"Title" library-number port-number* }

  If no library is attached, LIBS returns an empty list. In the home directory, the list can contain a multiple of three elements, with one group of three for each attached library. Note that LIBS provides the only means for viewing a library's full title when the title is longer than 22 characters.

## 5.3.1 Plug-In Ports

When you plug a memory card into either HP48SX card slot and turn the calculator on, the calculator checks the card to determine whether the card memory contains a valid sequence of libraries and port variables. When the HP48 cannot recognize the card contents, the message Invalid Card Data is displayed. If the card is ROM, the card is not usable in the HP 48, and should be removed. If the card contains RAM, you can ignore the message--the first attempt to merge the card memory or store a library or port variable there will organize its memory properly, and prevent further Invalid Card Data errors.

When a newly inserted card is recognized as valid by the HP48, the card memory is configured as an independent port, and an entry for the port will appear in the LIBRARY menu. The libraries and variables in the card are then available using the procedures outlined in the preceding parts of this chapter. If the card contains RAM, you also have the option to leave it as an independent port, or to merge the card's memory with main RAM. As long a card is configured as independent RAM, you can remove and replace it at will (remember to turn the calculator off when inserting or removing cards), or move it to another HP48. An independent RAM card is a very fast and convenient means for transferring objects from one calculator to another.

The command MERGE merges the memory in a plug-in RAM card into main memory. MERGE takes a real number 1 or 2 as an argument to identify the slot containing the card to be merged; if the argument is another number, or if the read/write switch on the card is set to write-only, the Port Not Available error is returned. If the card memory contains port variables and libraries, these are moved automatically to port 0. The amount of free memory returned by MEM is increased by the amount of memory in the card (32K or 128K bytes) less the memory used by the port variables and libraries. Once you have merged card memory, however, you can not remove the card without potentially corrupting memory contents. If a merged card is removed (including the case where the calculator is dropped hard enough to jar a card loose), the HP48 warns you of impending disaster by beeping and displaying Replace RAM, Press ON. By following that instruction, you can preserve memory contents; otherwise the contents of main memory and the card memory are lost.

The reverse of merging a card's memory is to *free* it using FREE. FREE needs two arguments: one a real number (level one) 1 or 2 to specify the port to be freed, and the other (level 2) to select objects from port 0 to the moved into the newly freed port. The latter argument can either be a library number, or the name of a port 0 variable, or a list containing any mixture of library numbers and names. If you just want to free a port without moving any objects there, use an empty list.

## 5.3.2 Archiving Memory

In addition to storing individual objects in port variables, the HP 48 allows you to store a copy of user memory, including current alarms and key assignments, in a port variable. This archival copy of memory can then be used to restore the calculator to a previous state, especially after an accidental or deliberate memory loss, or to copy the contents of one HP 48 into another. You can make an archival copy quite safe by saving it in an independent RAM port, then setting the read/write switch on the card to read-only (archiving to a personal computer via the serial port is also a good alternative).

The ARCHIVE command takes as its argument a port number, then creates a port variable in the specified port, to store a replica of the home directory. A good choice for the port name is one that represents the date on which the archive was made, such as NOV2590 or APR2191, to help you choose among multiple archive copies.

Once an archive is made, you can replace the current user memory with the archival version by executing *port-name* RESTORE, where *port-name* specifies the port variable created by ARCHIVE. RESTORE terminates by performing a system halt, so that the display blanks momentarily then shows an empty stack, with the MTH menu and the path { HOME }. Note: RESTORE begins by executing the equivalent of PGDIR on the home directory. This can be very time consuming when user memory is large and contains a lot of subdirectories. In such cases, it is much faster to perform a memory reset ( ON - A - F , with the NO option) before executing RESTORE.

If you recall the contents of a port variable created by ARCHIVE, you will see what appears to be an ordinary directory object. However, this directory is unusual in that it may contain a "nameless" subdirectory (if you use ◁ EDIT to copy the directory to the command line, you can see a DIR entry with no name preceding it). The subdirectory contains three variables: Alarms, UserKeys, and UserKeys.CRC. The first two, as you might guess, contain the alarm catalog and the user key assignments; UserKeys.CRC contains a memory checksum that is used by the HP 48 to verify the integrity of the key assignments. The alarms and key assignments are kept in the nameless subdirectory in order to prevent their being accidentally or deliberately edited into a form that might corrupt the HP 48 system.

ARCHIVE only saves the contents of user memory; in particular, it does not save the current flag values or the contents of port 0. The program XARCHIVE listed below demonstrates a method of extending ARCHIVE to save these objects as well. XARCHIVE takes an argument 0, 1, 2, and calls the program DATENAME to automatically create a destination name (for either a port variable or a computer file). The name has the form *n:mmmdd*, where *n* is the argument, *mmm* is a three letter abbreviation for the current month, and *dd* is the two-digit day of the month. Then XARCHIVE creates a temporary directory archtemp, moves all of the port 0 objects to that directory, and also saves the current flag values there. Furthermore, XARCHIVE creates a program fixup in the home directory, which is also saved as part of the archive. After the archive is made, XARCHIVE moves the objects back to port 0, and deletes the temporary variables. XARCHIVE requires enough free memory to make a copy of the largest port 0 object; if it runs out of memory, it still restores the contents of port 0 and deletes temporary variables. After XARCHIVE is finished, you should turn the HP48 off then on to reattach any port 0 libraries.

To later rebuild HP48 memory from the archival memory made by XARCHIVE, execute RESTORE as usual using the name of the variable or file created by XARCHIVE. Then press VAR FIXUP . The latter step restores the saved port 0 objects and flags, and purges the temporary variables, including fixup. (It is not possible to combine RESTORE and fixup into a single program, because RESTORE performs a system halt, preventing execution of any program objects following it.) If the archive includes any port 0 libraries, you should turn the calculator off then on to reattach those libraries.

XARCHIVE also lets you substitute the string argument "IO" instead of a port number. In that case, the archival user memory is transmitted as a backup object (section 3.4.12) to either the serial or infrared output port, for storage on a personal computer or another HP48. To rebuild memory from an external archive, you must transfer the backup object into a global variable using a Kermit transfer. Then recall the backup object to the stack, and execute RESTORE followed by FIXUP, as before.

| DATENAME | *Create a Name from the Current Date* | 3A18 |
|---|---|---|
| | *level 1* | *level 1* |
| | ☞ | *'mmmdd'* |

| | |
|---|---|
| « RCLF -42 CF DATE TIME TSTR | Get the time string. |
| "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC" | |
| OVER 5 6 SUB OBJ→ | Get the month number. |
| 1 - 3 * 1 + DUP 2 + SUB | Get the month name. |
| SWAP 8 9 SUB | Get the day number. |
| STD + "'" SWAP + OBJ→ | Make the name. |
| SWAP STOF | Restore flag −42. |
| » | |

| XARCHIVE | *Extended Archive* | 3D84 |
|---|---|---|
| | *level 1* | |
| | "IO" | ☞ |
| | *n* | ☞ |

```
<< DEPTH PATH RCLF 0 → m d p f e    Save the port, depth, path, flags, signal.
 << IFERR HOME STD                  Trap errors.
   DATENAME m →TAG                  Tag the name with the port number.
   << WHILE 0 PVARS DROP DUP SIZE   Program for use after RESTORE:
     REPEAT 1 GET                   Get the next port 0 name/number.
      IF DUP OBJ→ DROP TYPE NOT     If it's a library,
      THEN DUP DETACH               detach it.
      END PURGE                     Purge the object.
     END DROP
     archtemp flags STOF →P0        Restore archived flags and port 0 objects.
   >> 'fixup' STO
   'archtemp' DUP CRDIR EVAL        Create temporary directory.
   << WHILE VARS DUP SIZE 2 >       Program to move objects back to port 0:
     REPEAT 1 GET DUP RCL
       OVER PURGE SWAP              Recall and purge the variable.
      IF OVER TYPE 16 SAME          If it's a library,
      THEN DROP 0                   Substitute a number for the name.
      END 0 →TAG STO                Store in port 0.
     END DROP CLVAR HOME
     { archtemp fixup } PURGE       Delete temporary variables.
   >> →P0 STO f 'flags' STO         Save program and flags in archtemp
  THEN 1 'e' STO                    Signal that an error occurred.
  ELSE                              No error so far.
   IFERR                            Trap error in moving objects or archiving.
    WHILE 0 PVARS DROP DUP SIZE     If there are port 0 objects...
    REPEAT 1 GET                    Get the next port name.
     DUP RCL OVER OBJ→ DROP
     IF DUP TYPE NOT                If the object is a library,
     THEN DUP HOME DETACH           then detach it, and
      archtemp "'L" SWAP + OBJ→     make a name from its number.
     END STO PURGE                  Store in archtemp, purge from port.
    END DROP
    IF m TYPE NOT
    THEN DUP PURGE                  Purge existing archive.
    END ARCHIVE                     Make the new archive.
   THEN 1 'e' STO                   Signal that an error occurred.
   END →P0                          Return objects to port 0.
  END p EVAL                        Restore path.
  IF e                              If an error occurred,
  THEN DEPTH d - 1 -
   DROPN ERRN DOERR                 clean up the stack and report.
  END
 >>
>>
```

## 5.4 Local Variables

Although we will defer a detailed discussion of local variables to section 9.7, they need to be described briefly here in the context of storing objects. Local variables are variables created for temporary use by a procedure. They are handy because they can have any name without conflicting with global variables (or another procedure's local variables), and because they are automatically deleted when their defining procedure completes execution.

Local variables are created by the program structure words → (section 9.7) and FOR (section 9.5.1). For example, enter the following program:

```
<< 'JACK'  'JILL'  →  local1  local2
   <<  HALT
   >>
>>
```

Store the program in the variable HILL, in directory DIR1:

[⮣] [HOME]   [VAR]   [DIR1]   'HILL'   [STO] .

Now execute the program--press [HILL] . Notice that the HALT annunciator turns on, but nothing else visible happens. But if you type local1 [ENTER] , the name 'JACK' is returned to the stack. When the program executes, the → creates two local variables (as many as there are names following the arrow), storing in them two objects taken from the stack (one for each name). The inner program that follows the final name local2 (and marks the end of the series of names) defines the "duration" of the local variables: the variables are maintained during the program's execution, then deleted by the closing >>. In this case, execution is suspended by the HALT (section 12.2), so the two local variables will remain available until you press [⮢] [CONT] to finish the program.

The names local1 and local2 are *local names*, which are a different object type than the global names used so far in this chapter. As mentioned in section 3.6.2, executing a local name recalls the object stored in the corresponding local variable without executing it, but otherwise local names are similar in use to global names.

For local variables, there is no automatic catalog like the VAR menu or the LIBRARY menu. A portion of RAM containing local variables is called a *local memory*, and is essentially invisible other than by recalling the stored objects. [Like the stack, a local variable does not contain a copy of an object stored there, but only a pointer to the object. Copying an object from a global variable to a local variable, for example, only requires enough memory for the name text plus a few additional bytes of overhead.]

## 5.5 Name Resolution

Figure 5.2 is a diagram of HP 48 memory showing schematically all of the named objects we have created in this chapter's ongoing example. The figure also has an entry for the built-in command libraries, to show where they fit logically. Finally, the local memory containing the local variables local1 and local2 created in the preceding section is shown associated with the program HILL (here we are assuming that that program is still suspended).

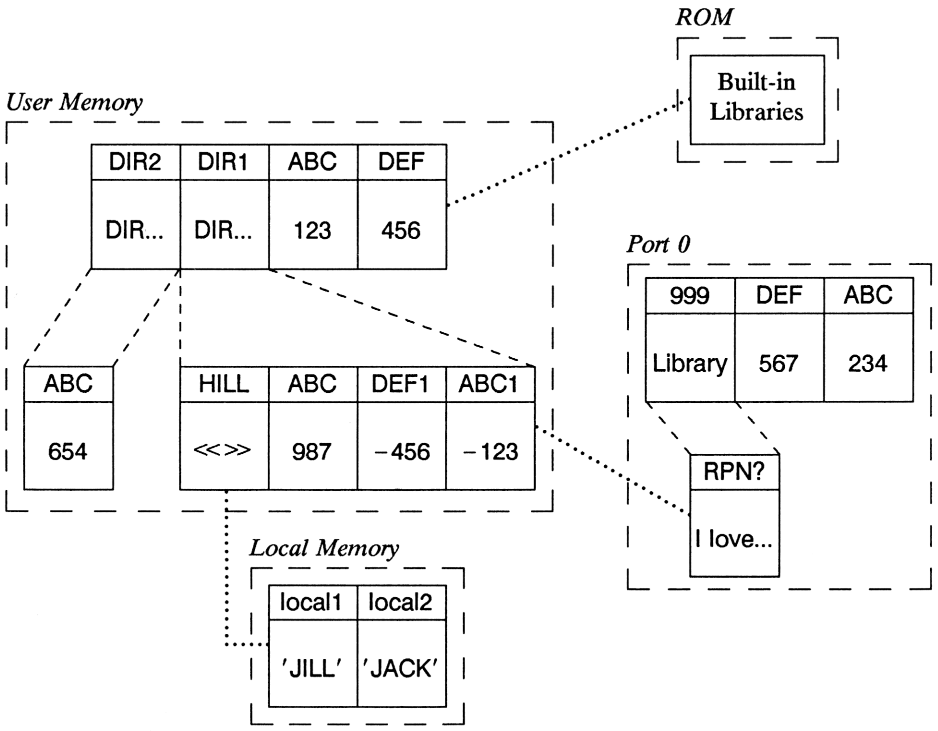


**Figure 5.2. Example Memory Organization**

The figure is helpful in explaining the details of HP 48 *name resolution*, the process by which the HP 48 creates name objects and finds named objects. Name creation and name finding are similar but distinct processes. The first takes place when a command line is entered and the HP 48 creates name objects from the command line text. The second happens when a name is executed or recalled, and the HP 48 must find the stored object associated with the name.

### 5.5.1 Command Line Entry

A series of non-delimiter characters in the command line that does not start with a number character (digit or fraction mark), and is not enclosed by string quotes " " or tag colons : :, is presumed to be a name. The type of name object created by ENTER is determined by a search through existing named objects for a name that matches the command line name. The precedence of the search is as follows:

1. If the name is contained within a local variable structure (section 9.7) in the command line, and matches one of the names defined for that structure, the name is entered as a local name.
2. If there is an local memory present that contains a local variable with a matching name, the name is entered as a local name.
3. If the name matches a global variable anywhere in the current directory, the name is entered as a global name.
4. If the name matches a command name in a library attached to the current directory, the name is entered as an XLIB name.
5. The preceding two steps are applied to the parent directory, and its parent, and so on back to the home directory. If the name is matched, it is entered as a global name or an XLIB name, as appropriate. If the search proceeds to the home directory, all of the libraries attached there are searched.
6. If the name matches a built-in command name, the name is replaced with the built-in object.
7. If the name is not matched in any of the preceding steps, it is entered as a global name.

▪ Examples from Figure 5.2 (DIR1 is the current directory, and the program HILL is currently suspended):

- → X << X Y >> local1 ABC. When this command line is entered, X and local1 are entered as local names. The first X is local because it is one of the local names defined by the arrow; the second X because it is included within the local variable program. local1 is also entered as a local name, because local1 is a local variable in the local memory associated with the suspended program HILL. ABC is entered as a global name, because it is matched by a global variable in the parent of the current directory. Y becomes a global name because it is not matched by any global or local variable, library command, or built-in command.

- → DEF1 << RPN? DEF1 >> DEF1. Here the first two DEF1's become local names, even though DEF1 is a global variable in the current directory, because DEF1 is defined as a local name in the local variable structure. The third DEF1 is entered as a global name, since it is not entered within the local variable program. RPN? is entered as an XLIB name, since it is not one of the program's local names, and is first matched by the library command in the library attached to DIR1.
- SIN RPN? → SIN RPN? << RPN? SIN >> COS. The first occurrence of SIN is entered as a command name; the first occurrence of RPN? is entered as an XLIB name. However, the subsequent uses of these names are entered as local names, because their assignment by → takes precedence over their presence as built-in or library command names. COS is entered as a built-in program object. The restriction that global names can not match built-in command names does not apply to local names. [This restriction is a property of the command line parser; the RPL language puts no such restriction on global names in general. The restriction is primarily to enforce structure rules for algebraic expressions.]

The rules described here for command line entry also apply to execution of OBJ→ (or STR→) on a string object, and to the processing of object files that are transferred to the HP 48 via an ASCII Kermit transfer.

### 5.5.2 Executing Name Objects

When a name object is used to find the object stored with the name, the process of searching for the named object is similar to that used during command line entry. However, since the type of name object is already known, the search can be more restrictive:

- For *global* names, the search is through user memory, starting in the current directory. Whether the search extends to parent directories depends on the nature of the operation using the name. For simple execution of the name, and for use with RCL, the search is made first in the current directory, and continues if necessary through all parent directories until the name is matched. For all other commands, the search is restricted to the current directory. The first variable checked is the leftmost variable in the VAR menu, nominally the newest variable unless the order has been changed by ORDER. You can achieve faster program execution by placing the global variables a program uses at the start of the current directory.
- For local names, the search extends through current local memories, starting with the newest.
- For XLIB names, no name matching is necessary; the stored object is found by means of the library and command numbers stored in the XLIB name, and the table of object locations that is part of each library.

Notice that the use of one type of name will never find an object stored with another type of name.

The resolution of XLIB names is usually significantly faster than that of global and local names. Resolving a local name is usually faster than resolving a global name, because local memories typically contain only a few variables. If a program stored in one directory frequently uses a variable in a parent directory or in another branch of user memory altogether, the program will run faster if it recalls the remote object once and stores it in a local variable, then retrieves it from the local variable for each subsequent execution.

### 5.5.2.1 Resolution Failures

When you execute a global name for which no corresponding global variable exists anywhere in the current path, the HP 48 just returns the name to the stack (this property of global names is central to the HP 48's symbolic algebra capabilities). However, in all other cases of name object resolution, an error is reported if no stored object is found. The error depends on the type of name, and the particular use:

| **Type** | **Execution** (EVAL, etc.) | **Recall** (RCL, GET, etc.) | **Store** (STO, PUT, etc.) |
|---|---|---|---|
| Global name | *no error* | Undefined Name | *no error** |
| Local name | Undefined Local Name | Undefined Name | Undefined Name |
| XLIB name | Undefined XLIB Name | Bad Argument Type | Bad Argument Type |

*Unless the variable is a non-empty directory.

The recall and store errors for XLIB names occur because those operations are not allowed, regardless of whether there is a corresponding library command.

## 5.5.3 Path Names

When one directory is current, but you want to recall a variable in another directory that is not in the current path, you can switch the HP 48 to the second directory so that the variable becomes available. If you save the original path first (using PATH), you can easily return to the original directory after using RCL. However, this procedure is a little cumbersome for repeated use, so the HP 48 provides an alternate method called a *path name*. A path name is an extended form of a variable name, where the variable's global name is entered in a list, preceded by the names of the directories that make up the path to the variable's directory. In general, a path list has this form:

$$\{\ \mathit{directory}_1 \quad \mathit{directory}_2 \quad \cdots \quad \mathit{directory}_n \quad \mathit{variable}\ \}$$

The first object in the list can be HOME or a directory name; of the remaining objects, all but the last must be directory names. The path defined by the directory names can be any path that will lead to the desired directory, but usually it is most convenient to start the list with HOME so that the path name will be usable no matter what directory is current.

Using a path name as an argument for RCL, then, is equivalent to 1) saving the current path, 2) switching to the directory defined by the path name, 3) recalling the named variable, and 4) restoring the original path. For example, if in our example user memory DIR1 is the current directory, recalling ABC returns 123, the value of ABC in the home directory. However, { HOME DIR2 ABC } RCL returns 654, the value of ABC in the DIR2 directory.

The HP 48 makes no special provision for the use of path names as described so far by EVAL, since the ordinary behavior of lists with EVAL makes path names suitable arguments. But notice that { HOME DIR2 ABC } EVAL, for example, is not quite equivalent to { HOME DIR2 ABC } RCL EVAL:

- { HOME DIR2 ABC } EVAL switches to the DIR2 directory (before executing ABC); { HOME DIR2 ABC } RCL EVAL does not.
- { HOME DIR2 ABC } EVAL evaluates the *name* ABC, whereas { HOME DIR2 ABC } RCL EVAL evaluates the *object* stored in variable ABC. The difference is significant if the stored object is a list, a directory, or an algebraic object (see section 3.3).

Also, a list argument for EVAL can contain any arguments, whereas a path name for RCL can contain only HOME and global names.

There is yet another extension to path names, which does apply to EVAL as well as to RCL. When one of these commands is applied to a path name list that is tagged with a port number 0, 1, or 2, the command is directed to the specified variable in a directory *stored in a port variable*. That is, the first name in the list is the name of a port variable that itself contains a directory. The remaining names specify a path within that directory to the desired variable. (In this case, you do *not* want the path name list to start with HOME.)

For example, try copying the directory DIR2 from our sample home directory to port 0:

[⮫] [HOME] [⮫] [DIR2] :0:DIR2P [STO]

Now you can recall the contents of the variable ABC in the port variable by executing

:0:{ DIR2P ABC } RCL ☞ 654.

This feature is especially useful when you have saved a copy of a large directory in a port variable, and want to retrieve the contents of a particular variable, but there isn't enough free memory to copy the directory to user memory.

You can also use the "wildcard" tag & for path names. With that tag, the HP 48 searches port 2, port 1, and port 0 for the specified port variable; if it is not found, the untagged path name is used to find the variable. In the latter case, EVAL switches to the directory specified by the path name--if the directory is found in a port variable, then the current directory does not change.

## 5.6 Named Objects vs. Registers and Files

Traditional calculators store data in fixed memory locations called *registers*, which are identified by a register number or letter. These calculators' programs are stored separately from the data registers, but the programs too are commonly specified by a number or letter; some advanced calculators permit multi-character program names. Computers, on the other hand, store both programs and data in *files*, which have multi-character names and are not generally limited in size or number. The HP 48 combines elements of the memory management of both traditional calculators and computers, but is generally closer in spirit to the latter. The *named object* is the closest analog in the HP 48 to a computer file or a calculator register. A named object is an object that has been stored in memory elsewhere from the stack, along with a text name that provides identification of and access to the object.

In many respects it is appropriate call named objects *files*, especially global variables, port variables, and library commands, but there are some differences between typical computer files and HP 48 named objects:

- HP 48 objects can exist independently of their names, that is, objects can be created, manipulated, changed, and executed without ever being named. Common computer operating systems, without any user-accessible structure analogous to the HP 48 stack, require you to create and store everything as named files.
- All HP 48 objects are automatically executable, either directly or by name when they are stored. Computer files must be designated as executable, e.g. executable MS-DOS files are named with the extensions .EXE, .COM, or .BAT. Those that are not executable are intended only for use as data, such as text files.
- The name associated with an HP 48 object does not "type" the object in any way, as does the extension on computer file names. Any type of object can have any name. You may choose names for objects that suggest the objects' uses, but this does not

affect the execution properties of the objects.

- Access to HP 48 stored objects is provided by *name objects* (section 3.6). This fact is central to the HP 48's symbolic capabilities--name objects can be substituted in algebraic and other operations on stored objects.
- HP 48 named objects do not record their times of creation or modification.

## 5.7 Additional Global and Local Variable Operations

The commands described in this section apply to global and local variables, but not to port variables.

### 5.7.1 Recalling Values

There are two fundamental ways to "recall" the value of a variable:

- *Execute a name object.* Executing a global name executes the object stored in the named variable (section 3.6.1). For data objects and algebraic objects, this just recalls the object to the stack. For example, if you have stored the number 25 in a variable named X, pressing [X] [ENTER] returns the number 25 to level 1. Executing a local name always recalls the stored object *without* execution, regardless of the object type.
- *Use* RCL. *'name'* RCL returns the object stored in the (local or global) variable *name* to the stack, without executing the object. RCL is primarily used for global variables that contain programs and names, in cases where you just want to put a copy of the stored object on the stack. For data and algebraic objects, *'name'* RCL has the same effect as just executing *name*, which requires fewer steps.

The commands GET and GETI allow you to recall individual elements from arrays and lists stored in variables, without having to recall the entire object to the stack. For GET, the stack use is

*object index* GET ☞ *element*,

where *index* specifies the element to retrieve:

- For a list or a vector, the index is a real number, or a list containing one real number.
- For an array, the index is either a real number (the element number, counting in "row order"-- left to right, top to bottom) or a list of two real numbers (the element row and column).
- When the index is entered as a list, the list elements can also be names or procedures that numerically evaluate to real numbers (section 11.5.5.1).

The *object* in the above sequence can either be the list or array itself, or the name of a global or local variable in which the list or array is stored. Thus,

{ A B C } 2 GET ☞ 'B',

or

{ A B C } 'D' STO 'D' 2 GET ☞ 'B'.

GETI is designed for sequential recall of the elements in a list or array, and returns the object or its name, and the index incremented to the next element, as well as the recalled element. The general form of GETI is

*object* *index* GETI ☞ *object* *index+* *element*,

where *object* and *index* are the same as for GET, and *index+* is the same as *index* except that its value is incremented to represent the next element. Thus,

{ A B C } 2 GETI ☞ { A B C } 3 'B',

If *index* points to the last element, GETI returns either 1, { 1 }, or { 1 1 } for *index+*, as appropriate to cycle back to the first element. GETI also sets flag −64 when this occurs, or clears the flag otherwise, so that a program can easily determine when it has come to the end of a list or array.

GET can also be executed implicitly within algebraic expressions by using a function syntax--see section 11.2.

## 5.7.2 Altering the Contents of Variables

The most straightforward means of changing the contents of a variable is to store a new object into the variable using STO. However, there are a number of commands that let you modify a stored object short of replacing it entirely, without having to recall the object to the stack. (These commands apply to global and local variables, but not to port variables.)

Nine commands of this type are collected in the storage menu (⬏ MEMORY ). These are the "storage arithmetic" commands STO+, STO−, STO*, and STO/, and the specialized versions INCR and DECR, plus the single argument commands SNEG, SINV, and SCONJ. In addition to the arithmetic commands, the four array commands CON, IDN, RDM, and TRN can be applied to arrays stored in variables. PUT and PUTI, the storing counterparts of GET and GETI, allow you to alter individual elements in a stored list or array. Finally, there are several commands associated with the reserved-name variables EQ, PPAR, ΣDAT, ΣPAR, PRTPAR, and IOPAR, used by various built-in systems. We will discuss these commands in the chapters of *Part II* that describe the associated systems.

### 5.7.2.1 Store Menu Commands

Storage arithmetic is the application of +, −, *, or / to two objects, where one object is on the stack and the other stored in a variable, without having to recall the latter to the stack. For example, 25 'X' STO+ adds 25 to a number stored in X. More generally, STO+, STO−, STO*, and STO/ use a syntax similar to that of STO:

*object 'name'* STO●,

where the ● stands for any of the symbols +, −, *, or /. *Name* is a global or local name, which must refer to an existing variable. Furthermore,

*'name' object* STO●

is also allowed. Either sequence combines the *object* in level 2 with the object stored in the variable *name*, leaving the result stored in the same variable. The object and the name are dropped from the stack. Note that (unlike on the HP 28) the two objects do not have to be numerical--they can be any types that are suitable arguments for the stack ● operation. For example, if the variable A contains the string "Hello there", then the sequence

'A' ", world" STO+

replaces the contents of A with the string "Hello there, world".

As for the corresponding stack operations, the order of the storage arithmetic commands' arguments is significant. In effect, the result is the same as if you replaced the name object on the stack with the object from the named variable, then performed the stack command:

- *object 'name'* STO● computes

$$(\textit{new value}) = (\textit{stack object}) \left\{ \begin{array}{c} + \\ - \\ * \\ / \end{array} \right\} (\textit{old value}).$$

In this case, STO● is equivalent to

DUP RCL ROT SWAP ● SWAP STO.

If X has the value 1, then 3 'X' STO− stores 2 in X.

- *'name'* *object* STO● computes

$$(\textit{new value}) = (\textit{old value}) \left\{ \begin{matrix} + \\ - \\ * \\ / \end{matrix} \right\} (\textit{stack object}).$$

Here STO● is equivalent to

OVER RCL SWAP ● SWAP STO.

With 1 stored in X, 'X' 3 STO− stores −2 in X.

There is an ambiguity in this design when *both* stack arguments are name objects. In this case, the HP 48 interprets the level 1 name as the variable name; this arbitrary choice to match the sense of the arguments for STO was made as an easy-to-remember rule. Thus if you have the list { C D } stored in variable B, 'A' 'B' STO+ returns the list { A C D } to B (rather than adding or concatenating the name B to the contents of A). The rule does imply that you can not use STO+ to concatenate a name to the *end* of a list stored in a variable.

### 5.7.2.2 Counter Variables

INCR and DECR are specialized forms of STO+ and STO− that make it easy to use a global or local variable as a simple counter. INCR adds 1 to a real number stored in the variable specified by a name argument; DECR subtracts 1. Both commands return the result value to the stack, where you can compare it, for example, with some limit value. Thus *'name'* INCR is equivalent to the sequence *'name'* DUP 1 STO+ RCL, but executes about twice as fast.

### 5.7.2.3 PUT and PUTI

PUT and PUTI allow you to store individual elements into an existing array or list, using a syntax similar to that of GET and GETI (section 5.2).

For example,

{A B C} 2 'D' PUT ☞ {A D C}.

Here the target list itself is on the stack. The target can also be identified by name:

```
'MAT' { 3 3 } 25 PUTI ☞ 'MAT' { 3 4 }
```

stores the number 25 in the 3-3 element of a matrix stored in the variable MAT, and leaves the name and the incremented index (here assumed to indicate the 3-4 element) on the stack.

### 5.7.2.4 Additional Array Commands

The four storage arithmetic commands described in section 5.7.2.1 treat arrays stored in global variables differently from other object types in order to save memory. For objects other than arrays, the arithmetic is performed the same way you might do it using stack commands-- the stored object is recalled to the stack, combined with the initial stack object, then stored back in the variable. For arrays, the arithmetic is performed in place, with the result array elements replacing the stored elements as they are computed. This makes it possible to perform the array arithmetic without needing enough free memory to copy the destination array. (If you interrupt such an operation with $\boxed{\text{ATTN}}$, the array will likely be worthless, since it will contain a mixture of old and new values.)

The HP 48 provides seven additional commands that modify a stored array in place to conserve memory. For each, the stored array is represented on the stack by the name of the variable in which it is stored; the result replaces the original array.

| | |
|---|---|
| SNEG | negates the stored object. |
| SINV | computes the reciprocal of a stored number or square matrix. |
| SCONJ | computes the complex conjugate of the stored object. |
| CON | converts an arbitrary array into a constant array (all elements are the same), where the constant number is specified on the stack. |
| IDN | converts a square matrix into the identity matrix. |
| TRN | transposes and conjugates an array. |
| RDM | redimensions an array according to the dimensions specified by a list of one or two real numbers. Note that RDM can change the total size of an array if the new dimensions correspond to more or fewer elements than are in the original array. |
| PUT | replaces an element in an array (or list, see section 11.5). |
| PUTI | replaces an element in an array (or list) and returns the index of the next element. |

SNEG, SINV, and SCONJ also work with stored real or complex numbers, unit objects, global or local names, and algebraic objects, although there are no memory savings for these types. There are also no savings for any of the nine commands if the target object is stored in a local variable instead of a global.

### 5.7.3 Cataloging and Finding Variables

The VAR menu and $\boxed{\leftarrow}$ $\boxed{\text{REVIEW}}$ are convenient for manual review of the current directory. For program application there are several commands that provide information about directories and their variables. Two of these, VARS and TVARS, are used in the following program:

| FIND | *Find a Variable* | | | 460A |
|---|---|---|---|---|
| | *level 1* | | *level 2* | *level 1* |
| | *'name'* | ☞ | *'name'* | { } |
| | *'name'* | ☞ | *'name'* | { *path* } |
| | *'name'* | ☞ | *'name'* | { {*path*$_1$} ... {*path*$_n$} } |

```
«  DUP  DUP  →  name  dodir                Save the name; create local variable dodir.
  «                                        Put the directory subroutine on the stack.
    «  {}                                  Start with an empty list.
      IF  VARS  name  POS                  If the variable is in this directory,
      THEN  PATH  1  →LIST  +              then add the name to the list.
      END  15  TVARS                       Now get a list of all the subdirectories.
      IF  DUP  {}  ≠                       If there are any,
      THEN  SWAP  1  3  PICK  SIZE         then apply dodir to each.
        FOR  n  OVER  n  GET               Get the nth subdirectory.
          EVAL  dodir  EVAL                Execute dodir.
          +  UPDIR                         Add any paths found to the list.
        NEXT  SWAP  DROP                   Repeat, or discard the directory list.
      ELSE  DROP                           Discard the empty list.
      END
    »
    DUP  'dodir'  STO  EVAL                Store the subroutine in dodir, then execute it.
    IF  DUP  SIZE  1  ==                   If there's only one path,
    THEN  OBJ→  DROP                       then shed the outer list.
    END
  »
»
```

FIND locates a global variable by name anywhere in the user memory labyrinth, by searching through the current directory and all of its subdirectories for variables with that name. Given a global name as its argument, FIND returns (to level 1) a list containing the path lists for any variables of that name. (To search all of user memory, execute HOME FIND). If there is only one such variable, the result is a single path list; if there are more than one, the result is a list containing two or more path lists. An empty list indicates that the variable is not present. FIND leaves the original name argument in level 2.

FIND uses the command VARS to obtain a list of all the the variables in the current directory. The list contains the variables' names in the same order in which they appear in the VAR menu. If you execute VARS *'name'* POS, for example, you will obtain the numerical position of the variable *name* in the current directory, or zero if the variable is not present. You can also create a list of variables containing objects of a certain type or types, using TVARS (*Typed VARiables*). TVARS takes a real number or a list of real numbers, and returns a list containing the names of all of the variables in the current directory that contain objects of the types specified by the argument.

For a single variable, the command VTYPE applied to the variable's name returns the type of the object stored there, as a real number (see Table 3.1 in section 3.2 for a list of object type numbers). It is equivalent to RCL TYPE, with the exception that VTYPE returns −1 if the specified variable does not exist in the current directory.

## 5.7.4 Moving A Variable

| MOVE | *Move a Variable* | | C4D3 |
|---|---|---|---|
| | *level 2* | *level 1* | |
| | *path-name$_1$* | *path-name$_2$* | ☞ |

```
<< PATH                                 Subroutine to find a variable:
  << IF DUP TYPE 5 SAME                 Is this a path name?
    THEN DUP SIZE DUP2 GET              Then get the variable name,
      3 ROLLD 1 SWAP 1 - SUB            and the path list.
    ELSE { }                            Otherwise, use a null path list.
    END EVAL
  >> → new p s                          Save the new name, old path and subroutine.
  << s EVAL DUP RCL SWAP                Recall the object.
    IF DUP TYPE 12 ==                   If object came from a port,
    THEN SWAP NEWOB SWAP                Then free the object.
    END
    IF DUP VTYPE 15 ==                  Purge the variable:
    THEN PGDIR                          Use PGDIR for a directory.
    ELSE PURGE                          Use PURGE otherwise.
    END
    p EVAL new s EVAL STO               Store the object in the new variable.
    p EVAL                              Return to original directory.
  >>
>>
```

In section 5.1.2 we listed a simple sequence for renaming a variable, keeping the variable in the same directory. The program MOVE on the previous page generalizes that sequence so that you can move a variable from one directory to another, including to and from a port. MOVE uses two arguments, either of which can be a name (tagged for a port variable) or a path name. The path specified by the latter should be the path from the current directory to the directory containing the new or old variable. The name of the original variable should be in level 2, and the name of the new variable in level 1.

## 5.8 Calculator Resets

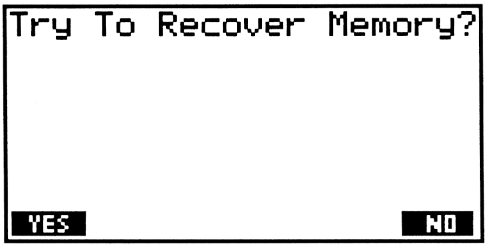
The HP48 provides a special operation, called a *memory reset*, that clears all global and port 0 variables and restores all of the calculator's default modes. Part of the memory reset is a *system halt*, that by itself resets the HP48's execution without affecting stored objects.

A system halt is obtained by pressing ATTN and the C menu key together. This operation does all of the following:

- aborts all execution;
- clears the stack, the return stack, all local memories, last arguments, the recovery stack, the command stack, and the graphics display;
- turns off user mode;
- sets the last error number to zero and the last error message (section 9.6) to an empty string;
- detaches all libraries currently attached to the home directory, and executes the configuration programs (section 5.3.2) of all libraries in the various ports;
- reestablishes the home directory as the current directory;
- activates the MTH menu;
- leaves global or port variables, alarms, and key assignments unchanged. All flags are also left unchanged, except flag −62, which is cleared (see section 7.2.2).

A system halt is performed automatically when you turn the HP48 on, if you have stored or removed any libraries from any ports since the previous time the HP48 was turned on, or if you have inserted or removed memory cards, or changed a RAM card's write-protect switch position. This ensures that there are no references (section 5.3.1.2) remaining to library objects that you may have removed.

A *memory reset*, for which you press the three keys ATTN , A , and C all together, starts by executing a system halt. Then the HP 48 displays



```
Try To Recover Memory?




YES                 NO
```

If you see this display when you turn the calculator on, or at any other time when you have not deliberately performed a memory reset, it indicates that the calculator has detected a corruption of memory contents such that it can not continue normal operation without at least a partial memory reset. This corruption can be caused by a hardware fault, including the effects of static electricity, or by the execution of SYSEVAL (section 3.10.1) with an incorrect system address.

If you choose ≣NO≣ , the HP 48 performs a complete reset, deleting all global variables, port 0 variables, key assignments, and alarms and resetting all flags to their default values. The calculator displays Memory Clear when it is ready to resume manual operation.

If you choose ≣YES≣ at the Recover RAM prompt, the HP 48 attempts to recover or restore as many user memory and port 0 variables as it can by scanning through memory for recognizable objects. If it detects a valid user memory, then it can usually restore it unchanged, except that key assignments and alarms are always lost. If it finds invalid objects, it discards them and rebuilds as much of the user memory structure as it can. In some cases when the home directory itself is corrupt, subdirectory objects there can be reconstructed, but lose their names. Then the HP 48 makes up variable names for these directories, naming them D.01, D.02, and so forth. When the automatic reconstruction process is finished, the standard display is restored. Then you can inspect the VAR menu to determine how much of user memory is intact.

During variable reconstruction, the HP 48 looks for library objects in memory to try to determine where port 0 begins. Unfortunately, if it encounters a library that was stored in a global variable, it takes that as the start of port 0, which means that some part of user memory will be discarded. For this reason, you should not keep libraries in user memory for long term storage--store them in a port instead.

# 6. Methods

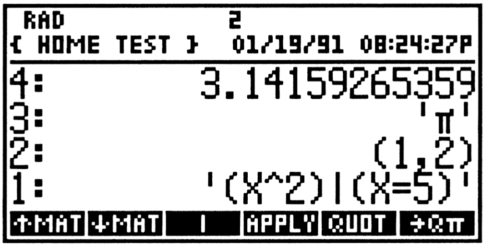
The HP 48 has such a large number of keys, commands, and methods that it is often difficult to sort out a correct approach for solving a particular problem. For example, there are many "hidden" operations available, for which no keyboard label is provided. Unless you know that the operations are there, and which keys to press, you may miss out on useful shortcuts that can save you time and effort. In this chapter, we will review the hidden operations, and some other HP 48 "methods," that can increase your confidence and comfort with the calculator before we launch into an exposition of its programming.

## 6.1 The Basic Interface

The HP 48 is fundamentally a key-per-function calculator, which means that its basic interface provides a platform for calculations on mathematical and logical objects, where each calculation in principle can be performed by means of a single keystroke. The use of an RPN stack as the focus of the interface combines facilities for the input of arguments and the output of results into a single mechanism, allowing for the endless chaining of arbitrarily complicated computations. Objects (section 3.2) are placed on the stack, commands are applied to the objects, and new objects that represent the results of the commands are returned to the stack.

*Key-per-function* means that any command can be executed by means of a single keystroke. This facility is important not only because of the mechanics of typing, but because there is a certain psychological satisfaction to making something happen with a single well-chosen keystroke. You think of a function like *sine* as one operation; the key-per-function approach makes a nice one-to-one correspondence between the abstract *sine* and the tangible keystroke. It is not an exaggeration to say that the primary goal of the HP 48's programming language, customization features, and plug-in memory ports is to allow you to extend the key-per-function interface to calculations that are not included in the built-in command set.

A central property of an RPN calculator is that the objects upon which you are operating are literally visible, as well as accessible computationally, in close proximity to the operations (i.e. the keys) that you are to apply to them. This is a succinct description of the HP 48 physical layout. In the basic HP 48 state, the display shows one or more objects on the RPN stack, within the same field of view as the keys that represent the current choice of operations. A typical display looks like this:



```
RAD            2
{ HOME TEST }  01/19/91 08:24:27P
4:        3.14159265359
3:                  'π'
2:                (1,2)
1:       '(X^2)|(X=5)'
↑MAT ↓MAT  |  APPLY QUOT →Qπ
```

The display rows numbered 1: through 4: show the first four objects on the RPN stack. Immediately below are *menu key labels* that identify the operations associated with the unlabeled keys below the labels. At the top of the display is the *status area*, that normally shows information about the status of the calculator, including the states of various modes (section 7.1), the current directory path (section 5.2), and optionally, the time and date. When you enter a new object, part or all of the stack display area is given over to a *command line* (section 6.4); when entry is complete, the display reverts to the stack.

The HP48 has a number of other display/keyboard states that are used in the course of computations, but the state described above is basic in that it is the "rest" state that is restored when all active and pending operations are complete, or when a system halt (section 5.8) resets the calculator. We shall refer to this state as the *standard environment*, which includes the *standard display* (status area, stack, menu labels) and *standard keyboard* (which may be redefined by user key assignments--see section 7.2). Another state of almost equal stature is the *plot environment*, in which the key-per-function interface is applied to graphical data instead of discrete objects. The graphical data is continuously presented to you, and the menus and keys are devoted to operations on that data, with the results immediately visible.

Because of the parallel importance of the standard environment and the plot environment, the display memory associated with each is maintained independently, so that you can switch back and forth between the two without losing data. We shall call the two display memories the *text screen* and the *graph screen*, from their respective activating commands, TEXT and GRAPH. This terminology helps focus on the logical purposes of the displays, and distinguishes them from the physical LCD and memory. We shall discuss more about these "screens" in Chapter 10.

While the plot environment is largely self-contained, the standard environment is almost indefinitely extensible. The menus and menu keys, for example, extend the basic key-per-function interface to the hundreds of built-in operations for which there are not enough keys for unique keyboard assignments. When you create programs, you are effectively adding to the built-in language (section 3.6) and providing more operations that can be applied in the same simple manner. Some programs may become complicated enough that they supplant the basic interface by redefining the keyboard and presenting special displays, in order to provide improved ease-of-use and functionality tailored to specific applications. The HP 48 itself contains several such programs, such as the EquationWriter and the MatrixWriter, the various catalogs, HP Solve, etc. The remainder of this book is essentially a description of the principles of the basic interface, and how you can develop your own extensions to that interface that span the entire spectrum from simple key-per-function operations to systems that rival the built-in environments.

## 6.2 Keyboard Mastery

The HP 48 keyboard may seem to be a complicated maze of nomenclature and colors, but there is some method in the madness. Understanding the organization of the keyboard, including the extended keyboard available through the menus, will help you remember what various keys mean, and where to find various operations.

Most personal computer keyboards are competely generic in the sense that they are not optimized for any particular software-driven application, but offer a typewriter-like "QWERTY" keyboard designed for text entry. Customization for a particular application is provided by function keys that can be labeled by keyboard overlays, or by the use of a mouse or other pointing device that makes the display into an extended keyboard. But the HP 48 is not designed to be quite so generic; rather, its keyboard is laid out with certain definite purposes in mind. The assignment of operations to the various keys reflects the priority order of these purposes:

1. *RPN calculator.* All of the sophisticated features of the HP 48 are subordinated to the requirement that the calculator must provide for convenient execution of ordinary arithmetic. Thus the number pad and the arithmetic operators are *primary* (i.e. unshifted), extra-wide keys. ENTER, of course, is given extra prominence due to its central role; the three most common stack operations, DUP ( $\boxed{\text{ENTER}}$ ), SWAP ( $\boxed{\triangleright}$ ), and DROP ( $\boxed{\Leftarrow}$ ) are available as primary keys.

2. *Scientific calculator.* The most commonly-used mathematical functions are gathered on the primary and shifted keys of the fourth row.

3. *Object entry.* The delimiters and other symbols associated with object entry are grouped on the shifted $\boxed{\div}$ , $\boxed{\times}$ , $\boxed{-}$ , and $\boxed{+}$ keys. The cursor keys are primary, and the EquationWriter and MatrixWriter, which are essentially specialized object editors, are available on shifted $\boxed{\text{ENTER}}$ .

4. *Built-in Programs.* These are what the owner's manuals call *applications*: HP Solve, automated plotting, algebra, time management, statistics, and unit management. They are activated by the shifted 7, 8, 9, 4, 5, and 6 keys. The associated menus are distinguished from ordinary menus by having special displays or key redefinitions associated with them.

5. *Customization.* The top two rows of keys are associated with customization: the menu keys (top rows), which provide access to the hundreds of operations for which there are not permanent key assignments, plus the NXT key, for navigating within the menus; VAR and CST which provide instant access to the additional operations that you define; and MTH and PRG. The latter two keys are effectively shift keys, for which the second part of each two-key combination is selected from the menu keys as labeled in the display.

6. *Text entry.* Of course, the entry of text is important for almost all of the purposes outlined above, so it may not deserve to be last in the priority list. However, we list it last to highlight the fact that the HP48 is optimized for calculator-style key-per-function operation rather than for computer-style operation via text typing. If this were not the case, the HP48 should also have primary alpha keys in a QWERTY layout.

### 6.2.1 Keystroke Strategies

Almost every operation that the HP48 can perform is available ultimately as a single key press, if you don't count shifts and menu changes. On the other hand, you can execute most (but not all) operations by typing in one or more command names with alpha characters, then pressing ENTER. You will probably want to choose an execution strategy that is intermediate between these two extremes, according to your personal skills and preferences.

- If you are good at remembering where (which menus) to find various commands, you may prefer to use menu keys for executing those commands or entering them into programs. For manual operation, it is less visually disruptive to press a menu key than to start up a command line for typing the name of a command. Also, you don't have to remember exactly how to spell each command. For programming, using a menu key to enter a command has the additional advantage that the key press also automatically enters spaces as necessary around the command name.

- If you don't like using menus, or whenever you can't find a command in a menu, you can just type the name of any command that does not appear on the keyboard or in the current menu. All of the characters used in command names are available (and labeled) on the alpha-shifted keyboard. This approach also has the advantage of leaving the current menu unchanged.

Regardless of which command execution method you prefer, you should select an alpha keyboard style. By default, [α] acts as a single-key shift, where only the next key (not counting [↰] and [↱]) is modified to produce an alpha character. To enter several consecutive alpha characters, you can hold [α] down while typing, or press [α] [α] initially to activate alpha-lock, then [α] again after typing, to turn alpha-lock off. If you frequently find yourself forgetting to press [α] twice for multi-character entries, you might consider setting flag −60, which alters the behavior of [α] so that a single press activates alpha-lock. With that choice, you must always press [α] or [ENTER] to turn off alpha-lock.

A similar style choice applies to user mode (section 7.2). Again, by default [↰] [USR] acts as a single-key modifier unless you press it twice consecutively to lock on user mode. This style is appropriate when you have a few user key definitions that you use occasionally. But if you switch back and forth to user mode frequently, you can set flag −61 so that a single press of [↰] [USR] locks user mode.

## 6.2.2 Navigating the Menus

The HP 48 menu system provides convenient, labeled access to the hundreds of HP 48 commands that don't appear directly on the keyboard. To take advantage of the convenience, however, you must understand the mechanics of moving through the menus, and also have a general idea of the menu organization--where to look for a command.

### 6.2.2.1 Mechanics

Most built-in HP 48 menus are available by pressing a two-key combination. For those that are labeled on the keyboard--PRINT, I/O, MODES, MEMORY, LIBRARY, SOLVE, PLOT, ALGEBRA, TIME, STAT, UNITS--the first key is one of the shift keys [↰] or [↱]. [MTH] and [PRG] act as shifts to activate twelve additional menus. Each actual menu contains one or more "pages" of up to six operations each. When you activate a menu by means of the menu key combination, the first page of the menu is visible in the menu labels. [NXT] pages forward through a menu, cycling back to the first page after the last page. [↰] [PREV] pages backward, wrapping to the last page from the first. You can always return to the first page by pressing [↱] [PREV] .

There are two methods to enter a menu on a page other than the first:

- [↱] [LAST MENU] returns to the menu and page that was active before the most recent use of a menu key combination. If, for example, you are viewing the third page of the parts menu ( [MTH] ≡PARTS≡ ) and press [PRG] ≡CTRL≡ [NXT] , then [↱] [LAST MENU] , you will return to back to the third page of the parts menu. Pressing [↱] [LAST MENU] again goes back to the second page of the program control menu. Note that the "intermediate" menus activated by [MTH] , [PRG] , and [↰] [UNITS] (including page changes), are not recorded for the ↱ LAST MENU operation.

- Executing MENU or TMENU with a numerical argument (section 7.3) activates the menu and page specified by the argument.

### 6.2.2.2 Content

Few people can memorize the location of every command in every menu. However, the menu titles are generally sufficiently suggestive of the menus' contents that you should be able to pick out the correct menu for a command, then page through it with NXT until you see the command label. The process is complicated somewhat because some menus "share" a keyboard label--for example, ↰ MODES and ↱ MODES activate separate menus. This label sharing is used because the HP48 designers felt that one spelled-out label for related menus is preferable to two cryptic, abbreviated labels. A rough guide for distinguishing two menus sharing a label is that the ↰ menu contains operations and commands more commonly used in manual operation, whereas the ↱ menu contains commands that frequently appear in programs. Thus the ↰ MODES menu contains mode-setting keys that make it easy for you to select various modes without having to remember flag numbers. To achieve the same effect in a program, you must use the flag commands present in the ↱ MODES menu.

Table 6.1 lists the various built-in HP48 menus and their general content. The table is not intended for memorization, but if you read through it once or twice you should get a general picture of where to look for various operations.

**Table 6.1. HP48 Menus**

| Number | Name | Keys | Content |
|---|---|---|---|
| 1 | Custom | CST | User-defined custom menu |
| 2 | Variables | VAR | Global variables |
| 3 | Math | MTH | Intermediate menu for math menus selection |
| 4 | Parts | MTH ≡PARTS≡ | Commands for extracting "parts" of real and complex numbers--mantissas, exponents, integer parts, percentages, etc. |
| 5 | Probability | MTH ≡PROB≡ | Permutations, factorial, random numbers |
| 6 | Hyperbolics | MTH ≡HYP≡ | Hyperbolic functions, $e^x - 1$, $\ln(x+1)$ |
| 7 | Matrix | MTH ≡MATR≡ | Matrix computations |
| 8 | Vector | MTH ≡VECT≡ | Coordinate system selection, dot and cross products, degree/radian conversions |

| | | | |
|---|---|---|---|
| 9 | Base Operations | MTH ≡BASE≡ | Binary integer operations, bit manipulations, logical operations |
| 10 | Program | PRG | Intermediate menu for program menus selection |
| 11 | Stack Manipulation | PRG ≡STK≡ | Copying, moving, deleting stack objects |
| 12 | Object Composition | PRG ≡OBJ≡ | Assembling and disassembling objects |
| 13 | Display | PRG ≡DISP≡ | Controlling the display and working with graphics objects |
| 14 | Program Control | PRG ≡CTRL≡ | Program debugging, starting and stopping |
| 15 | Program Branch | PRG ≡BRCH≡ | Program branch structures |
| 16 | Logical Tests | PRG ≡TEST≡ | Logical operations and object comparisons |
| 17 | Printer | ◁ PRINT | Printer output and control |
| 18 | Input/Output | ◁ I/O | Kermit transfers and serial operations |
| 19 | I/O Setup | ◁ I/O ≡SETUP≡ | Serial handshaking configuration |
| 20 | Mode Switches | ◁ MODES | Toggle keys for mode selections |
| 21 | Customization | ▷ MODES | Key assignments, flags, custom menus |
| 22 | Memory Management | ◁ MEMORY | Management of global and port variables |
| 23 | Storage Arithmetic | ▷ MEMORY | "In-place" operations on stored objects |
| 24 | Library | ◁ LIBRARY | Access to libraries and port variables |
| 25 | Port 0 | ◁ LIBRARY ≡PORT≡ | Port 0 objects |
| 26 | Port 1 | ◁ LIBRARY ≡PORT≡ | Port 1 objects |
| 27 | Port 2 | ◁ LIBRARY ≡PORT≡ | Port 2 objects |
| 28 | Edit | ◁ EDIT | Extended editing operations |
| 29 | HP Solve Equation Entry | ◁ SOLV | Entry, editing, and selection of HP Solve equations |
| 30 | HP Solve Variables | ◁ SOLV ≡SOLVR≡ or ▷ SOLV | Interactive HP Solve |
| 31 | Plot Equation Entry | ◁ PLOT | Entry, editing, and selection of plot formulae |
| 32 | Plot Type Selection | ◁ PLOT ≡PTYPE≡ | Selection of automatic plot type |
| 33 | Plot Execution | ▷ PLOT | Execution of plots, plot screen configuration |
| 34 | Algebra | ◁ ALGEBRA | Solving and manipulating symbolic expressions, →Qπ |
| 35 | Time Management | ◁ TIME | Time and date operations |
| 36 | Clock Adjustment | ◁ TIME ≡ADJ≡ | Correcting the current time |
| 37 | Alarm Setting | TIME ≡ALRM≡ | Setting appointment and control alarms |
| 38 | Alarm Repeat | TIME ≡ALRM≡ ≡RPT≡ | Choosing a repeat interval |
| 39 | Time Set | TIME ≡SET≡ | Setting the time and date |

| | | | |
|---|---|---|---|
| 40 | Statistics | ⬑ STAT | Statistics computations, plotting, distributions, regressions |
| 41 | Regression Model | STAT MODL | Selecting the curve fit model |
| 42 | Unit Conversions | ⬑ UNITS | Intermediate menu for unit categories |
| 43-58 | Unit Categories | ⬑ UNITS LENG etc. | Interactive unit entry and conversions menus |
| 59 | Unit Objects | ⮣ UNITS | Creating, converting, and disassembling unit objects |

### 6.2.2.3 Exiting

The HP48 menu system is defined without a "home menu"--there is no master menu to which you return when you are finished with the current menu. Moreover, there is always a menu present, except in the plot environment's and the EquationWriter's graphics scrolling modes. Thus, in general you don't "exit" from any menu, you just select another menu. There are two kinds of exceptions to this general rule:

- "One-shot" menus. For most menus, you are at least as likely to select two consecutive operations from a menu as you are to select one then return to the previous menu. There are are several menus, however, that are designed for a single choice followed by an automatic return to the previous menu. These menus are the plot type menu ( PTYPE ) the regression model menu ( MODL ), the alarm repeat menu ( RPT ), and the zoom menu ( ZOOM ) in the plot environment. The latter two menus also include provide for returning to the previous menu when you don't use any of the current menu's operations ( NONE for the repeat menu, and EXIT for the zoom menu). In the plot type and regression model menus, you can make a similar exit by pressing the menu key corresponding to the current type, or by going directly to another menu.

  The various RULES operations menus also belong to this category, although executing a RULES operation does not return to the top-level menu containing RULES but instead activates the transformations menu appropriate for the newly transformed subexpression (which may be the same menu). To exit from one of these menus you press a cursor key to select a new subexpression and return to the main RULES menu.

- The main RULES menu in the EquationWriter and the function ( FCN ) menu in the plot environment menu permit multiple operations, but are sub-menus of other menus that are not directly accessible via labeled keys. The sub-menus therefore include an EXIT key that returns to the parent menu.

In this section we have been speaking of *menus* as distinct from *environments* such as the plot catalog (⬑ PLOT CAT ), that provide their own special menus as part of their execution. These environments in general supplant the standard environment with a redefined keyboard and their own special displays. ≡CAT≡ includes a menu as part of its keyboard redefinition, but notice that the CAT label does not include the little tab that indicates that the key is a menu selection key. The distinction is useful because whereas you don't need to exit a menu except as outlined above, you must terminate an environment by pressing [ATTN], which returns to the standard environment.

### 6.2.3 ATTN

The use of [ATTN] (*ATTentioN!*) to terminate environments like the EquationWriter or a catalog is one example of the use of ATTN . The basic purpose of ATTN is to provide an exit path from ongoing operations back ultimately to the standard environment. This sometimes can be a multi-step process--for example, when you are using the interactive stack from within the MatrixWriter, (section 6.6), a first [ATTN] exits from the interactive stack and returns to the MatrixWriter display, a second clears the command line, and a third terminates the MatrixWriter and returns to the standard environment.

ATTN also serves to halt program execution, including the programs you create and built-in commands like ∫ or COLCT that may take a significant amount of time. It is a "gentle" form of interruption, in that the stack is not cleared and no stored objects are affected (except for the local variables associated with currently executing programs). In general, you can use [ATTN] as the all-purpose "quit this and start fresh" operation. If you using a catalog or a similar environment where the standard keyboard is unavailable, and there is no menu key provided for exiting, pressing [ATTN] will get you back to the standard environment.

See also section 9.6.1 for more information about ATTN's behavior as an error condition.

## 6.3 Hidden Operations

Unless you have perused the HP 48 owner's manuals very carefully, you may not realize, for example, that pressing [⮣] ≡CENT≡ (in the [⮣] [PLOT] menu) returns the logical coordinates of the current graphics screen. This is an example of a *hidden operation*--a useful manual operation that is not explicitly labeled on the keyboard or on a menu label. There are dozens of such operations, which you probably encountered as you read through the owner's manuals, but many of which you may have forgotten because of their number and invisibility. In this section, we will review all such operations, and present the various conceptual models that can help you remember what and where the operations are. We have already discussed unlabeled menus (section 6.2.2.2); here we will focus on operations other than menu selection.

In almost all cases, the hidden operations can be executed by a non-hidden method. The hidden operations are provided only as keystroke shortcuts that are useful when you can remember them, but not vital. There are some exceptions, such as the use of [△] to activate the interactive stack (section 4.5), which is not explicitly labeled on the keyboard. However, here and in the other cases the model (moving up the stack with [△] ) is so clear that you should not have any trouble remembering it.

### 6.3.1 No-Command-Line Key Actions

Several primary key definitions are associated with the command line, and therefore have no meaning when the command line is not present. An obvious example of this is [ENTER] , the primary definition of which is to execute the command line. When the command line is absent, pressing [ENTER] executes DUP, copying the level 1 object into level 2. The association between ENTER and DUP goes back to the [ENTER↑] key on previous HP calculators such as the HP41, on which that key both terminates digit entry and copies the x-register (level 1) into the y-register (level 2). When you enter a new object on the HP48, the first press of [ENTER] creates a first version of the object, and subsequent presses create additional copies, so the association is natural even without the historical derivation.

The remaining no-command-line key actions may have less obvious relationships with the key labels, but they are useful nonetheless:

- [⇦] executes DROP.
- [▷] executes SWAP. With the assignments of DUP and DROP, this means that the three most common stack operations are available as primary keys. The remaining operations are provided in the interactive stack.
- [◁] executes GRAPH, providing single-key access to the graph screen (section 10.0).
- [△] activates the interactive stack.
- [▽] activates the command line, MatrixWriter, or EquationWriter, as appropriate to view the level 1 object.

When the command line is present, you can enter the commands DUP, DROP, SWAP, and GRAPH in the usual way by pressing the [◁] key first (for DUP, [▷] [ENTER] works as well).

### 6.3.2 Shift Key Meanings

There are number of hidden shifted-key operations that are derived from two models of shift-key meanings. The first model associates the shift keys with "extended" actions of the primary keys; the second relates the shifts to store and recall operations.

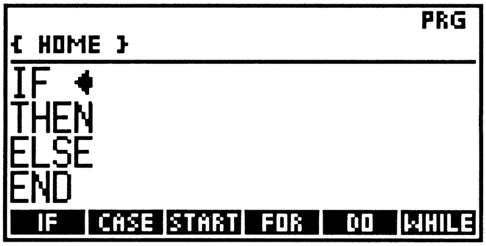
### 6.3.2.1 Extended Actions

An obvious example of the action of a shifted key as an extension of the primary key action is provided by [▷] and [↱] [▷]. In the command line, [▷] moves the cursor one character to the right; [↱] [▷] moves it "all the way" to the end of the current line. The cursor keys behave similarly on the graphics screen: [▷] moves right one pixel, whereas [↱] [▷] jumps the cursor to the right edge of the display (if the cursor is already there, [↱] [▷] moves the display window to the right to bring the far right edge of the graphics screen into view at the right edge of the display).

The "all the way" actions are most frequently associated with the [↱] key, but there are some cases where [↰] is also used for intermediate extensions of the primary key. For example, in the alarm catalog ( [↰] [TIME] CAT ), [▽] moves the catalog pointer down one line, and [↱] [▽] moves the pointer to the bottom of the catalog, as you might expect from the [↱] model. [↰] [▽] moves the pointer down by one "page" of five entries, which is intermediate between the [▽] and [↱] [▽] actions.

This model of shifted actions as extensions is not limited to cursor motions. Consider [⇐] for "erasing:" [⇐] erases part of an object (in the command line); [↰] [DROP] erases a complete object; and [↱] [CLR] (CLEAR) erases the entire stack of objects.

Another example is provided by the shift extensions to the menu keys in the program branch menu ( [PRG] BRCH ). The primary menu keys enter the individual program structure words (section 9.2) shown on the key labels, but wherever appropriate, the shifted keys enter structure word combinations or entire structures. The IF key is the most extreme case: pressing [↱] IF enters four lines simultaneously:



```
                              PRG
{ HOME }
IF ◆
THEN
ELSE
END
 IF  CASE START FOR  DO  WHILE
```

This is a great help in entering an IF structure (section 9.4.1). The cursor is initially positioned at the start of the *if-sequence*; after entering it, press [▽] (unless you have entered a newline), and the cursor then is positioned for entry of the *then-sequence*, and so on through the remainder of the structure. Not only do you not have to type the other structure words, but you can't forget to enter them. Notice also that [↰] IF enters the IF structure words minus the ELSE, so that the [↰]/[↱] model of intermediate/extreme case is maintained.

Tables 6.2 and 6.3 list all of the hidden shift key actions that follow the primary key extension model.

**Table 6.2. Shift-key Extended Actions**

| **Key** | **Primary** | [↰] | [↱] |
|---|---|---|---|
| [NXT] | Next menu page | | First menu page |
| Command line [◁][▷][△] and [▽] | Move cursor one character or line | | Move far left, right, up or down. |
| Interactive stack [△] or [▽] | Move stack pointer up or down one level | Move up or down four levels | Move to top or bottom of stack |
| Graphics screen [◁][▷][△] and [▽] | Move cursor or window one pixel | | Move to edge of display or edge of graphics screen |
| EquationWriter entry [▷] | Close subexpression | | Close all subexpressions |
| EquationWriter subexpression environment [◁][▷][△] and [▽] | Move subexpression highlight by one object | | Move to "edge" of expression |
| MatrixWriter [◁][▷][△] and [▽] | Move cell cursor in indicated direction by one row or column | | Move to first or last row or column |
| EDIT menu ←DEL and DEL→ | Delete previous or next "word" | | Delete to beginning or end of line |
| [DEL] | Delete a character (command line) | Purge a variable | Clear the current directory (CLVAR) |
| [⇦] | Backspace (command line) | DROP | CLEAR |
| Alarm, plot, solve, and statistics catalogs [△] and [▽] | Move catalog pointer up or down one entry | Move pointer up or down five entries | Move pointer to top or bottom of catalog |

**Table 6.3. Program Branch Menu Shifted Key Extensions**

| Key | Primary | ↰ | ↱ |
|---|---|---|---|
| ≡ IF ≡ | IF | IF<br>THEN<br>END | IF<br>THEN<br>ELSE<br>END |
| ≡CASE≡ | CASE | CASE<br>THEN<br>END<br>END | THEN<br>END |
| ≡START≡ | START | START<br>NEXT | START<br>STEP |
| ≡FOR≡ | FOR | FOR<br>NEXT | FOR<br>STEP |
| ≡DO≡ | DO | DO<br>UNTIL<br>END | |
| ≡WHILE≡ | WHILE | WHILE<br>REPEAT<br>END | |
| ≡ IFERR ≡ | IFERR | IFERR<br>THEN<br>END | IFERR<br>THEN<br>ELSE<br>END |

### 6.3.2.2 Store and Recall Actions

When applied to a menu key associated with a name object, [↰] has the meaning "store into this variable." Similarly, [↱] means "recall the contents of this variable." The primary key itself executes the name object. This is the default action for custom menu keys (section 7.3) defined with global, local, and port names, and is extended to the VAR (section 5.1) and the LIBRARY menu (section 5.3.1.1). It is also applicable to ≡CST≡ in the customization menu ([↱] [MODES] ). As a mnemonic to help you remember which shift key is which in this context, note that [↱] [RCL] is right-shifted, and [↰] [DEF] (which is a form of store) is left-shifted.

There are several instances of menu keys that are associated with commands rather than names, but for which the ⮣ key retains its recall action. The ≡CENT≡ key cited earlier is an example. Its primary definition is the command CENTR, which redefines the plot parameters in the variable PPAR so that the new center of the graph screen corresponds to its complex number argument. Although there is no center coordinates variable as such, ⮣ ≡CENTR≡ computes those coordinates and returns them to the stack as if recalling them from an ordinary variable. This is handy when you want to recenter the graph screen to a position based on the current position.

Table 6.4 (on the next page) lists all of the ⮣/RCL extensions available in the various HP 48 menus.

The menu keys in the HP Solve variables menu (⮣ SOLVE or ⮢ SOLVE ≡SOLVR≡ ) exhibit a deliberate reversal of the normal primary and left-shifted actions of menu keys defined by names. In this menu, the primary key action is to *store* the level 1 object in the labeled variable; the left-shifted action is to *solve* for the variable, which you can view as a form of evaluation (especially since the solved value ends up in the variable). This reversal of roles is motivated by aligning the HP 48's HP Solve interface as closely as possible with that on other HP calculators. The use of dark letters against a light background for the menu labels, reversed from other menus, identifies the HP Solve variables menu, and reminds you of the reverse key actions. (The right-shifted menu keys maintain the usual RCL definition.)

#### 6.3.2.3 Other Hidden Shift Actions

There remain a number of hidden operations that don't conform to any particular model, but which are reasonably obvious extensions of primary key actions. As with other hidden operations, these are usually duplicates of operations that are available by other means.

- ⮣ PRINT executes PR1, which prints the object in level 1.
- ⮣ I/O activates Kermit server mode.
- Pressing ON - MTH together prints an image of the current display.
- ⮣ SOLVE activates the HP Solve variables menu.
- ⮣ ALGEBRA activates the plot/solve equation catalog.
- ⮣ TIME activates the alarm catalog.
- ⮣ PLOT ⮢ ≡DRAW≡ executes STEQ. This matches ⮣ ≡DRAW≡ , which executes RCEQ. The existence of these two hidden operations allows you to do simple one-time plotting (i.e. where you don't bother to store the plot equation in its own variable) from a single menu.

- [◁] [STAT] [◁] ≡Σ+≡ enters Σ−. This is the only key for Σ−, so this particular hidden key is definitely worth remembering.

**Table 6.4. Hidden RCL Operations**

| Menu Key | [▷] Action |
|---|---|
| [▷] [MODES] *menu* | |
| ≡CST≡ | Return contents of CST. |
| [◁] [SOLVE] *menu* | |
| ≡STEQ≡ | RCEQ |
| [◁] [PLOT] *menu* | |
| ≡STEQ≡ | RCEQ |
| [▷] [PLOT] *menu* | |
| ≡DRAW≡ | RCEQ |
| ≡XRNG≡ | Return $x_{min}$ and $x_{max}$ |
| ≡YRNG≡ | Return $y_{min}$ and $y_{max}$ |
| ≡INDEP≡ | Return independent variable name or list from PPAR |
| ≡DEPN≡ | Return dependent variable name or list from PPAR |
| ≡RES≡ | Return plot resolution parameter from PPAR |
| ≡CENTR≡ | Return coordinates of center pixel in graphics screen |
| ≡SCALE≡ | Return *x*- and *y*-scale values from graphics screen |
| ≡AXES≡ | Return axes coordinates from PPAR |
| ≡PDIM≡ | Return current dimensions of graphics screen |
| [◁] [TIME] ≡ALRM≡ *menu* | |
| ≡EXEC≡ | Recall current alarm execution object |
| [◁] [STAT] *menu* | |
| ≡STOΣ≡ | RCLΣ |
| ≡XCOL≡ | Recall current independent variable column |
| ≡YCOL≡ | Recall current dependent variable column |

## 6.4 Object Entry

The central focus of the HP48 is *objects*, the elements of data or procedure that you (or the calculator) enter as representations of the calculations you are making. We discussed the theory and meanings of the various object types in Chapter 3; here we will look more closely at how you create new objects.

The basic mechanism for the manual entry of objects is the *command line*. The command line derives its name from the fact that you can enter a "line" of commands--a series of calculator instructions that are executed all together when you press ENTER . A better term might be *command editor*, since it is not restricted to a single line, or better yet, *object editor*, since commands are only one of the many kinds of objects you can create there. In any case, you create objects by typing text representing the objects into the command line. You terminate a particular editing session by pressing ENTER or any of the other keys that perform an implicit ENTER. At that point the HP 48 converts the command line text into the objects you specify.

The double-width ENTER↑ key has always been the trademark of HP RPN calculators that sets them apart from so-called "algebraic" calculators with their prominent = key. In all RPN calculators including the HP48, the fundamental purpose of ENTER is to terminate object entry. In pre-HP 28 calculators, the only objects that can be entered are real numbers, so that terminating entry just means turning off digit entry mode and leaving the completed number in the X-register. In the HP 48, ENTER retains the basic action of terminating entry and entering new objects. However, because the HP 48 replaces ordinary calculator digit entry with a *command line* that can contain any number of objects and commands, ENTER can invoke almost any of the calculator's capabilities as well as just entering numbers onto the stack.

The fundamental definition of the HP 48 operation ENTER is:

> *Take the text in the command line, check it for correct object syntax, then treat it as a program and execute the objects defined there.*

This is a much-elaborated version of the old "terminate-digit-entry and enter a number onto the stack," but in simple cases, it amounts to the same thing. If you press a series of digit keys, then ENTER , you will end up with a number in level 1. The same key sequence on an HP 41 or a similar calculator yields the same result. For the sake of keystroke efficiency and to preserve additional consistency with other RPN calculators, many HP 48 keys besides ENTER also execute ENTER as well as their own specific definitions. This feature is called *implicit* ENTER, to distinguish it from *explicit* ENTER, which is the direct use of ENTER .

An example of the use of implicit ENTER is the sequence 1 ENTER 2 + . This adds the 1 and the 2, just as it always has in HP RPN calculators. At the time you press + , the 2 is still in the command line; the implicit ENTER performed by + puts the 2 on the stack before the addition is performed.

## 6.4.1 Key Definitions and Entry Modes

A *key definition* is the object assigned to the key, i.e. the object that is used when the key is pressed. We say the object is "used" rather than "executed", because the object may or may not be executed, depending on the object type and the current entry mode. Any key does one of two things when you press it:

- The key acts as a *typing key*, merely adding one or more characters to the command line. In this case, for example, it might be the *name* of the key definition object that is used rather than the object itself.
- The key acts as an *immediate-execute* key, causing any other kind of action.

With this distinction, we can sort HP48 keys (including menu keys) into three types:

1. Keys that are always typing keys. These include the alpha-keyboard character keys, the digit keys, and the delimiter keys, plus the program structure word menu keys in the program branch menu ( PRG ≡BRCH≡ ).
2. Keys that are always immediate-execute keys. These are keys that never add characters to the command line. Examples are ENTER and menu selection keys such as PRG or MTH .
3. Keys that may act either as immediate-execute keys or typing keys, according to the current entry mode. These *mode-dependent* keys are the most common key type in the HP48; nearly all are command keys.

(Here we are speaking only of the standard key definitions, i.e. the keyboard that is available when user mode is off. The key definition object of any key be changed--see section 7.2--but the ideas presented here are common to the user and standard keyboards.)

The mode-dependent keys are so-called because they are sensitive to the four entry modes entry modes that determine their behavior. The entry modes are as follows:

- *Immediate mode.* All mode-dependent keys act as immediate-execute keys. This is the default mode, to which the HP48 normally returns after ENTER.
- *Algebraic mode* (ALG annunciator in the status area). Mode-dependent keys with definitions that are functions permitted in algebraic expressions, such as SIN, +, or LOG, act as typing keys. Parentheses are automatically added after the function names if appropriate. Other mode-dependent keys act as immediate-execute keys.
- *Program mode* (PRG annunciator). All mode-dependent keys corresponding to programmable commands act as typing keys. Spaces are automatically added around the command names to separate them from previous command line entries. There are a few mode-dependent keys that have no command line text associated with them, such as ≡SST≡ , or programs used as user key definitions; these keys just beep when pressed in program mode.

- *Algebraic/program mode* (ALG PRG annunciator). Same as program mode, except that the names of functions are not surrounded by spaces, and parentheses are added where appropriate.

Whether or not a key performs ENTER depends on more than just the current entry mode. It is true that only immediate-execute keys *may* do ENTER; there are no cases where a key acting as a typing key adds characters to the command line, and then also does ENTER. Furthermore, the great majority of immediate-execute *command* keys *do* perform ENTER. For example, all keys for commands that use stack arguments do an implicit ENTER to insure that the command is applied to the most recently entered arguments, including those that are still pending in the command line. This saves you the extra [ENTER] keystroke that you would otherwise need.

The command keys that *do not* perform ENTER regardless of the entry mode are menu keys for the commands that control calculator numerical modes, and which require no arguments: ≡STD≡ , ≡DEG≡ , and ≡RAD≡ in the MODE menu, and ≡DEC≡ , ≡HEX≡ , ≡OCT≡ and ≡BIN≡ in the BINARY menu. Because the modes can affect the interpretation of command line numbers, these exceptions to the general implicit ENTER rule are provided to allow you to change the modes *after* you have started a command line.

Generally, an immediate-execute non-programmable operation key performs ENTER if the operation uses stack objects. For example, [EDIT] and [⮣] [VISIT] use stack arguments, and therefore do implicit ENTER before performing their respective operations. [⮣] [POLAR] is an example of an key that does not affect the stack, and can therefore execute without ENTER, even in program mode. There are some exceptions to the general rule; [⮢] [LAST STACK] obviously changes the stack, but the operation executes without ENTER and without changing the command line. You can also modify the stack from within the command line editor using ≡↑STK≡ in the edit menu, but this also does not require an ENTER.

Most command keys are mode-dependent, but there are a few that always act as typing keys. These are the program structure words (section 9.2) found in the program branch menu, plus HALT and PROMPT (section 12.2). CLVAR ( [⮣] [PURGE] ) is also defined as a typing-only key, because its effects are so drastic. By entering the command name into the command line rather than executing CLVAR immediately, you get a chance to double-check that it is what you really want to do.

## 6.4.2 Controlling the Entry Mode

The preceding key-behavior rules may appear elaborate, but in actual use they are generally not difficult to master (in fact, you seldom need to think about them at all). This is due in large part to the fact that the HP 48 *automatically* changes its entry mode to match the objects that you enter. Also, you can manually change the entry mode for those cases when the HP 48's automatic choice is not what you want.

1. The default mode following an ENTER is immediate entry mode. This choice is derived from the traditional behavior of RPN calculators, where pressing a function key causes immediate execution of the function. When you type digits or letters to start a new command line, the HP 48 remains in immediate entry mode.
2. The HP 48 automatically changes to algebraic entry mode when you press [ ' ] to start entry of a quoted name or an algebraic object. The ALG annunciator appears.
3. If you press [◁] [« »] or [◁] [{ }] to start entry of a program or list, the HP 48 automatically switches to program entry mode, indicated by the PRG annunciator.
4. While the HP 48 is in program entry mode, pressing [ ' ] activates algebraic/program mode, turning on both the ALG and PRG annunciators. This is intended to aid entering algebraic objects within programs and lists. Pressing a key corresponding to an object or delimiter that is not allowed in algebraic expressions restores ordinary program mode and turns off the ALG annunciator.

This progression works reasonably well to spare you from having to control the entry mode yourself, especially if you are entering one object at a time. However, there are some circumstances in which you may need to override the automatic entry mode selection. To accumulate a series of commands into the command line without creating a program object, you must turn program mode on to prevent the commands from executing. Or, to enter a function into a program following a quoted name or an algebraic object (e.g. 'X' SIN), you must turn off algebraic/program entry mode to prevent the HP 48 from adding parentheses to the function name. These mode changes are made with [▷] [ENTRY]:

- In immediate entry mode, pressing [▷] [ENTRY] turns on program mode.
- In program entry mode, [▷] [ENTRY] turns on algebraic/program mode.
- In algebraic/program mode, [▷] [ENTRY] restores program mode (turns ALG off).

Note that once you have selected program mode, you can't return to immediate mode while the current command line is still active.

### 6.4.3 ENTER in Detail

Now that we've established at some length which keys perform ENTER, and under what circumstances, we can return to the precise definition of ENTER. The following are the actions that take place at every explicit or implicit ENTER. (The normal ENTER sequence described here can be redefined; see section 7.4).

1. A copy of the current stack is saved. It is important to note that the stack save is performed *before* the command line is processed. If the ENTER is caused by an immediate-execute operation key, the stack save also *precedes* execution of the operation. This means that although breaking up a series of commands with ENTER (either explicit or implicit) gives the same computed results as executing all of the commands at once in a single command line, the results of pressing [◁] [LAST STACK] at the ends of the series are different. For example, each of the following keystroke sequences adds 1+2 and returns 3 to the stack. However, [◁] [LAST STACK] gives a different result in each case (assume an empty stack to start with):

| **Keystrokes:** | **Stack after** LAST STACK: |
|---|---|
| [1] [ENTER] [2] [ENTER] [+] | 2: 1<br>1: 2 |
| [1] [ENTER] [2] [+] | 1: 1 |
| [1] [SPACE] [2] [+] | *(empty)* |

2. The command line text is *parsed*--converted from text into a series of objects. (This step can be bypassed or postponed by using *vectored* ENTER--see section 7.4). First, the text is broken into *object strings*, individual portions of the command line text that will become objects. The object strings are defined by *delimiters* and *separators*:

   - A *delimiter* is one of the symbols (, ), ', ", [, ], {, }, <<, >>, _, #, GROB, C$, and DIR, that identify the different object types. The comment character @ can also be considered as a delimiter, even though it doesn't identify an object.
   - A *separator* is either a space, a newline, a semicolon, or whichever of "." or "," is not the current fraction mark. Separators are used to separate real numbers, commands, and names, which have no special delimiters, from other objects, and are generally used to make the command line more legible. Unlike delimiters, separators can be repeated--extra ones are ignored.

   For example, the command line

   ```
   12345.789  'FRED'  "123"  <<  DROP  'SAM'  STO  >>  PETE
   ```

is broken into the object strings

```
            12345.789
             'FRED'
              "123"
<< DROP     'SAM'     STO >>
              PETE
```

The process is repeated as necessary within algebraic objects, programs and lists, which contain other objects. In the above example, the program object is further broken into the object strings DROP, 'SAM', and STO.

3. Each object string is checked against the syntax rules appropriate for its object type. As each object string passes its tests, an object is created from the string and pushed onto the stack. (This step is invisible--you won't see a stack display again until all of the new objects have been executed.) If any object string is found to violate a syntax rule, all of the newly created objects are dropped from the stack, and the command line is reactivated, with the cursor placed at the position in the command line where the error was encountered.
4. When the command line has successfully been converted into stack objects, a copy of the original text string is saved in the command stack (unless it has been disabled). Normally, this only happens if there are no syntax errors. However, if the HP48 runs out of memory while it is creating the command line objects, the command line is saved, giving you a chance to try again after you have cleared some additional memory. If the command stack is disabled, the command line text is never saved.
5. The new stack objects are combined into a program, which is then executed.
6. If the ENTER was implicit, the operation associated with the key that started the ENTER is executed.
7. If vectored ENTER (section 7.4) is in effect, the post-entry object (βENTER) is executed.
8. When the command line program plus the implicit ENTER key operation are finished, the HP48 checks to see if there have been any keys pressed since the ENTER. If there have, the "busy" annunciator remains off, and those keys are processed.
9. Finally, when all execution is complete, and no unprocessed keys remain, the stack is displayed (unless some special display supersedes the normal stack display) and the busy annunciator is turned off. Since the stack display can take an appreciable amount of time, the display is postponed when keys are pending, to speed up the overall process.

There are several advantages of using command lines instead of immediate-execute command keys:

- You can repeat a sequence of commands without having to make the sequence into a program. Each time you execute the sequence, you can recover the command line with ◁ LAST CMD, then press ENTER to execute it again. You can also modify the sequence each time you execute it.
- If you get an unexpected result, you can press ◁ LAST STACK to recover the stack, then ◁ LAST CMD to reexamine what you did. (Since ◁ LAST CMD itself does an implicit ENTER, you should press ◁ LAST STACK *before* ◁ LAST CMD to recover the original stack.)
- It's the fastest way to execute the command sequence from the keyboard, since you don't have to wait for the stack display after each object is executed.
- Because the command line is a program, you can do anything within the command line that you can in a program--create local variables, use program branch structures, HALT, single-step, set error traps, etc.

You can also turn this picture around and imagine a program as a command line for which execution is postponed. You can take any command line, surround it with ≪ ≫, and obtain a program that enters level 1 unexecuted when you press ENTER.

#### 6.4.3.1 Comments

The @ character is a special delimiter that allows you to embed *comments* within command lines. A comment is text that is not converted into any object; it is discarded when the command line is entered. This is obviously of little use when you are creating objects directly on the HP 48; however, it is very useful when you are creating or editing objects using a word processor on a personal computer. In that case, comments can be very helpful in documenting a program for later review, or in keeping track of stack objects as you are writing a program. When you transfer the program to the HP 48, the comments are automatically removed by the calculator.

Any text between two @ symbols in the same line is treated as a comment. This allows you to insert a comment at any point between objects--in fact, at any point where a space is allowed, such as between the elements of a vector (within string and name objects, the @ is treated as an ordinary character). If there is only one @ in a line, then all of the line to the right of the @ becomes a comment. The latter form is most common in programs, where you might include comments at the end of most program lines.

## 6.5 Object Editing and Viewing

Editing an object is the process of recreating a text representation of the object, changing the text, then constructing a new version of the object from the altered text. For most types of objects, this is achieved by recalling the object to the command line using EDIT or VISIT:

- ◁ EDIT copies the object in level 1 to the command line in text form, automatically activating program entry mode. There you can make any desired changes, then press ENTER to replace the original object with the modified version (more precisely, ENTER drops the original object, and executes the command line). If you press ATTN instead, the command line is abandoned and the level 1 object is left intact.

- ▷ VISIT is an indirect form of EDIT, where the object to be edited is specified in level 1 rather than being present in level 1 itself. If level 1 contains a real number, that indicates the stack level of the target object, e.g. 3 ▷ VISIT edits the object in level 3. VISIT's argument may also be a global name, in which case the object stored in the corresponding global variable is copied to the command line. In either case, pressing ENTER replaces the original object with the edited version. More accurately, when the argument is a global name, ▷ VISIT replaces the stored object with whatever object ends up in level 1 after the command line is executed. In the case of a numerical argument, VISIT rolls the object in the specified level to level 1 and then applies EDIT. ENTER drops the level 1 argument, executes the command line, and rolls the resulting level 1 object back to the original VISIT level.

  As for EDIT, ATTN cancels VISIT, discarding the command line (and the level 1 argument), and leaving the original object unchanged.

Both EDIT and VISIT automatically activate the *edit menu*, which contains additional editing operations. You can also activate this menu when you create a command line to enter new objects, or restore the menu after switching to another menu, by pressing ◁ EDIT whenever the command line is present. The menu contains the following operations:

- SKIP→ and ←SKIP move the cursor forward and backward by several characters at at time, so that you can you move the cursor quickly to a point where you wish to make changes. Each skip moves to the next or previous non-space character that follows a space or a newline.

- DEL→ and ←DEL "move" in the same manner as SKIP→ and ←SKIP , except that they delete the characters between the original cursor position and the destination (so the cursor doesn't move on the display). DEL→ deletes to the right, from the current character up to (but not including) the destination character. ←DEL deletes to the left, from the character to the left of the cursor through the destination character to the left.

- ⟨↱⟩ DEL→ is an extension of DEL→ , deleting all characters from the cursor position through the end of the current display line. Similarly, ⟨↱⟩ ←DEL deletes all characters preceding the cursor position on the current line.
- INS turns *insert mode* off (and back on, if you press it again), so that subsequent typing overwrites the characters under the cursor instead of inserting new characters. This is useful when you are replacing a sequence of command line text with another of comparable size. The state of insert mode is preserved during the current edit session until you deliberately change it, but insert mode is always restored after ENTER .
- ↑STK activates a restricted form of the *interactive stack* (section 4.5), where the menu contains the single key ECHO . This key *echoes*, i.e. copies, the object selected by the stack pointer to the command line at the cursor location. After echoing any number of stack objects, press ENTER or ATTN to return to normal command line entry.

  ECHO is particularly useful when you want to include in a program or a list an object that you have previously entered or computed, without having to retype the object. It also provides a means by which you can create an algebraic object using the EquationWriter, or an array using the MatrixWriter, then enter the object into a program without having to retype it in command line format.

### 6.5.1 Viewing Objects

The command line also is the standard mechanism for viewing all of an object even when you don't want to edit it. The standard display will show up to four 22-characters lines of the level 1 object, but often this isn't enough. Since viewing an object too large for the display requires many of the same display scrolling operations as editing, the HP48 just uses EDIT as its default object viewer. To streamline the operation, the ▽ key (when no cursor is present) provides single-key access to edit/view an object. This key choice is associated with the interactive stack (section 4.5). You can picture the standard display as a window on the stack, which shows all of the level 1 object, and one-line displays of the remaining stack objects. Then just as you press △ to view the objects above the initial display, you press ▽ to view the rest of the first object, hidden "below" the initial display.

For most types of objects, viewing an object with ▽ is the same as editing the object with ⟨↰⟩ EDIT . However, for unit objects and algebraic objects, ▽ , with its emphasis on *viewing*, activates the EquationWriter to display the objects. Similarly, for arrays ▽ copies the object to the MatrixWriter instead of to the command line. For these three object types, therefore, you must choose which style of edit/viewing you want:

- For an algebraic object or an unit object, use $\boxed{\nabla}$ for the EquationWriter when you want to view the object in a "textbook" form, or if you want to apply operations to the object using RULES in the subexpression menu. Use $\boxed{\Lsh}\,\boxed{\text{EDIT}}$ when you want to edit the object in a way that changes the structure or formal value of the object. For example, consider the algebraic object 'A+B+C+D'. To commute the last two terms to obtain 'A+B+D+C', you can use RULES since the operation is an identity operation that preserves the formal value of the expression. But a modification of the expression to 'A+(B+D)/C', for example, is essentially the entry of a totally new expression where you are just using the original to save a few keystrokes. For this type of change, RULES is of little use; $\boxed{\Lsh}\,\boxed{\text{EDIT}}$ is the appropriate choice.
- For arrays, $\boxed{\nabla}$ to the MatrixWriter is almost always the best choice because of the superior viewing and editing resources it provides compared with the command line. One case where the command line does provide an advantage is that of two- and three-element vectors. For these objects, the command line allows you to enter or edit the components in polar coordinates (section 11.3), whereas the MatrixWriter can deal only with rectangular coordinates.

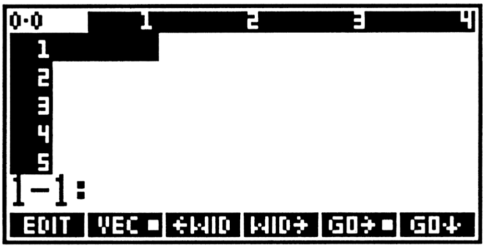
## 6.6 The Matrix Writer

Although command line entry of arrays is straightforward and efficient, the lack of any automatic formatting as you enter numbers makes it easy for you to lose track of which element is which. When you edit an existing matrix, its rows are displayed on separate lines, but there is no attempt to align the columns. With the *MatrixWriter*, the HP 48 provides formatted entry, viewing, and editing of arrays, plus other operations that are useful in array analysis.

There are three methods of activating the MatrixWriter:

- To start entry of a new array, press $\boxed{\Rsh}\,\boxed{\text{MATRIX}}$ . This activates the MatrixWriter display, with empty element cells.
- To view or edit an existing array, press $\boxed{\nabla}$ with the array in level 1. This copies the array to the MatrixWriter.
- EDITΣ (in the $\boxed{\Lsh}\,\boxed{\text{STAT}}$ menu) copies the statistics matrix specified in ΣDAT to the MatrixWriter for viewing or editing.

When you start by pressing $\boxed{\Rsh}\,\boxed{\text{MATRIX}}$, the initial MatrixWriter display looks like this:



The MatrixWriter is modeled in many respects after computer spreadsheet programs. The row-column format of a spreadsheet is a natural one for working with an array, where each *cell* contains one array element, real or complex. Since the HP 48 does not provide symbolic arrays, the MatrixWriter does not implement a spreadsheet's cell formulae, but many other operations are common to the MatrixWriter and spreadsheets. A spreadsheet's row and column labeling translates to the matrix row and column numbers that are shown in the small font along the top and left edges of the display. As an additional reference, the current dimensions of the array are shown in the upper left corner in the format *rows·columns*. In a new array, the dimensions start as 0·0, as in the picture above.

Also like a spreadsheet, the MatrixWriter provides a cell cursor that consists of an inverse-video highlight of the active cell, which you can move to select any cell by using the cursor keys (prefixing a cursor key with ⮣ moves the cursor to the end of the array in the indicated direction). The row-column indices of the cursor are initially displayed in the line above the menu keys. If the current cell contains a value, that value is also displayed with the coordinates in the format *row-column: value*. When you begin to enter or edit an element, the index/value line becomes a command line where you can enter one or more objects.

When you activate the MatrixWriter, a two-page menu of MatrixWriter operations is provided. You can change to other menus as you enter array elements; to return to the MatrixWriter menu, press ⮣ MATRIX.

The first page of the MatrixWriter menu contains the WID→ and ←WID operations, which you may use to increase or decrease the displayed column width to see more or fewer characters in any cell. WID→ increases the column width so that one fewer column is displayed (minimum one), apportioning the extra display space to the remaining columns. Similarly, ←WID increases the number of displayed columns by one (maximum five). The HP 48 remembers the width setting between MatrixWriter sessions.

## 6.6.1 Array Entry

Entering array elements in the MatrixWriter is quite similar to entering numbers onto the stack. You can enter one number at a time, following each with [ENTER], or you can use the command line (which is automatically set to program entry mode) to accumulate several values to be entered sequentially with [ENTER]. The command line is executed in the usual way (section 6.4.3), so you can include sequences that compute an array element as well as entering the element directly. For example, to enter $\sqrt{3}$ as an element, you can type 3 $\sqrt{}$ or '$\sqrt{}$3' →NUM followed by [ENTER]. When you start a command line for one or more elements, the HP 48 notes the current stack depth. After [ENTER], if the stack depth has increased, each new stack object is moved in order to successive array cells, starting with the highest stack level of the new objects. If the stack depth has not increased, the array is not changed. Of course, all of the new objects must be real or complex numbers; if any are other types, then the MatrixWriter exits with the error message Invalid Array Element. When this happens, the existing array is returned to level 1, and any objects from the command line are left in higher levels, with the invalid object in level 2.

While the command line is active, the cursor keys move the character cursor within the command line, not the cell highlight cursor. To move the cell cursor, you must first use [ENTER] (or [ATTN]) to complete command line entry. When you start a new command line, the array display remains visible during entry, unless you enter a newline, at which point the array display disappears in favor of the command line. The array display is restored when you complete command line entry (or if you back up to a single line).

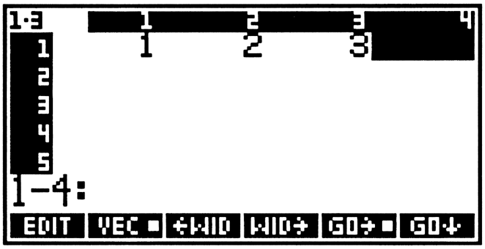
Although our discussion here uses real arrays for examples, the MatrixWriter works equally well with complex arrays. To create a new complex array, you must enter a complex number into cell 1-1. After that, you can enter real or complex numbers; real numbers are automatically converted to complex by [ENTER]. You can not, however, enter a complex value into an array that has been established as a real array.

To enter a completed array onto the stack and exit the MatrixWriter, press [ENTER] with no command line. [ATTN] clears the current command line; if there is no command line, [ATTN] terminates the MatrixWriter but does *not* enter the current array, which is discarded.

Initially, when you are creating a new array, the array dimensions are not determined; successive elements that you enter are placed in cells starting at 1-1 and going across the first row or down the first column. You can choose the direction of entry by using the GO→ and GO↓ keys. The menu key labels for these keys indicate the current mode; if a label has a white box in it, the cursor will move in the direction indicated by the arrow in the label after each cell value is entered. Pressing either key toggles its box on or off; if on, then the box in the other label is turned off.

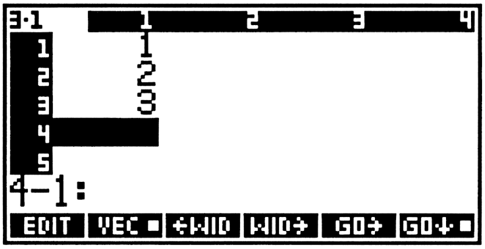
- Choosing GO→ (as indicated by the white box in the key label) causes successive elements to be entered in the first row:

[↱] [MATRIX] ≣GO→≣ 1 2 3 [ENTER] ☞



- Selecting GO↓ causes the elements to be entered in the first column:

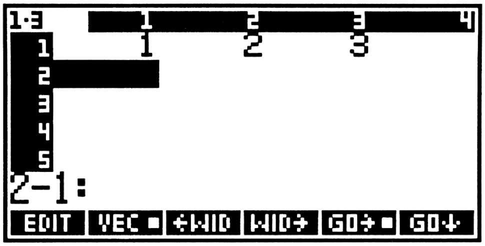
[↱] [MATRIX] ≣GO↓≣ 1 2 3 [ENTER] ☞



- If you turn off both GO→ and GO↓ (by pressing the key that has the white box), then the cursor does not advance after a number is entered, and successive entries overwrite the current cell unless you move the cell cursor to a new cell.

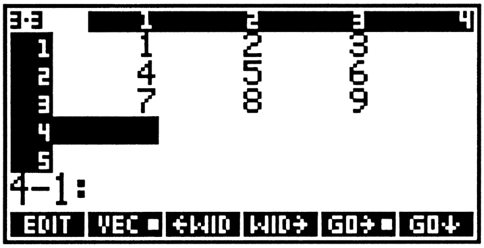
When you are entering an array by rows (GO→), you must specify the width of the array by pressing [▽] after entering the last element in the first row:

[↱] [MATRIX] 1 2 3 [ENTER] [▽] ☞



The cursor has moved to the beginning of the second row. Now succeeding entries will automatically "wrap" at the end of each row:

4 5 6 7 8 9 ENTER ☞



Similarly, if you are entering in columns (GO↓), you must press ▷ to mark the end of the first column. Then succeeding entries will automatically wrap to the next column after each column is full.

You can change directions at any time. If you do so while the cursor is positioned in a partially-completed row or column, the remainder of the row or column is automatically filled with zeros. However, the white-box active symbol does not change to reflect the new direction choice until you actually enter a new cell value.

The HP48 remembers the GO→/GO↓ mode between MatrixWriter settings so that you don't have to reset it to your preference each time you activate the MatrixWriter.

### 6.6.1.1 Vector Entry

By default, the MatrixWriter assumes that when you create an array consisting of only one row, it is to be entered as a vector. When you press ↱ MATRIX , you can observe that the ≣VEC□≣ menu label contains a white square, which indicates that a one-row array will be entered as a vector. If you press ≣VEC□≣ (which removes the white square from the label) any time before the final ENTER , a one-row array is entered as a $1 \times n$

matrix.

When you activate the MatrixWriter via [▽] to edit an existing array, the VEC setting automatically matches the array type, indicating *vector* type (white square) if the array is a vector, or *matrix* type (no white square) otherwise. Thus it is a simple matter, for example, to change a one-row matrix into a vector by pressing [▽] ≣VEC≣ [ENTER]. Note that the VEC setting is irrelevant if an array has two or more rows.

## 6.6.2 Editing Cells

You can change the contents of any MatrixWriter cell by moving the cursor there with the arrow keys, then:

- To replace the current number, type a new command line and press [ENTER].
- To copy the current value to the command line for minor changes, press ≣EDIT≣. Then make any desired changes in the command line text, and press [ENTER] to replace the old value ( [ATTN] [ATTN] cancels the change).

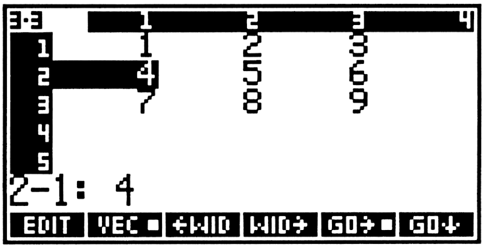
≣EDIT≣ does not change the current menu; if you want the command line EDIT menu (section 6.5), press [◁] [EDIT].

### 6.6.2.1 Changing Array Dimensions

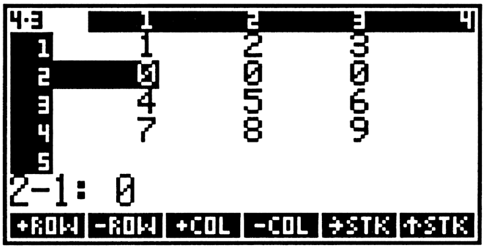
You can add a row or column to the current MatrixWriter array by placing the cursor in the first empty row or column, and entering a value. Unless this happens to be the next normal entry position (determined by GO→ and GO↓), zeros are automatically entered into other cells as necessary to keep the array fully rectangular.

You can also add and delete columns and rows within the existing matrix by using the keys in the second page of the MatrixWriter menu.

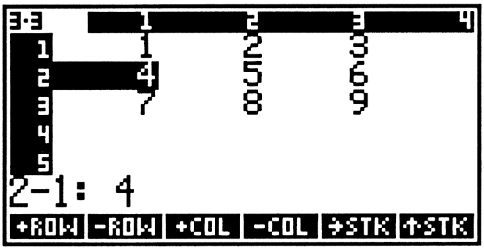
- [+ROW] inserts a row of zeros in the current cursor row, moving the current row and below down by one row. Thus with


```
3·3    1    2    3   4
 1     1    2    3
 2     4    5    6
 3     7    8    9
 4
 5
2-1: 4
```

+ROW yields



- -ROW removes the row containing the cursor, moving the contents of rows below the cursor up by one row. From the preceding picture, -ROW yields



- +COL inserts a new column of zeros at the cursor column, moving the contents of columns at and to the right of the cursor to the right by one column.
- -COL deletes the column containing the cursor, moving the contents of columns to the right of the cursor one column to the left.

### 6.6.2.2 Stack Access

The final two entries in the MatrixWriter menu provide for the exchange of numbers between the MatrixWriter and the stack:

- →STK enters the contents of the cursor cell onto the stack.
- ↑STK replaces the MatrixWriter display with the interactive stack (section 4.5). If there is no command line active, the menu is the full interactive stack menu; otherwise the menu contains only ECHO . In either case, you can use ECHO , to copy a stack object to the MatrixWriter command line (since ECHO creates a command

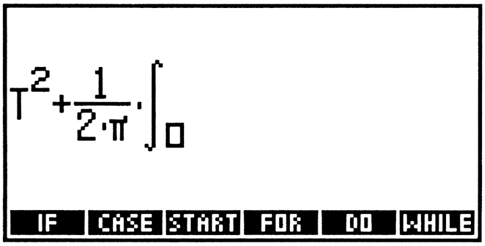
line if one does not already exist, the interactive stack menu subsequently is restricted only to ≣ECHO≣ ). Either ENTER or ATTN terminates the interactive stack and returns to the MatrixWriter display.

## 6.7 The EquationWriter

The HP28C was the first calculator to combine the computational flexibility of RPN with the ability to represent and manipulate algebraic expressions in a readable form. The HP28's expression format resembles that common to most computer languages--expressions are shown as a line of text, using various precedence conventions to minimize the use of parentheses. This *linear format* is much easier to read than the equivalent RPN representation, but still falls short of common written notation (see also section 2.1), in which precedence and other information is conveyed by vertical and horizontal positioning and various special symbols that are not available in the linear format. The HP48 is the first handheld calculator to provide two-dimensional *graphical* entry and display of expressions, by means of the *EquationWriter.*

It is fair to say at the outset that the EquationWriter strains the HP48 processing system to the limit. That system is limited to a modest performance by modern computer standards for reasons of physical size and battery life. The EquationWriter can be frustratingly slow, especially in the backspace operation, which requires rebuilding the entire expression picture. Nevertheless, the EquationWriter is an invaluable tool:

● The entry of constructs such as integrals is much easier in the EquationWriter than using the linear format, simply because the graphical format provides a visual guide to the entry of arguments; when you see a picture like this:

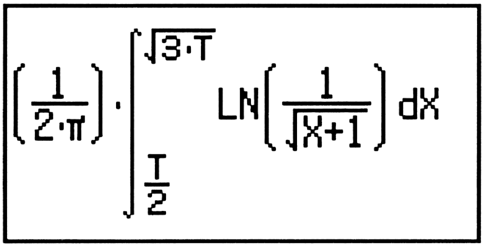


you know it is time to enter the lower limit of an integral. In the linear format, you see

'T^2+(1/(2*π))*∫( ◊)'

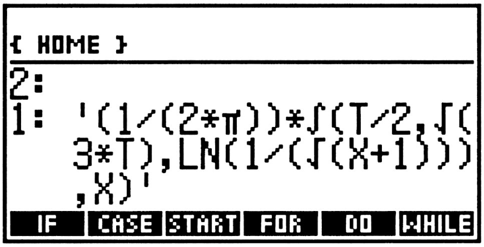
without any help except your memory for choosing which among four arguments is to be entered next.

• After you perform various symbolic calculations, the EquationWriter is very helpful for viewing and understanding a result when the linear format is overwhelmed with parentheses and precedence. The contrast between



$$\left(\frac{1}{2\cdot\pi}\right)\cdot\int_{\frac{T}{2}}^{\sqrt{3\cdot T}} \text{LN}\left(\frac{1}{\sqrt{X+1}}\right) dX$$

and



```
{ HOME }
2:
1: '(1/(2*π))*∫(T/2,√(
   3*T),LN(1/(√(X+1)))
   ,X)'
```

speaks for itself.

• For the interactive application of mathematical identity rules to rearrange and solve expressions the HP48 RULES system using the EquationWriter is a distinct improvement over HP28 FORM, in which specification of a subexpression often is effectively impossible because of the superabundance of parentheses.

The EquationWriter is specifically *not* designed for *editing* expressions. It will not permit operations that change the formal mathematical value of an expression, such as inserting or deleting parentheses, substituting different functions, inserting or deleting terms, etc., except by means of the EDIT operation in the subexpression menu, which activates the command line editor for a selected subexpression.

The *subexpression* (section 3.5.2.1) is a key concept in EquationWriter operation. Again, a subexpression is any portion of a mathematical expression that can stand alone; that is, it can be treated as a complete expression by itself. Specifically, a subexpression consists of a number, a name, or a function and its arguments. A number--real or complex--is the simplest case; if you like, you can think of a number as a function that takes no arguments and always returns the same value.

For example, consider the expression $a + \sin(b-c)$. Rewriting this in Polish notation (section 2.1), you obtain $+(a, \sin(-(b,c)))$. The "outermost" subexpression is the entire expression, consisting of the function + and its arguments $a$ and $\sin(-(b,c))$. Each of the two arguments is a subexpression--the first is just the name $a$, the second is the function sin and its argument $-(b,c)$. The latter in turn is a subexpression consisting of − and its arguments $b$ and $c$, and so on as you peel off the layers of parentheses.

## 6.7.1 The EquationWriter Display

While the EquationWriter is active, the text screen is dedicated to the expression picture. Menu keys retain their normal definitions and menus; however, keys that correspond to commands that have no meaning in an algebraic context merely beep and do nothing. Similarly, the primary and shifted keyboard keys are usable only if they make sense:

- Keys corresponding to algebraic functions enter those functions into the expression in their graphical form.
- Menu keys, $\boxed{↱}\boxed{\text{LAST MENU}}$, $\boxed{\text{NXT}}$, and $\boxed{↰}$ and $\boxed{↱}\boxed{\text{PREV}}$ switch menus as usual.
- Alpha-shifted keys retain their usual actions.
- User mode is available, although the USER annunciator is not visible; however, only keys defined with algebraically legal objects are active.
- The cursor keys have special meanings that combine cursor "movement" with mathematical function entry.
- $\boxed{⇦}$ performs a limited destructive backspace.
- $\boxed{,}$ enters a comma or semicolon, to separate the arguments of multi-argument functions.
- $\boxed{=}$ enters the = sign for an equation, or for the lower limit of a sum.
- $\boxed{\text{SPC}}$ is used to enter any "required" characters--separators (comma or semicolon) between arguments, = signs in $\Sigma$ start assignments, etc.
- $\boxed{↰}\boxed{\text{EDIT}}$ transfers the current EquationWriter expression to the command line.

- $\boxed{\text{↱}}$ $\boxed{\text{OFF}}$ turns the HP48 off normally; pressing $\boxed{\text{ON}}$ restores the active EquationWriter display intact.
- $\boxed{\text{↰}}$ $\boxed{\text{GRAPH}}$ switches off the menu and the cursor so that you can use the cursor keys to scroll the current expression picture through the display. This permits viewing portions of the expression that have moved out of view during expression entry. A second press of $\boxed{\text{↰}}$ $\boxed{\text{GRAPH}}$ restores the menu and puts the cursor back at the end of the expression for further entry.
- $\boxed{\text{STO}}$ captures the current expression picture as a graphics object on the stack. (This is analogous to the $\boxed{\text{STO}}$ action in interactive plotting.)
- $\boxed{\text{EVAL}}$ (or $\boxed{\text{↱}}$ $\boxed{\text{→NUM}}$ ) is equivalent to $\boxed{\text{ENTER}}$ $\boxed{\text{EVAL}}$ ( $\boxed{\text{→NUM}}$ ), for immediate entry and evaluation of the current expression.
- $\boxed{\text{↱}}$ $\boxed{\text{( )}}$ toggles *implied parentheses mode* on and off (default on). See section 6.7.2.5.
- $\boxed{\text{↱}}$ $\boxed{\text{" "}}$ captures the current expression as a string object on the stack (section 6.7.5).
- $\boxed{\text{↱}}$ $\boxed{\text{RCL}}$ takes an (algebraically legal) object from the stack and appends it to the current expression. The object may also be a string, such as that captured by $\boxed{\text{↱}}$ $\boxed{\text{" "}}$ .
- $\boxed{\text{↱}}$ $\boxed{\text{CLR}}$ clears the current expression without leaving the EquationWriter.

EquationWriter execution is terminated by $\boxed{\text{ENTER}}$, which closes all pending subexpressions and enters the current expression onto the stack, where you will see it in linear form. $\boxed{\text{EVAL}}$ and $\boxed{\text{↱}}$ $\boxed{\text{→NUM}}$ act as shortcuts; either key performs $\boxed{\text{ENTER}}$ and then executes its normal operation before returning to the standard environment. You can also exit from the EquationWriter with $\boxed{\text{ATTN}}$ , which returns to the standard environment but abandons the current EquationWriter expression (if you activated the EquationWriter with $\boxed{\nabla}$, the original level 1 object is preserved).

#### 6.7.1.1 Invoking the EquationWriter

You may activate the EquationWriter in three ways:

- $\boxed{\text{↰}}$ $\boxed{\text{EQUATION}}$ starts the EquationWriter with an initially blank screen, for the entry of an entirely new expression.
- Pressing $\boxed{\nabla}$ with an algebraic object or a unit object in level 1 activates the EquationWriter with that object as its current expression.
- →GROB (section 10.3.2) specified with the 0 font argument creates a graphics object containing the EquationWriter picture of an algebraic object or a unit object.

## 6.7.2 Basic Expression Entry

Entering an expression in the EquationWriter environment consists of "drawing" the expression in a two-dimensional graphical form, in more-or-less the same order as the expression is written by hand, working left-to-right. Object entry takes place at the cursor, which is always at the end of the new expression. All three HP 48 character fonts are used in building an expression picture, starting with the large font for the main line of an expression, dropping to the medium font for exponents and for the limits of integrals and sums, and finally to the small font for exponents of exponents, etc. The cursor grows and shrinks also to match the current font size at the cursor.

To minimize memorization of arbitrary key sequences, the EquationWriter makes as close a correspondence as possible between cursor movement and the hand motions you make when writing an expression on paper. The crucial key is $\boxed{\triangleright}$, which terminates, or *closes*, entry of a subexpression. The choice of $\boxed{\triangleright}$ arises from a general model of entering expressions from left to right. The cursor is always at the right end during expression entry, so pressing $\boxed{\triangleright}$ is taken to mean "go even farther right"--i.e. close the current subexpression and start a new one. In some cases, such as when entering an exponent or a numerator, the natural terminating motion is "down"; hence $\boxed{\triangledown}$ is also allowed, and is equivalent to $\boxed{\triangleright}$. Closing a subexpression means:

- entering a right parenthesis (this is the only way to do this);
- finishing an exponent;
- finishing a numerator or denominator;
- completing a square root or XROOT argument;
- completing any of the various arguments in a multi-argument function. In this case, $\boxed{\triangleright}$ enters an argument separator "," or ";" to separate parenthesized arguments, or moves to the next argument location in structures such as integrals, sums, and | (*where*).

When the cursor is in a position representing the end of several nested subexpressions, you can use $\boxed{\Rsh}$ $\boxed{\triangleright}$ as a shortcut to complete all pending subexpressions. It is equivalent to pressing $\boxed{\triangleright}$ repeatedly until all subexpressions are closed and the cursor is at the right end of the main entry line. Thus if you have entered

$$\frac{A}{B^{C^{D\,\square}}}$$

pressing $\boxed{\Rsh}\boxed{\triangleright}$ closes both exponents and the fraction to

$$\frac{A}{B^{C^D}}\ \square$$

The space key plays a role similar to, but not quite the same as $\boxed{\triangleright}$. $\boxed{\text{SPC}}$ (you can also use $\boxed{\leftarrow\!\rceil}\ \boxed{,}$) is used to separate the required arguments of a multi-argument function (not counting infix functions, where the function itself separates the arguments). Like $\boxed{\triangleright}$, $\boxed{\text{SPC}}$ enters an argument separator "," or ";" or moves to the next argument location. However, you can not use $\boxed{\text{SPC}}$ to terminate the *final* argument of any function; it will beep and display Invalid Syntax to indicate that no further arguments are permitted. Another distinction between $\boxed{\text{SPC}}$ and $\boxed{\triangleright}$ is in their application to functions of an indefinite number of arguments (including user-defined functions): $\boxed{\text{SPC}}$ *must* be used to separate the arguments, since $\boxed{\triangleright}$ will close the subexpression. For example, if you have entered

UDF(1▯

then pressing $\boxed{\text{SPC}}$ yields

UDF(1,▯

ready for another argument, whereas $\boxed{\triangleright}$ gives

UDF(1)▯ .

Although there is some overlap between the actions of $\boxed{\text{SPC}}$ and $\boxed{\triangleright}$, we recommend that you use $\boxed{\text{SPC}}$ for separating successive arguments within parentheses, and $\boxed{\triangleright}$ for moving between physically separated argument locations.

You naturally can not leave any required argument location empty; if you press $\boxed{\triangleright}$ in such a situation, it just does nothing and leaves the cursor in place. You can not properly close a subexpression unless all of the required arguments of the function that defines the subexpression are present. $\boxed{\text{ENTER}}$ in this case beeps and displays Incomplete Subexpression

Upward motion when writing an expression can arise from a number of constructs, in particular exponents and division numerators. The EquationWriter chooses the latter for its $\boxed{\triangle}$ action, since exponentiation is easily represented by the $\boxed{y^x}$ key, and since two keys are really needed for division--see the preceding section 6.7.2.4.

Finally, motion to the left implies a correction of already-entered symbols. The simplest case is the erasure from the right represented by $\boxed{\Leftarrow}$ (section 6.7.4). $\boxed{\triangleleft}$ is directed to more elaborate manipulations; it activates the subexpression environment (section 6.7.6).

### 6.7.2.1 Number Entry

Numbers are entered into the EquationWriter in same manner as in the command line, with certain exceptions that arise from the non-RPN context:

- [+/-] merely echoes a minus sign at the cursor and does not affect any sign to the left of a number. You can use either [+/-] or [-] to prefix a negative quantity or for subtraction.
- [EEX] just types an E at the cursor.
- You must separate the real and imaginary parts of a complex number with a comma or a semicolon. After you enter the real part, [SPC] will enter the separator appropriate for the current fraction mark mode.

### 6.7.2.2 Names and Prefix Functions

You can enter a global or a local name by typing the name with alpha keys, or by pressing a CST, VAR, or LIBRARY menu key corresponding to the name. The same method applies to ordinary prefix functions (functions with their arguments following within parentheses), including functions represented by XLIB names, except that the EquationWriter is also sensitive to their definitions. This means that when you complete entering a function name, by pressing [▷] , [◁] [( )] , [SPC] , or another function key, the EquationWriter immediately checks the syntax, and adds a following left parenthesis if needed. Furthermore, if the entry is an RPN command name, it is rejected with the Invalid Syntax message.

Since the EquationWriter does not allow entry of spaces ( [SPC] enters argument separators), you must enter those infix functions with multi-character names, such as MOD, AND, NOT, etc., by pressing a menu key or a user key for the function.

### 6.7.2.3 +, −, ×

These *infix* operators (functions that appear between their arguments) appear in their natural form, with the extension that the EquationWriter's graphics allow substitution of the centered dot "·" instead of the more obtrusive "*" of the linear format:

[A] [×] [B] ☞ A·B.

Although the HP 48 does not explicitly support *implied* multiplication (in order to provide for multi-character variable names), the EquationWriter will automatically insert a multiply ("·") whenever the syntax is sufficiently unambiguous to permit it:

- in front of an alpha character entered after a number: [1][A] ☞ 1·A □
- between right and left parentheses: ...) [⮣][( )] ☞ ...)·( □
- in front of prefix functions (unless typed in with alpha keys): [A][SIN] ☞ A·SIN( □
- in front of the divide bar: [A][Δ] ☞ A· $\frac{\Box}{}$
- in front of square root: [A][√] ☞ A·$\sqrt{\Box}$

You should need to use [×] only to separate objects entered with typed sequences rather than with single-keystrokes, such as the products of numbers and names. If you are uncertain of whether implied multiplication will happen, it is always acceptable to press [×] directly.

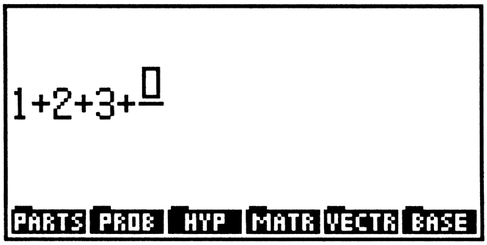
**6.7.2.4 Division**

Symbolic fractions are displayed by the EquationWriter as a numerator above a divide bar above a denominator, with the divide bar two pixels wider than the longer of the numerator and the denominator (left-to-right length). There are two ways to enter a fraction. The first is to enter the numerator, press [÷] , then enter the denominator, terminating the latter with [▷] . For example,

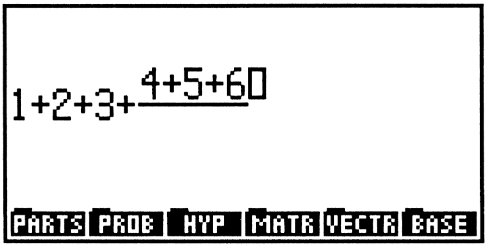
[1][÷][2][+][3][▷] yields $\frac{1}{2+3}$ □.

With this method, which is derived from the ordinary infix divide used in the linear format, it is not necessary to enclose the denominator in parentheses. However, if the numerator contains more than one object, it is necessary to enclose the numerator in parentheses to indicate the extent of the numerator subexpression. Requiring numerator parentheses violates the spirit of entering expressions as you write them, so an alternate method is provided.

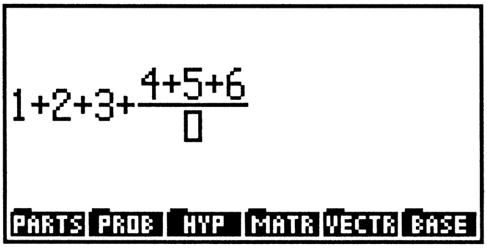
The second method uses [Δ] to mark the start of the numerator, following the motion of a pencil moving up the paper as you start writing a numerator. Pressing [Δ] moves the cursor up half a line and draws a divide bar under the cursor:



As you enter subsequent objects, the bar stretches under the new objects (the stretching occurs when each object is terminated, not when individual letters or digits are typed):


$1+2+3+\frac{4+5+6\square}{}$

You signal the end of the numerator by pressing [▽] or [▷] , whereupon the cursor moves down to the empty denominator:


$1+2+3+\frac{4+5+6}{\square}$

Now the divide bar stretches further when and if the denominator width exceeds that of the numerator. [▷] terminates the denominator, redraws the fraction with the numerator and denominator centered, and moves the cursor to the right end of the fraction.

The division initiated by [△] actually corresponds to the prefix function RATIO instead of to /. This function is equivalent to / when executed, and is automatically converted to / when you exit the EquationWriter with [ENTER] . Because it offers no non-EquationWriter functionality not provided by /, RATIO does not appear in any menus. Unless you deliberately enter it in the command line, the only time you are likely to see RATIO by that name is in strings created by the [↱] [" "] key from within the EquationWriter (section 6.7.5).

#### 6.7.2.5 Exponents

You enter an exponent by pressing [$y^x$] immediately following the object or subexpression (the *base*) to be exponentiated. This causes the cursor to move up half a line, and to reduce to the next-smaller font (unless already using the small font). If the base expression is defined by a multi-argument function, parentheses are automatically added around the expression if they are not already present. [▷] or [▽] terminates the exponent entry, moves the cursor down to the base line, and returns to the previous font.

You must parenthesize multi-term base expressions, as you would in written notation. It is not necessary to parenthesize the exponent, regardless of its structure. However, this means that you must always use [▷] or [▽] to terminate the exponent, which may appear to be an inconvenience if you are entering, for example, a polynomial containing nothing but single term exponents. For this reason, the EquationWriter allows you to disable *implicit parentheses*.

In the normal operation of [÷], √, and [$y^x$], subsequent entry adds objects to the denominator, square root argument, and exponent subexpressions, respectively, as if invisible parentheses surrounded the subexpression. If you press [↱] [{ }], the implicit parenthesization is disabled (Implicit ( ) off is displayed), and entry of the subexpression following one of these operators is automatically terminated by any subsequent function key. Moreover, that is the only way to terminate (except [ENTER]); pressing [▷] has no effect. This is convenient for entering polynomials: each exponent is completed by entry of the function that starts the next term. A second use of [{ }] (Implicit ( ) on) reenables implicit parenthesization (which is always active upon entry to the EquationWriter).

### 6.7.3 Special Forms

In addition to basic expression entry described so far, using names, numbers, prefix functions, and the infix functions +, −, × and ÷, the EquationWriter provides special forms for square root, xth-root (XROOT), integral (∫), derivative (∂), sum (Σ), and *where* (|).

#### 6.7.3.1 Square Root

Pressing [√] displays a square root symbol with an overbar above the cursor: $\sqrt{\square}$. As you enter the argument, the overbar stretches horizontally (in the manner of the divide bar) and the leading √ stretches vertically, to match the growing argument. As usual, [▷] or [▽] marks the end of argument entry, whereupon the overbar shrinks if necessary to the length of the argument without the cursor, and the cursor moves two dot columns to the right of the end of the overbar.

### 6.7.3.2 *x*th-Root

Whether XROOT is considered as a prefix or an infix function in its written form is ambiguous. In the EquationWriter, you press the $\boxed{{}^{x}\sqrt{y}}$ key *before* entering either argument. This moves the cursor up a half line, and reduces the font (but does not yet enter the $\sqrt{}$ symbol). You then enter the *x* argument; when you press $\boxed{\triangleright}$ (or $\boxed{\triangledown}$ or $\boxed{\text{SPC}}$ ) to terminate the argument, the $\sqrt{}$ symbol is drawn as well:

$$\cdots\ +{}^{x}\sqrt{\square}$$

Now you enter the *y* argument, during which the $\sqrt{}$ symbol stretches as for ordinary square roots. Another $\boxed{\triangleright}$ terminates the entire XROOT subexpression.

The fact that the *x* argument is written in the EquationWriter before the *y* argument means that the linear format syntax for XROOT is XROOT(*x*,*y*). However, you should note that the RPN syntax for XROOT is *y x* XROOT; *x* is entered after *y*. This makes XROOT consistent with ^, and more convenient for manual calculations, but it means that XROOT is an (the only) exception to the usual HP 48 rule that the order of arguments within parentheses is the same as the order in which they are entered for RPN execution.

### 6.7.3.3 Derivative

Pressing $\boxed{\partial}$ enters this form:

$$\frac{\partial}{\partial\ \square}$$

The cursor is positioned at the differentiation variable name field. Keying in a name terminated by $\boxed{\triangleright}$ then yields

$$\frac{\partial}{\partial name}(\square$$

Now the cursor is positioned for entry of the expression to be differentiated; the expression's entry is also terminated by $\boxed{\triangleright}$, which closes the parentheses.

### 6.7.3.4 Integral

$\boxed{\vdash\!\triangleright}\boxed{\int}$ draws a large (about three times a character's height) integral sign, with the cursor positioned at the lower integration limit:

$$\int_{\square}$$

The integral sign changes in size as the integral's arguments, so that the symbol is as tall as the sum of the heights of the limits and the integrand. The integrand does not overlap either limit horizontally or vertically. It starts at the horizontal position beyond the right ends of the of the lower and upper limits.

An integral has four fields:

$$\int_{lower}^{upper} integrand\,\mathrm{d}\,name$$

You enter these in the order *lower*, *upper*, *integrand*, *name*, ending each successive field with $\boxed{\triangleright}$ . Terminating the integrand automatically enters the "d" symbol a half space past the end of the integrand. Terminating the name (the integration variable) completes entry of the entire integral and moves the cursor a full space to the right of the end of the name.

The integration variable name field can only contain a name; the other fields can contain arbitrary expressions.

### 6.7.3.5 Summation

$\boxed{\Rsh}$ $\boxed{\Sigma}$ draws a large summation symbol Σ, with the cursor positioned at the lower integration limit:

$$\sum_{\square}$$

Unlike the integral sign, the summation symbol does not change in size as the various arguments are entered. The start index expression grows downward and the stop limit expression upward to avoid overlapping the Σ itself. Similarly, the summand expression starts at the horizontal position beyond the right ends of the of the index expressions

An sum has four fields:

$$\sum_{name=start}^{stop} summand$$

You enter these in the order *name*, *start*, *stop*, *summand*, terminating each successive field with $\boxed{\triangleright}$ , which moves the cursor to the next field. When you end the name field (which can only contain a single name) $\boxed{\triangleright}$ automatically enters the "=" symbol after the name (you can also use $\boxed{\Lsh}$ $\boxed{=}$ ). Terminating the summand completes entry of the entire structure, and moves the cursor one space to the right of the end of the summand expression.

#### 6.7.3.6 Where

The function | (pronounced "where") is an infix function with one preceding argument and an indefinite number of following arguments. Pressing ≣|≣ draws a vertical bar and places the cursor at the bottom right of the bar:

$$A(X,Y)\,|_{\square}$$

At this point, you enter a series of one or more assignments of the form *name=value*, separated by commas or semicolons. You can use $\boxed{\text{SPC}}$ to enter either = or the comma, or you can use $\boxed{\leftarrow\!\!\!\lrcorner}\,\boxed{=}$ or $\boxed{\leftarrow\!\!\!\lrcorner}\,\boxed{,}$ as appropriate. A typical entry looks like this:

$$A(X,Y)\Big|_{X=2,Y=3\ \square}$$

Pressing $\boxed{\triangleright}$ after completing an assignment expression completes the | subexpression.

#### 6.7.3.7 Units

In the EquationWriter, the underscore delimiter _ is treated as an infix function (section 2.1); no other special provision is required for units. You enter a unit object in the usual form *magnitude_units*, where the *units* part is a subexpression with exponents, multiplication signs, and divide bar displayed in the usual EquationWriter style. You can use various UNITS menu keys function as typing aids during unit entry.

In the entry of the unit part, the EquationWriter does not attempt to prevent you from entering otherwise valid subexpressions that contain functions not permitted in units. In this respect the EquationWriter behaves the same as the command line for the case where a unit object is entered within an algebraic object. No error is reported until the resulting expression is evaluated.

### 6.7.4 Correcting Mistakes

The EquationWriter provides a simple destructive backspace operation ( $\boxed{\Leftarrow}$ ) for correction of ordinary wrong-key-press errors. "Simple" here means that while you can quickly erase digits and letters, backing up over a function or into any closed subexpression is slow. This is because the HP48 must recompute the entire picture as if it had been entered from the start. This is preferable to making you retype the expression yourself, but the delay can be frustrating. Furthermore, the destructive backspace performed by $\boxed{\Leftarrow}$ is not an suitable method for structural revisions, such as inserting new terms and parentheses. For these reasons, the command line editor is made available from within the EquationWriter. Pressing $\boxed{\leftarrow\!\!\!\lrcorner}$ $\boxed{\text{EDIT}}$ copies the entire current

EquationWriter expression into the command line (this is also true in RULES operation, where ◁ EDIT copies the selected subexpression to the command line). ' ' delimiters are automatically inserted around the command line object to identify it as an algebraic type. Note, however that you can only edit a complete expression; you must make temporary entries for any missing arguments in order to start the command line edit (once the command line is active, you can replace the dummy entries).

Normal command line facilities are available, including the interactive stack ECHO . The entry mode is automatically set to ALG PRG. ENTER returns the edited expression to the EquationWriter; ATTN cancels the edit and restores the original expression in the EquationWriter.

## 6.7.5 Stack Access

In addition to the "back door" to the stack via ↑STK from the command line, the EquationWriter provides more direct object exchange with the stack. For example, you can capture the current EquationWriter picture by pressing STO ; a graphics object representing the picture is invisibly entered into level 1. The current expression does not have to be complete, which is useful when you are trying to capture a series of step-by-step pictures of EquationWriter operation.

You can also store the actual entry sequence that led to the current expression at any time by pressing ▷ " " . The choice of this key arises from its association with strings, since the key sequence is stored on the stack as a string. You can later use the string as a typing aid for reentering the same expression: pressing ▷ RCL with such a string in level 1 drops the string from the stack and appends it to the current expression as if the string characters were typed in. When you observe an EquationWriter string object on the stack, you will notice that the expression represented by the string follows different precedence rules than used in ordinary algebraic objects; for example, the expression

$$\left(\frac{1}{2+3}\right)^4$$

appears as

" RATIO(1,2+3)^(4)"

in string form, but as

'(1/(2+3))^4'

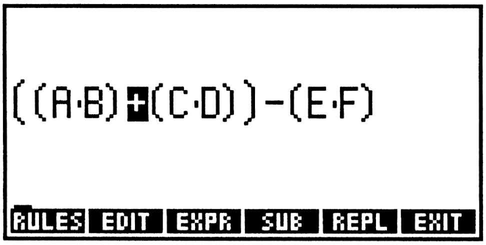
in the linear form of an algebraic object. This difference makes it impractical for you to create these strings other than from within the EquationWriter. Instead, you can use proper algebraic objects, since the EquationWriter ⮣ RCL can take any algebraic object from the stack as well as an EquationWriter-generated string.

You may also see EquationWriter strings on the stack when the HP48 runs out of memory during EquationWriter entry, which causes the current expression to be saved on the stack as a string. After you free some memory, you can restart the Equation-Writer, and use ⮣ RCL to recover the expression.

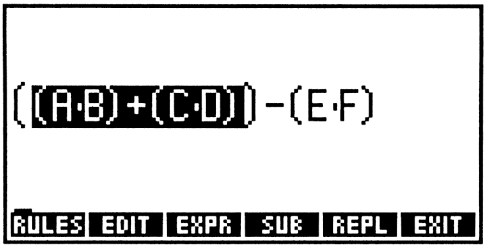
## 6.7.6 Subexpression Operations

During expression entry, the EquationWriter cursor is an open box that is always at the *end* of the expression--the point at which object entry is taking place. Pressing ◁ moves the cursor "back into" the expression, simultaneously activating the *subexpression menu*. The box cursor disappears, to be replaced by the *subexpression cursor*, an inverse-video highlight of an object, which you can move around the expression to select different objects and subexpressions.

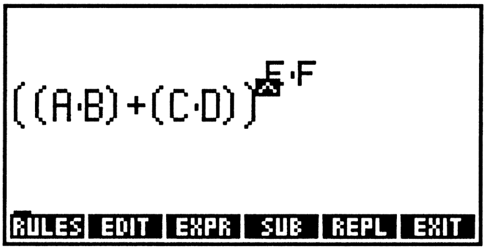
As discussed at the start of section 6.7, a subexpression is defined by a function and its arguments, where we include the zero-argument cases of names, numbers, and symbolic constants. All of the operations in the subexpression menu apply to the subexpression selected by the cursor. As you move the cursor, it jumps from object to object, but at any point you can expand the cursor to highlight an entire subexpression by pressing EXPR. For example, with the cursor positioned like this:


((A·B)+(C·D))-(E·F)

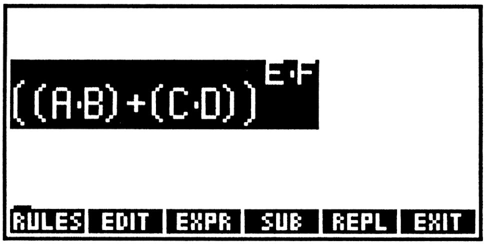
pressing EXPR shows the subexpression defined by the object +:



The exponentiation function ^ is "invisible" in the EquationWriter, since an exponent is defined by its geometrical position. However, when you move the cursor between the base and the exponent, the ^ pops into view so that you can select the corresponding subexpression:

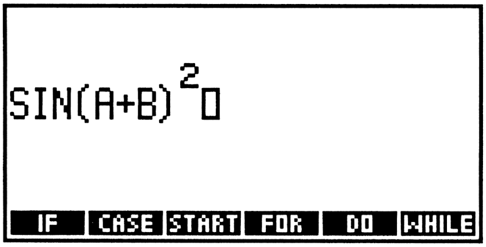


Then EXPR :



All subexpression menu operations are applied to the selected subexpression. These operations are defined as follows:

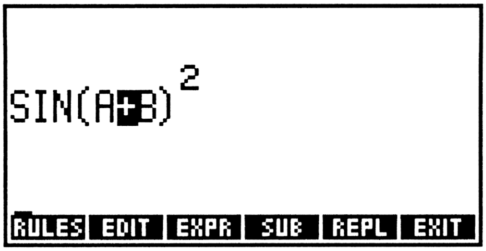
- ≣RULES≣ provides a set of identity operations that you may apply to the subexpression. We will defer a detailed discussion of RULES to *Part II*, where we will describe the broader topic of symbolic algebra on the HP 48.
- ≣EDIT≣ copies the selected subexpression to the command line, where you can use character editing to change it to any new subexpression (the only restriction is that certain arguments, such as a differentiation or summation variable names, must remain as names). [ENTER] restores the EquationWriter picture, with the edited subexpression replacing the original. You can also cancel the edit with [ATTN], leaving the initial subexpression intact.
- ≣EXPR≣ switches the cursor between highlighting an object and highlighting a subexpression. Pressing a cursor key to move the cursor always reverts to the object highlight.
- ≣SUB≣ enters the selected subexpression onto the stack. When you leave the EquationWriter, any objects entered by ≣SUB≣ will appear starting in level 2, since the full EquationWriter expression object is returned to level 1 (this is also true for objects entered by [STO] or [↱] [" "].
- ≣REPL≣ replaces the selected subexpression with an object taken from the stack. The object is taken (and dropped) from level 1. if you entered the EquationWriter via [▽] on an algebraic object or a unit object, that object is removed from the stack for the duration of EquationWriter execution. Objects intended for REPL should therefore start in level 2 (before [▽]). For example, to replace the A+B in 'SIN(A+B)^2' with √(C+D), start with the 'SIN(A+B)^2' in level 1, and '√(C+D)' in level 2. Then [▽] displays the sine expression:



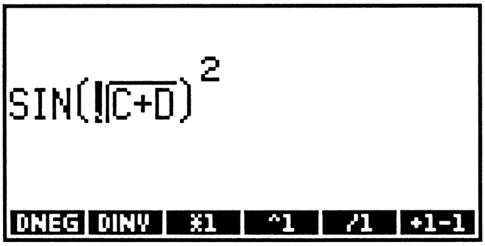
```
SIN(A+B)²□

 IF  CASE START FOR  DO  WHILE
```

[◁] four times highlights the +:



Now REPL makes the replacement:



The highlight is now on the √, since it is the function that defines the replacement subexpression. You might also notice that the menu changes to the RULES menu appropriate for √; the REPL substitution is treated as an extension of RULES even though it is not necessarily an identity operation. Any cursor movement restores the subexpression menu.

- EXIT returns the EquationWriter to entry mode, with the box cursor at the end of the expression.

# 7. Customization

One of the strongest features of the HP 48 is its extensive customization ability. That is, for the sake of a particular application, or just for general use, you can turn the HP 48 into a highly personalized tool, focusing on the computations and interactions that you prefer. The customizing facilities of the HP 48 are as follows:

- 57 *system flags* give you on/off control over the many HP 48 modes.
- *Custom menus* enable you to augment the built-in menus with your own specialized menus.
- *Key assignments* change the actions of any of the shifted or unshifted keys.
- The *vectored ENTER* mechanism allows you to redefine the way the command line interprets its entries, and to change what the HP 48 does after each keyboard action.

The basis of all of these mechanisms is the HP 48's programming capability, which allows you to define complicated procedures to associate with keys and menus. In this chapter, we will concentrate on the explicit customizing techniques, including some programs that illustrate the methods as well as serving as programming examples.

## 7.1 Modes and Flags.

A *mode* is a calculator setting that acts as a form of global argument for certain operations, that saves you from having to supply that argument every time you execute the operations. The classic example of a mode, which is common to most scientific calculators, is the *trigonometric angle mode*, which determines how the trigonometric functions interpret their arguments and results. The sine function is defined mathematically in terms of dimensionless arguments expressed in radians; to compute the sine of an angle expressed in degrees, you must multiply the argument by $\pi/180$ before applying the sine algorithm. On the HP 48, you can skip the multiplication by setting the angle mode to degrees, in which the SIN command assumes that its (real) arguments are entered in degrees. Similarly, the ASIN command returns its (real) results in degrees, performing the multiplication by $180/\pi$ automatically.

The current setting of a calculator mode is recorded by means of one or more *flags* where a flag is a memory location that contains one binary bit. For a simple "on/off" mode like the ticking clock display, only one flag is needed. A single flag is usually considered to be *set* or *clear*--if the flag bit is 1, the flag is *set*; if it is 0, the flag is *clear*. For a multi-state mode like the angle mode, which has three settings, two or more flags are needed. In these cases the flag values taken together make up a binary number with two or more digits, ranging from the two-bit number that encodes the angle mode up to the six-bit number that records the binary integer wordsize.

Some HP 48 modes are controlled by flags that are only accessible to the operating system. You must switch these modes with manual operations; there is no programmable control. Examples of these modes are the stack recovery or command stack active/disabled modes, which are selected by means of menu keys in the [◁] [MODES] menu; command line insert/replace, selected by the ≡INS≡ menu key in the EDIT menu, and the Matrix Writer entry-order mode, controlled by the ≡GO→≡ and ≡ GO↓ ≡ menu keys.

The majority of HP 48 modes are represented by *user flags*, so called because you can control their values manually and in programs. There are 128 user flags, numbered from −63 to +64. Flags in the range −63 to −1 are used for HP 48 modes and *signals*. Signal flags are used to convey the nature of certain results, such as floating-point overflow, when the use of an additional stack result would be inconvenient. There are a few unused flags in this range, which is ordered to keep related flags in groups numbered starting with a multiple of 5, plus 1. Flags 0-31 are strictly reserved for users' programs. The remaining flags 32-64 are nominally reserved for libraries (the HP Solve Equation Library uses flags 60-62), but you can use any of these flags as long as they don't conflict with the libraries' use.

The least commonly altered modes, such as the Kermit receive overwrite mode, or the vectored ENTER mode (section 7.4), can only be selected by means of their respective numbered flags. More common modes like the ticking clock display or symbolic execution (section 3.5.5.2) can be user flag controlled but also have keyboard or menu keys with mnemonic labels (e.g. ≡CLK≡ and ≡SYM≡ ). Finally, the most important modes have dedicated commands, like FIX and DEG, which are programmable as well as mnemonic. (The relative importance of the various modes was decided by the designers--if your favorite mode was relegated to a mere flag, you can always write a little program to alter the mode, and give it a mnemonic name).

In the HP 48, the default state of all of the system mode flags is clear, except for the binary integer wordsize flags −5 to −10, which are set. This means that in general, a *clear* mode flag means "do the default behavior" and a *set* flag means "do the non-default behavior" for the affected operations. Thus if you're trying to remember whether to set or clear a particular flag in order to select a mode, you can use the calculator's defaults as a guide (assuming that you can remember those).

## 7.1.1 Flag Commands

The commands you need to select a mode by means of its flags are SF (*Set Flag*) and CF (*Clear Flag*), which set and clear the flag specified by a real number argument. For example, −3 SF turns on numeric evaluation mode; −3 CF turns it off. You can also determine the state of a flag; for example, 9 FS? returns a 1 to the stack if flag 9 is set, or a 0 otherwise. The real numbers 0 and 1 used in this context are called *stack flags*, because they can represent the binary values of a user flag so that you can manipulate those values on the stack. The FS? command in effect copies a user flag value to the stack. Stack flags are also useful in programming as logical *false* (0) or *true* (1) values (section 7.1). Note that *set*, *true*, and 1 are synonymous, as are *clear* and *false*, and 0.

In addition to FS?, the HP48 also provides FC?, which returns *true* if a flag is clear; and FS?C and FC?C which test a specified flag and then clear it. You can also recall the values of all 128 flags by executing RCLF (*ReCalL Flags*). This command returns a list of the form { *#m* *#n* }. *#m* is a 64-bit binary integer representing flags −63 to 0; its leftmost, or most-significant bit corresponds to flag −63, and its least-significant bit is flag 0. *#n* similarly represents flags 1 (least-significant) through 64 (most-significant). The principal use of RCLF is to record the values of the flags so that those values can be restored later by the complementary command STOF (*STOre Flags*). STOF takes a list like that returned by RCLF and sets all 128 flags according to the values of the two binary integers in the list. Examples of using RCLF and STOF are shown in the programs ASN41 (section 7.2.1.1) and XARCHIVE (section 5.3.4).

STOF and RCLF provide a convenient means for applying individual bit operations to binary integers. The programs listed next allow you to set, clear, and test a specified bit in a binary integer, where the bits are numbered from 0 as the least significant (rightmost) bit.

| SB | *Set Bit* | | | 8DB7 |
|---|---|---|---|---|
| | *level 2* | *level 1* | | *level 1* |
| | *#n* | *m* | ☞ | *#n'* |

```
« RCLF ROT STOF         Swap system flags and binary integer.
  SWAP NEG SF           Set the bit.
  RCLF 1 GET            Get the new integer value.
  SWAP STOF             Restore the original flags.
»
```

| CB | *Clear Bit* | | | AC26 |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | *#n* | *m* | ☞ | *#n'* |

```
<< RCLF ROT STOF          Swap system flags and binary integer.
   SWAP NEG CF            Set the bit.
   RCLF 1 GET             Get the new integer value.
   SWAP STOF              Restore the original flags.
>>
```

| BS? | *Bit Set?* | | | 822A |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | *#n* | *m* | ☞ | *flag* |

```
<<
   RCLF ROT STOF          Swap system flags and binary integer.
   SWAP NEG FS?           Test the bit.
   SWAP STOF              Restore the original flags.
>>
```

## 7.1.2 System Flag Assignments

Table 7.1 summarizes the HP48 mode and signal flags, showing the modes associated with setting each flag.

**Table 7.1. HP48 System Flags**

| **Flag** | **Name** | **Meaning when Set** |
|---|---|---|
| *Symbolic Mathematics* | | |
| −1 | Principal Values | "Solving" returns only principal values |
| −2 | Symbolic Constants | Symbolic constants evaluate to numbers |
| −3 | Numeric Execution | Functions return numerical results |
| *Binary Integer Math* | | |
| −5 to −10 | Binary Integer Wordsize | Encode binary integer wordsize |
| −11, −12 | Binary Integer Base | Specify base |
| *Floating Point Math* | | |
| −15, −16 | Coordinate System | Specify coordinate system |
| −17, −18 | Trigonometric Angle | Specify angle mode |

| | | |
|---|---|---|
| −19 | Complex 2D | 2D, →V2 create complex numbers |
| −20 | Underflow Exception | Underflow is an error |
| −21 | Overflow Exception | Overflow is an error |
| −22 | Infinite Result Exception | Infinite result is not an error |
| −23 | Negative Underflow | Negative underflow occurred |
| −24 | Positive Underflow | Positive underflow occurred |
| −25 | Overflow | Overflow occurred |
| −26 | Infinite Result | Infinite result occurred |
| *I/O and Plotting* | | |
| −30 | Function Plot | Equations $y = f(x)$ plot $y$ independently |
| −31 | Curve Filling | No curve filling |
| −32 | XOR Cursor | Graphic cursor XOR's with picture |
| −33 | I/O Device | I/O is directed to the IR port |
| −34 | Printer Device | Printer output directed to the serial port |
| −35 | Binary I/O | File transfer in binary mode |
| −36 | RECV Overwrite | RECV overwrites variables of same name |
| −37 | Double Space Printing | Printed text is double-spaced |
| −38 | Linefeed | Suppress auto-insertion of linefeeds |
| −39 | No Kermit Messages | Suppress display of Kermit messages |
| *Time Management* | | |
| −40 | Ticking Clock | Date and time are displayed |
| −41 | 24-Hour Clock | Times in 24-hour format |
| −42 | DMY Date Mode | Dates in DD/MM/YY format |
| −43 | Rescheduling Repeat Alarms | Unacknowledged repeat alarms not rescheduled |
| −44 | Save Acknowledged Alarms | Acknowledged alarms remain in appointment list |
| *Display Format* | | |
| −45 to −48 | Decimal Number Digits | Specify number of digits for FIX, SCI, ENG |
| −49, −50 | Decimal Number Format | Select FIX, SCI, ENG or STD |
| −51 | Fraction Mark Selection | European fraction mark choice |
| −52 | Single-Line Display | Single-line display of level 1 |
| −53 | Precedence | Display hidden parentheses in algebraics |
| *Miscellaneous* | | |
| −55 | Last Arguments | No last arguments |
| −56 | Error Beeps | No error or BEEP beeps |
| −57 | Alarm Beeps | No alarm beeps |
| −58 | Verbose Messages | Suppress prompt messages |
| −59 | Fast Catalog | Show catalog equations by name only |
| −60 | Alpha Key Action | One key alpha-lock |
| −61 | USR Key Action | One key user keyboard-lock |
| −62 | User mode | User mode active |
| −63 | Vectored ENTER | User-defined ENTER active |
| −64 | Index Wrap | GETI or PUTI index has wrapped |

## 7.2 Key Assignments

In many of its built-in environments, such as the plot environment or the the EquationWriter, the HP 48 redefines the actions of various keys to perform operations specific to the environments, and also disables other keys that have no relevance there. This same capability is available for customizing purposes, through *user key assignments*. To *assign* a key means to specify an object that is to be executed when you press that key, in place of the normal built-in key action. You can assign any key, even ATTN, including any of the six variations unshifted, left- and right-shifted, alpha-shifted, and alpha-left- and alpha-right-shifted. Key assignments that you make are active whenever the HP 48 is in *user mode*, and disabled otherwise; this makes it easy to switch between the normal keyboard and your custom keyboard.

In manual operation, you can switch the HP 48 into user mode by pressing the ◁ USR key. By default, this key is a 3-state key similar to α ; pressing it once turns on the 1USR annunciator, signaling that the action of the next key pressed will be the user key assignment of that key. The next key after that reverts to its default definition. However, if you press ◁ USR twice consecutively, the 1USR annunciator changes to USER, which indicates that user mode is locked on. All subsequent key presses will execute the user key assignments for those keys (the EquationWriter and the MatrixWriter also respect the user assignments; other built-in environments do not). User mode remains in effect until you again press ◁ USR .

If you prefer, you can disable single-key user mode by setting flag −61. In that case, a single press of ◁ USR activates user mode, much like the behavior of the USER key on the HP 41, which was the original upon which HP 48 user mode is modeled (single-key user mode was copied from the HP 71B). The state of user mode is reflected in flag −62; setting that flag turns on user mode, clearing it turns it off, and −62 FS? indicates whether the mode is active.

### 7.2.1 Single Key Assignments

To make an individual key assignment, the command ASN takes the object to be assigned to a key from level 2, and a keycode number *rc.p* from level 1:

- The digit *r* is the key row counting from 1 at the top row (menu keys), and *c* is the key column, counting from 1 at the leftmost column. The digit *p* represents the key *plane* (shift):

| **Shift** | **Plane** *p* |
|---|---|
| *none* | 0 or 1 |
| [◁] | 2 |
| [▷] | 3 |
| [α] | 4 |
| [α] [◁] | 5 |
| [α] [▷] | 6 |

Thus, for example,

```
'ABC'  34.3  ASN
```

assigns the name ABC to [▷] [EVAL] (row 3, column 4, shift 3--[▷]).

The key assignment object can be any single object, either a built-in command, an XLIB name for a library command, or any user-created object. For most of these object types, the user mode behavior of the assigned key is similar to the action of default keys: in immediate-mode, the key object is executed; in algebraic entry mode the key object is copied to the command line if it is allowed within algebraic expressions; in program entry mode, the object is copied to the command line. There are two exceptions:

- Keys assigned to string objects echo those strings to the command line, without their surrounding "" delimiters, regardless of the entry mode. This allows you to provide single-key entry for inaccessible characters or multi-character strings. For example, if you need to use the × character, you can assign it to the [α] [×] key:

```
215  CHR  75.4  ASN
```

- Keys assigned to programs are not usable in program entry mode--they just beep. This restriction is based on the assumption that keys defined by programs are meant for immediate execution, and so to echo them into the command line would more likely be a nuisance than a positive feature.

### 7.2.1.1 An Interactive Key Assignment Program

In the HP 41, ASN is an interactive operation in which you assign a command or program by spelling its name, and specify a key by pressing it. This friendly style can be imitated on the HP 48 by means of the program ASN41, listed below. Executing ASN41 prompts you to enter a key assignment object into the command line (you can press

$\boxed{\text{ATTN}}$ to cancel the new assignment), either by typing it in or by pressing a keyboard or menu key for the object. When you then press $\boxed{\text{ENTER}}$, ASN41 displays (Press a key), and waits for a key press. After the key press, the display shows the key code for one second, and the assignment is complete. If you press ENTER at the first prompt without entering any object, any current key assignment for the designated key is cleared.

| ASN41 | *ASN HP41-style* | 9CF1 |
|---|---|---|

```
«  RCLF  STD  -55  CF                   Save current modes, activate STD
                                         and argument recovery.
  "Assign: "  DUP  { V }                 Prompts for definition object.
  IFERR  INPUT                           Enter definition.
  THEN  3  DROPN                         If ATTN, then quit.
  ELSE                                   Otherwise, proceed.
    IF  DUP  ""  SAME                    If no entry,
    THEN  "(Clear)"  SWAP                  then show (Clear);
    ELSE  "{"  OVER  +  OBJ→  1  GET       else convert entry to an object.
    END
    3  ROLLD  +  3  DISP                 Show the object.
    "To: "  DUP  5  DISP
    "(Press a key)"  10  CHR  +  6  DISP Prompt for a key.
    IFERR  0  WAIT                       Wait for a key.
    THEN  DROP  91                       If ATTN, then keycode 91.
    END
    SWAP  OVER  +  5  DISP               Show the keycode.
    ""  6  DISP
    IF  OVER  ""  SAME                   If definition is null,
    THEN  DELKEYS  DROP                    clear the key definition;
    ELSE  ASN                              else make the assignment.
    END  1  WAIT                         Pause to make the display visible.
  END  STOF                              Restore old modes.
»
```

### 7.2.2 Multiple Key Assignments

In application programs, it is often desirable to assign several keys, or even the entire keyboard. You can achieve this with the command STOKEYS, which takes as input a list of object-key pairs like those used by ASN:

$$\{\, object_1 \quad rc.p_1 \quad object_2 \quad rc.p_2 \quad \cdots \quad object_n \quad rc.p_n \,\}$$

The assignments made by STOKEYS (and ASN) are cumulative; the new assignments specified in the argument list are added to those already activated by previous uses of STOKEYS and ASN.

You can recall the list of all current key-object pairs by executing RCLKEYS. Like RCLF and RCLALARMS, RCLKEYS is most useful for saving the current state of the HP 48 so that it may be restored later.

The list returned by RCLKEYS may include the name S (for *System*) at the start of the list, without any corresponding key code (making an odd number of list elements). If it is present, the S means that keys that are not otherwise assigned retain their default unassigned behavior in user mode. Similarly, if you include an S at the head of a key-object list used by STOKEYS, the default behavior of unassigned keys is restored.

Disabling unassigned keys is one of the features of DELKEYS (*DELete KEYs*). In general, DELKEYS removes the user key assignment of one key specified by a keycode *rc.p*, or of multiple keys specified by a list of keycodes. As a shortcut, 0 DELKEYS clears all current user key assignments and restores all keys' default actions. Furthermore, the name S is also accepted as an argument by DELKEYS; 'S' DELKEYS disables all keys that do *not* have user key assignments, so that they merely beep when pressed in user mode. This is useful for programs that want to halt for user input, and wish to restrict the user's choices to a few selected keys. A typical program sequence might look like this:

| | |
|---|---|
| `RCLF RCLKEYS → flags keys` | Save the current key assignments and flags. |
| `« 0 DELKEYS` | Clear current key assignments. |
| `'S' DELKEYS` | Disable unassigned keys. |
| `{ PRG1 82 PRG2 83 PRG3 84 }` | Assignments for 1, 2, and 3 keys. |
| `STOKEYS` | Make the assignments. |
| `-62 SF` | Turn on user mode. |
| `"Press 1, 2, or 3" PROMPT` | Stop and prompt for a choice. |
| `keys STOKEYS` | Restore the original assignments. |
| `flags STOF` | Restore the flags. |
| `»` | |

Executing this sequence shows the prompt Press 1, 2, or 3, inviting the user to press one of those three keys. All other keys are disabled, so that you can not do anything else that might disrupt what the program is trying to do. When you press one of the indicated keys, it executes one of the names PRG1, PRG2, or PRG3, which presumably are the names of programs. Each of those programs should terminate with CONT, to return execution to the above sequence, which finishes by restoring previous key assignments and flag settings.

For cases where you want to suppress most, but not all, default key assignments, STOKEYS accepts the name SKEY as a special object that you can assign to one or more keys. When you do so, the selected keys retain their default behavior even if you have executed 'S' DELKEYS to disable unassigned keys. For example,

```
{ SKEY 25 SKEY 34 SKEY 35 SKEY 36 } STOKEYS 'S' DELKEYS
```

disables all user mode keys except the four arrow keys.

You can paint yourself into a corner with DELKEYS: in user mode, if you execute 0 DELKEYS 'S' DELKEYS, you disable the entire keyboard--including the ⬅ USR key you need to turn off user mode. The only recourse in this situation is to execute a system halt ( ON - △ together). A system halt turns user mode off (flag −62 is the only flag affected by a system halt). Afterwards, you might want to execute { S } STOKEYS or 0 DELKEYS to prevent falling into the same trap again.

### 7.2.3 Key assignments and memory

If you use MEM to check the amount of free memory before and after you make your first key assignment, you will find that the assignment has used more than 275 bytes of memory (to be precise, 275 bytes plus the size of the assigned object). Fortunately, subsequent key assignments are not so expensive. The HP 48 stores its key assignments in a list stored in a normally inaccessible part of the home directory. When there are no assignments, the list is empty. However, upon the first execution of ASN, the HP 48 adds 49 objects to the list, each of which records the assignments for one key. With one assignment, 48 of the objects are themselves empty lists (five bytes each). The remaining object is a list that contains the assignment objects for each of the six planes of the assigned key; five of these are empty lists, and the sixth is the assignment object (30 bytes plus the assignment object size). For subsequent assignments,

- Each previously unassigned key costs 25 bytes plus the new object size, as the empty list for the key is replaced by a list containing five empty lists plus the object;
- Each assignment for a new plane of a previously assigned key replaces an empty list with the new object, requiring the object's size less five bytes.

You can assign any object to any key. However, if a large object is stored in a global variable or a port variable, it is more efficient to assign the object's variable name to a key rather than the object itself, since the assignment list then contains the name rather than the object. Keeping assignment objects small also generally maximizes the speed of key assignment execution by minimizing the size of the list that the HP 48 has to search to find an assignment.

## 7.3 Custom Menus

The VAR menu (section 5.1) is a convenient facility for displaying the names of stored objects, and providing single-key store, recall, and execution of the objects. However, once you have more than few global variables, the VAR menu becomes harder to manage, since the positions of entries in the menu changes as objects are stored and purged, and also the variables may be distributed among several directories and so harder to find.

The HP 48 *custom menu* system allows you to define one or more menus of your own devising, in which you can mix commands and other objects as well as variable names, in any order you choose. There is even a primary key [CST] that activates a custom menu, making such menus extremely convenient. Custom menus can be temporary or permanent, and you can associate one permanent custom menu with any directory.

A permanent custom menu is defined by a list of one or more objects, that is stored in the reserved-name global variable CST. The first six objects define the first page of the menu, in left-to-right order, the second six define the second page, etc., just like the VAR menu. When you press [CST], the HP 48 searches the current directory for CST; if it is not present, the search continues in the usual way up through the parent directories. CST may contain a list of objects, or it may contain the name of a global variable (in the current path) that contains a list. For example, if you execute

{ A B C } [⮣] [MODES] [⮢] ≣CST≣ ,

then press [CST] , you will see a menu containing entries for A, B, and C. The corresponding menu keys have the same behavior as VAR menu keys for those variables, including the left- and right-shifted actions. In general, the actions of shifted and unshifted CST menu keys depend on the type of the matching objects in the CST list.

The MENU command (also in the [⮣] [MODES] menu) provides an alternate way to store a custom menu list. [MENU] takes a menu list (or the name of a list) and stores it in CST in the current directory, then automatically activates the custom menu. Generally, the only time you might use 'CST' STO instead of MENU is when you want to define a custom menu for future use, but do not want to activate that menu immediately.

Like any other menu, the custom menu remains active until another menu is activated. If you change directories while the custom menu is active, the menu is updated if necessary to reflect the contents of CST in the new directory. However, storing a new menu list in CST (or purging it) does not affect the displayed menu until you press [CST] again or change directories.

It is also possible to activate a *temporary* custom menu that does not use or change the contents of CST, by using TMENU instead of MENU. The menu defined by TMENU's list or name argument persists until you change menus (⮫ LAST MENU restores it). Pressing CST reverts to the menu defined in CST, not that activated by TMENU. TMENU is most useful in programs, where you wish to prompt the user with a particular menu, then have no further use for the menu. In many cases a menu used within a program has no meaning once the program is finished, so TMENU is a better choice than MENU.

## 7.3.1 Built-in Menus

You can also use MENU and TMENU to activate a built-in menu, by supplying a real number argument for either command (there is no difference between the commands in this case). The number must be of the form *mmmm.pp*, where *mmmm* is a one- to four-digit number that specifies the menu, and *pp* is a two-digit number that specifies the menu page. For example, menu 1.01 is the first page (.01) of the custom menu (menu 1), 2.02 is the second page of the VAR menu, and 12.04 is the last page of the PRG OBJ menu. (For page 1 of any menu, you can omit the *pp* digits and just specify an integer menu number). Note that the contents of CST remain unchanged when you use TMENU or MENU with a number argument.

Except for temporary custom menus (0), the CST menu (1) and the VAR menu (2), the built-in menus are numbered in the order that they "appear" on the HP48 keyboard, starting with the MTH menu as menu 3, the MTH PARTS menu as 4, and so on across and down the keyboard and through sub-menus. The number of a library command menu (one activated by pressing ⮨ LIBRARY followed by the menu key for the library) is the same as the library number. However, the easiest way to determine a menu number is not to count menus or even to look up the number in a manual, but simply to activate the desired menu and page, then execute RCLMENU, which returns a number *mm.pp* for the current menu. Of course, you have to execute RCLMENU by typing it into the command line, or by assigning it to a user key--using the RCLMENU key in the ⮫ MODES menu always returns 21.02. (Executing MENU or TMENU with the number of a non-existent menu returns a blank menu.)

To illustrate the use of MENU for built-in menus, suppose you find yourself using PUT and GET more frequently than other PRG menu commands. Then it might be helpful to assign « 12.04 MENU » to PRG (key 22); then in user mode, pressing PRG takes you immediately to the menu containing PUT and GET.

## 7.3.2 Custom Menu Object Types

The precise action of a custom menu key depends on the type of the object corresponding to that key in the custom menu list. As we mentioned previously, if an object is a name, the custom menu key action is the same as that of a VAR menu key. For most other object types, the "execute this object" the immediate-mode meaning of an unshifted menu key is retained from the VAR menu, but the shifted keys are only active in a few cases. Furthermore, the effect of the menu keys in algebraic- and program-entry mode also depends on the type of object. Table 7.2 shows the custom menu behavior exhibited by each HP 48 object type, for all four entry modes.

**Table 7.2. Custom Menu Key Actions by Object Type**

| Key-Object Type | Entry Mode | Unshifted Action | Left-Shifted Action | Right-Shifted Action |
|---|---|---|---|---|
| Name | Immed. | Enter the name | 'name' STO | 'name' RCL |
| | ALG | Echo the name | 'name' STO | 'name' RCL |
| | PRG | Echo the name | None | None |
| | ALG PRG | Echo the name | None | None |
| Port Name | Immed. | 'name' RCL EVAL | 'name' STO | 'name' RCL |
| | ALG | 'name' RCL EVAL | 'name' STO | 'name' RCL |
| | PRG | Echo the name | None | None |
| | ALG PRG | Echo the name | None | None |
| Number | Immed. | Enter number | None | None |
| | ALG | Echo with no spaces | None | None |
| | PRG | Echo with spaces | None | None |
| | ALG PRG | Echo with no spaces | None | None |
| String | any | Echo string, no quotes | None | None |
| Unit | Immed. | Enter unit, multiply | Enter unit, CONVERT | Enter unit, divide |
| | ALG | Echo unit part | Enter unit, CONVERT | Enter unit, CONVERT |
| | PRG | Echo unit part | None | None |
| | ALG PRG | Echo unit part | None | None |
| Algebraic | Immed. | Enter object | None | None |
| | ALG | Echo object, no ' ' | None | None |
| | PRG | Echo object with ' ' | None | None |
| | ALG PRG | Echo object, no ' ' | None | None |
| Program | Immed. | Enter program | None | None |
| | ALG | Enter program | None | None |
| | PRG | None | None | None |
| | ALG PRG | None | None | None |
| RPN Command | Immed. | Enter command | None | None |
| | ALG | Enter command | None | None |
| | PRG | Echo, no spaces | None | None |
| | ALG PRG | Echo with spaces | None | None |

| | | | | |
|---|---|---|---|---|
| Function | Immed. | Enter function | None | None |
| | ALG | Echo, with alg. syntax | None | None |
| | PRG | Echo with spaces | None | None |
| | ALG PRG | Echo, with alg. syntax | None | None |
| List | | *(See section 7.3.3)* | | |
| Other | Immed. | Enter object | None | None |
| | ALG | Enter object | None | None |
| | PRG | Echo object with spaces | None | None |
| | ALG PRG | Echo object with spaces | None | None |

- "Enter object" means perform ENTER, then execute the object.
- "Echo object" means copy the object to the command line.
- "Alg. syntax" means appending parentheses where appropriate, and surrounding with spaces if the function is a multi-character infix operator like MOD or XOR.
- The actions associated with built-in RPN commands and functions also apply to XLIB names, according to whether a name refers to a library command or to a function.
- The actions described for port names also apply when the name has the extended form *:tag:*{ *list* } (section 5.5.3). Note, however, that the left-shifted store action fails if the corresponding port variable already exists (section 5.3.1.2).

Some points worth noting from Table 7.2:

- The menu key for a string echoes the string to the command line *without* quote delimiters, which enables you to define *typing aids*--keys that echo a character sequence that you use frequently, or perhaps a special character that is unavailable or inconvenient on the alpha keyboard.
- The menu keys for unit objects work just like the keys in the various ⬑ UNITS menus. This is very useful for creating units menus that combine units from different built-in menus or pages plus units that you have defined yourself.
- A program is the only object type that is not echoed to the command line when an assigned key is pressed in algebraic or program entry mode. Generally, program assignments are meant for immediate execution, so this is not a very important limitation.

By default, a custom menu key label is derived from the associated menu list object, showing the first few (up to five) characters from the display form of the object. Especially for extended objects like programs, where you can only see the leading ≪ and one or two characters from the first object in the program, such labels may not be too helpful, since you can't see enough of the object to recognize it. The HP 48 solves this problem by allowing you to define labels that are independent of the objects that define the menu key actions.

### 7.3.3 Menu Key Labels and Shifted Menu Key Actions

If one of the objects in the custom menu key list is itself a list, the contents of that list are used to create an extended form of menu key definition that permits specification of the menu key label, and assignment of one or both shifted key actions. Usually the (inner) list contains two objects:

{ *label-object   action-object* }

- The first object in the list, normally a string or name with up to 5 characters, is used to form the menu key label. If the object is other than a string, a name, or a 21×8 graphics object, the label text will include the leading object delimiter, if any. If the object is a 21×8 graphics object, the label becomes an icon defined by the graphics object.
- The second object in the list defines the key actions, following the rules listed in Table 7.2. (If the second object is absent, there will be a menu key with a label defined by the first object, but it just beeps when pressed.) One more level of extension is available: if the second object is itself a list, it may contain one, two or three objects, so that the most general custom menu list object looks like this:

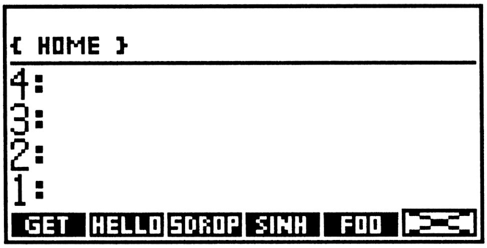
  { *label* { *no-shift   left-shift   right-shift* } }.

  The three objects in the inner list define the unshifted, left-shifted, and right-shifted key actions for the menu key (which is labeled by the *label* object), following the rules for *unshifted* actions listed in Table 7.2. Note that this applies to algebraic and program entry mode (section 6.4.1) as well as immediate entry mode--the key action object determines what text is echoed to the command line (and whether parentheses are included), not the label object.

By way of example, consider a custom menu defined by the following list:

```
{ GET
  "HELLO"
  { "5DROP"  << 5  DROPN >> }
  { SINH  { SINH  ASINH } }
  { FOO  { << { HOME  UTIL  FOO }  RCL  EVAL >>
           << PATH  HOME  UTIL  SWAP  'FOO'  STO  EVAL >>
           << { HOME  UTIL  FOO }  RCL >>
         }
  }
  { GROB 21 8 FFFFF1D100711E0E0110F10110F1011E0E01D10071FFFFF1   KILL }
}
```

Executing MENU with this list yields a menu that looks like this:



The individual menu keys are defined by the menu list as follows:

- The first key GET illustrates the simple assignment of a command to a key, with no shifted actions.
- The section key HELLO comes from the string "HELLO". This is a "typing aid"; the unshifted key echoes HELLO to the command line without delimiters or surrounding spaces. The shifted key has no action.
- The next key 5DROP illustrates labeling a menu key with a string while the key action is defined by a program.
- The SINH key has both an unshifted action (SINH) and a left-shifted action (ASINH).
- FOO has actions defined for both the left- and right-shifted key as well as the unshifted key. It is designed to act like a VAR menu key for the variable FOO in the HOME UTIL directory, that will work regardless of the current directory.
- The next key is labeled by the 21×8 graphics object, and executes or echoes KILL when pressed.

You can create the graphics object by executing the following sequence:

```
ERASE  PICT  { #0  #0 }  { #20d  #7d }  DUP2  DUP2
LINE  BOX  { #0  #7 }  { #20d  #0 }  LINE  SUB
```

## 7.4 Vectored ENTER

The normal process associated with ENTER is described in section 6.4.3. As mentioned there, however, you can modify that process by means of an HP48 feature called *vectored* ENTER (the name comes from computer science jargon, referring to the fact that the system looks for a vector--a pointer to a replacement procedure--before executing a standard procedure). This feature gives you a powerful customization capability, since it allows you to redefine the way command line text is interpreted, and a chance to execute additional commands after command line entry and execution are completed.

Three conditions must be met to activate vectored ENTER:

1. At least one of the variables αENTER and βENTER must exist, in the current path.
2. Flag −63 must be set. The use of a flag prevents the HP48 from searching for the special variables when the flag is clear, thus speeding up the ordinary ENTER process.
3. Flag −62 must be set. This is the user mode flag; including this flag as part of the vectored ENTER setup gives you a convenient keyboard means (⬅ USR ) with which to turn vectored ENTER on and off.

When the two flags are set, the HP48 searches for the variable αENTER before parsing the command line in the usual way (step 2 in section 6.4.3). If the variable exists, the command line text is not parsed but is just entered into stack level 1 as a string object, following which then αENTER is executed. Since this execution replaces normal command line parsing and execution, you can store in αENTER a program that interprets and uses the command line text in any manner you please. Furthermore, since OBJ→ "executes" a string object as if its text were entered in the command line, you can define αENTER as merely a preprocessor that modifies the command line text and then uses OBJ→ to continue with normal processing. This technique is used in the binary calculator program BINCALC described below, to save you from having to type a # when you enter a binary integer.

After αENTER is finished, the object assigned to the key that started the ENTER process is executed. Then, after its execution is complete, the HP48 searches the current path for a variable βENTER. If that variable exists, a string representing the key object is entered into level 1 and βENTER is executed. In general, βENTER is intended to contain a program that performs some operation on the result of a command line entry; the key object string is made available for record keeping purposes.

A straightforward example of the use of vectored ENTER is to create a simple calculation-tracing mode using a printer. Store the following program in αENTER:

```
<< PR1  OBJ→ >>
```

This routine copies the command line contents to the printer, then uses OBJ→ to do normal command line processing. You also need this program for βENTER:

```
<< "[" SWAP + "]" + PR1 DROP PR1 >>
```

The βENTER program surrounds the key object string with brackets [ ], then prints it, followed by the level 1 result of the entire execution. This example reveals one limitation of the βENTER process: only keys that correspond to programmable, named objects--commands, XLIB names, global names, and local names--return a meaningful string for βENTER. For other object types, plus unnamed built-in objects such as ENTER itself, only an empty string is returned. For these cases, the above βENTER program prints empty brackets [ ].

## 7.4.1 Examples

The vectored ENTER system along with the other HP 48 customization facilities enable you to tailor the HP 48 into many different specialized calculators. In this section, we will give two examples, one that focuses the HP 48 on binary arithmetic calculations, and another that turns the HP 48 into a "fraction calculator."

| BINCALC | *Binary Integer Calculator* 1F6D |
|---|---|
| `<<` | αENTER program: |
| `<< IF DUP "" ≠` | If there is command line text, |
| `THEN "#" SWAP + OBJ→` | prepend #, then execute. |
| `ELSE DROP` | |
| `END` | |
| `" Binary Calculator" 10 CHR +` | Show a message. |
| `1 DISP 1 FREEZE` | |
| `>> "" OVER EVAL 'αENTER' STO` | Show the message and store the program. |
| `RCLKEYS RCLF RCLMENU` | |
| `→ keys flags smenu` | Save the key assignments, flags, menu. |
| `<< 0 DELKEYS` | Remove current key assignments. |
| `{ "A" 41 "B" 42 "C" 43` | |
| `"D" 44 "E" 45 "F" 46` | |
| `} STOKEYS` | Assign hexadecimal letters to row 4. |
| `-63 SF -62` | Activate vectored ENTER. |
| `SF 9.01 TMENU` | Turn on the binary menu. |
| `HALT` | Halt for binary calculations. |
| `flags STOF` | Restore flags. |
| `0 DELKEYS keys STOKEYS` | Restore key assignments. |
| `'αENTER' PURGE` | Discard αENTER. |
| `smenu MENU` | Restore the original menu. |
| `>>` | |
| `>>` | |

Executing BINCALC displays the message Binary Calculator, and activates an environment in which it is assumed that all command line entries are to be binary integer objects, one per command line. The keyboard is redefined so that the fourth-row keys echo the hexadecimal digits A - F, to supplement the ordinary number pad for hexadecimal entry. The program uses those keys rather than the menu keys, in order to leave the latter available for other menus, especially for the base operations menu ( [MTH] ≡BASE≡ ) menu. As long as the environment is active, you can perform RPN arithmetic and other operations on binary integers, entering the integers *without* the # delimiter. You can temporarily disable the special environment with the [⬅] [USR] key, and reenable it with the same key. Finally, when you want to resume normal operations, press [⬅] [CONT]. This restores the key assignments, flags, and menu that were present when you executed BINCALC, and reverts to the standard environment.

BINCALC's demands on vectored ENTER are modest. In the next example, FRACALC, the program takes over command line interpretation entirely. FRACALC executes similarly to BINCALC: an environment is established in which command line entries are assumed to be fractional numbers. You enter numbers in the form *i n.d*, where *i* is the integer part, *n* (separated from *i* by a space) is the numerator, and *d* is the denominator. *n* and *d* may be separated by a period or a comma. Examples:

```
1 2.3     ☞    '1+2/3'
-4 5.6    ☞   '-4-5/6'
8.12      ☞      '2/3'
-1,2      ☞     '-1/2'
```

You can apply immediate-execute commands to the stack fractions, and their results will also be fractions:

```
1 1.2 [ENTER] 3 2.3 [+] ☞ '5+1/6'
```

You can disable fraction entry by turning off user mode. Press [⬅] [CONT] to terminate the fraction environment entirely.

FRACALC uses →Q to convert decimal numbers to fractions, with 5 decimal places of accuracy. You can change the 5 FIX in the program to another value to change this accuracy when you want to deal with denominators larger than three or four digits. However, too large a value may cause unexpected fractions to be returned for some small denominators.

FRACALC *Fraction Calculator* C01B

```
«
 «                                      Start of αENTER procedure.
   IF DUP SIZE DUP                      If there is command line text, parse it.
   THEN OVER " " POS                    End of integer part, if any
     3 PICK DUP "." POS                 Find a ".", or
     SWAP "," POS MAX                   find a ",".
     DEPTH 4 - → cmd len int div stk    Store parameters and stack depth.
     « IFERR                            Error trap for invalid entries.
         IF div                         If there is a fraction,
         THEN cmd int div 1 - SUB OBJ→  separate out the numerator,
           cmd div 1 + len SUB OBJ→ /   and divide by the denominator.
           IF int                       If there is also an integer part,
           THEN cmd 1 int SUB OBJ→      get the number.
           DUP SIGN ROT * +             Add the fraction with the same sign.
           END
         ELSE cmd OBJ→                  No fraction, so assume integer entry.
         END
       THEN                             Error handler:
         IF DEPTH stk - DUP 0 >         Discard extra stack objects.
         THEN DROPN
         END cmd "Invalid Entry" DOERR  Report the error
       END
     »
   ELSE DROP2                           Discard empty command string and size.
   END " Fraction Calculator" 10 CHR +
   1 DISP 1 FREEZE
 » "" OVER EVAL 'αENTER' STO            Show the message and store the program.
 « DROP
   IF DEPTH                             Do nothing if stack is empty.
   THEN
     IF DUP TYPE 9 OVER SAME            If the last entry is an algebraic,
       SWAP NOT OR                      or a real number,
     THEN →NUM                          then convert to a fraction:
       DUP IP SWAP FP                   Get the integer and fractional parts.
       10 RND 5 FIX →Q STD +            Convert to a symbolic fraction.
       END
   END
 » 'βENTER' STO
 RCLF → flags                           Save the current flags.
 « -62 SF -63 SF -51 CF STD HALT        Halt for binary calculations.
   flags STOF                           Restore original flags.
 »
 'αENTER' PURGE
 'βENTER' PURGE
»
```

# 8. Problem Solving

The rich command set of the HP 48 allows you to solve many problems merely by pressing a few keys. However, where the HP 48 really excels is in the ease with which you can link command sequences together into procedures. This allows you to solve complex problems by breaking them down into simple pieces. Once a procedure corresponding to a problem's solution is developed and stored, you can execute it any number of times while you vary the input data.

The term *programming* is conventionally used for the process of recording a sequence of calculator instructions in such a manner that you can later replay the sequence any number of times without having to reenter the instructions. Here, we will use the more general term *problem solving* to describe the various HP 48 solution strategies, of which programming--creating program objects--is just one of several.

A problem solution generally consists of three parts:

1. Data input;
2. Data processing and calculations;
3. Results output.

Each of these stages can be simple or complicated. To enter data, for example, you can use a program that just takes one or more objects from the stack which are presumed to be there when the program is executed. Or, your program can prompt for each required value by halting with a text display that asks you for a specific input. Similarly, a program can return its results to the stack, or it can display each result with an identifying text label.

Regardless of the complexity of a calculation, in most calculators the only method of automating calculations is to create a program, complete with labels and line numbers. While this restriction has the virtue of simplicity in that there are no alternatives, the process can be cumbersome for simple procedures, particularly for straightforward mathematical expression evaluation. The HP 48 provides a series of problem solving alternatives, ranging from simple expression evaluation to programs with loops, branches, recursion, etc. Problem solving can be both simpler and more complicated than in other calculators. In general, it is easier to program any given calculation on the HP 48; additional complication only arises really when you are dealing with problems that are not soluble at all on other calculators.

The HP 48 problem-solving alternatives sort roughly into four approaches:

- HP Solve;
- User-defined functions;
- Symbolic manipulations;
- General programming.

These are listed roughly in order of increasing complexity; not so much in the complexity of the mathematics involved but rather in the amount of mental effort you need to translate a real problem into HP48 terms. The classification is somewhat imprecise because there's a great deal of overlap, such as programs that contain user-defined functions; HP Solve exercises that use programs; even algebraic objects that execute programs. With all of these options, your challenge is to determine which approach is most appropriate for a particular problem.

In the remainder of this chapter, we will show which types of problems are suitable for each general problem solving method, then consider user-defined functions as an initial exercise in HP48 problem solving. HP Solve and symbolic algebra are left for detailed study in *Part II*. The remaining chapters of *Part I* are devoted to various programming tools and methods.

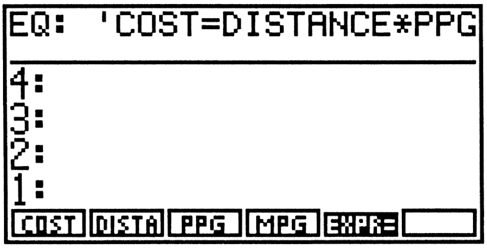
## 8.1 HP Solve

*HP Solve*, which is essentially a combination expression-evaluator and root-finder, provides an exceptionally easy method of problem solving on the HP48. It is suitable for any problem that can be reduced to a single equation relating all of the variables in the problem, and for which a real-valued numerical answer is sufficient. The greatest benefit of HP Solve is that you don't have to solve the equation formally for the unknown--all you have to do is enter any equation that relates the unknown to the known variables. Furthermore, you can interchange the roles of known and unknown variables as you go along, without doing any additional work to restate the problem.

A prototype problem ideal for the solver is the simple "cost-of-travel" equation:

$$\text{COST} = \text{DISTANCE} \times \text{PPG} / \text{MPG},$$

where PPG stands for "price per gallon," and MPG stands for "miles-per-gallon." This single equation relates all the relevant parameters, and has the virtue of containing only simple arithmetic operations, so that there is only one possible solution for any choice of values for any three of the variables. To address this problem with HP Solve, all you have to do is enter the equation in algebraic form as written above, press ◁ SOLVE STEQ to select it as the current equation, then press SOLVR . The calculator presents you with the *solve variables menu*, which provides a menu key for each of the four variables:


```
EQ:  'COST=DISTANCE*PPG
4:
3:
2:
1:
COST DISTA PPG  MPG  EXPR=
```

You can use the menu to store values in any three of the variables and solve for the fourth.

Contrast this simplicity with the process you have to follow on other calculators without HP Solve. For each choice of unknown variable, you have to

a. Solve the equation formally (on paper) for the unknown;

b. Translate the solved equation to program form;

c. Add input prompting steps to the start of the program, and output labeling to the end.

d. Enter the program using the calculator's program editor;

e. Run the program for each new set of input parameters.

If you're very clever, you can figure out how to combine the four separate programs into one, where the program figures out from the inputs which variable is to be calculated and thus which branch of the program to use--in other words, to duplicate what the HP Solve does for you automatically.

## 8.2 Symbolic Manipulations

The HP 48 and its predecessor the HP 28 are unique among calculators in their ability to apply mathematical operations to "symbolic" quantities--objects for which no numerical value has been assigned. If you're a student learning algebra or calculus, or using their techniques in other mathematical or scientific studies, this capability may be very exciting. However, if you're not directly interested in algebra for its own sake, you might wonder why these symbolic capabilities are important to you.

Actually, if you use a programmable calculator at all for more than simple keyboard arithmetic, you are already performing a kind of symbolic operation. Any time you perform a calculation more than once, using varying data, you probably represent the calculation symbolically at some point. In particular, when you write a program to automate the calculation, that program is a symbolic operation. You write it to accept certain inputs, without specifying their values, and to compute an unknown result. This is no different in principle from writing an algebraic expression on paper. An expression also "works" with unspecified inputs (variables) and returns a previously unknown value when you evaluate it.

So in the sense that any program is a symbolic calculation, any programmable calculator is a "symbolic" machine. The important contribution of the HP 48's symbolic capabilities is that they allow you to apply mathematical operations to the programs themselves, and obtain new programs as results. For example, consider a program that recalls the value of a variable and doubles it. In a conventional language like BASIC, the program is

```
100 Y=2*X
200 END
```

But suppose that after entering the program you realize that you are really interested in the sine of the result, sin(2*x*). You have no choice except to rewrite the program, in this case, editing line 100, being sure to enter the SIN in the right place and to include the parentheses.

On the HP 48, the original "program" consists of the algebraic object '2*X'. To change this into the new program 'SIN(2*X)', all you have to do is execute SIN when the original expression is in level 1. The parentheses are automatically inserted. In effect, the calculator writes a new program for you--all you have to do is use the same keystrokes on the symbolic "program" as you would use with a numerical quantity.

Another way to see the value of the HP 48 capabilities is to consider a general problem-solving process that consists of these steps:

1. Identify the problem.
2. Determine the known and unknown quantities.
3. Figure out the mathematical relationships between the quantities.
4. Solve the relationships for the unknowns in terms of the knowns.
5. For each set of known quantities, evaluate the solved relationships to obtain numerical values for the unknowns.

When you use a conventional calculator, the calculator can only enter the process at the final stage. Once you have equations for the unknowns, you can program those equations into the calculator, enter numerical values for the known variables, and run the programs to return the numerical values for the unknowns. The HP 48, on the other hand, can enter the process as early as step 2. You can use its symbolic capabilities to work out the relationships and solve for the unknowns--steps for which you would need pencil and paper using another calculator. The symbolic solution that you find with the HP 48 is also the "program" you can use for repeated evaluation of the unknowns with different inputs. Even if the equations you derive can not be solved symbolically for the unknowns, you can still use the Solver to obtain numerical results, without any further programming.

As an example of this process, consider the classic introductory calculus problem:

*A farmer has 100 yards of fencing to enclose a rectangular field, which is bounded on one side by a river. What length (L) and width (W) of the field gives the maximum area?*

■ *Solution*:

| **Steps** | **Keystrokes** | **Results** |
|---|---|---|
| 1. The length of the fence is 100 yards. | 'L+2*W=100_yd' [ENTER] | 'L+2*W=100_yd' |
| 2. Solve for L. | 'L' [◁] [ALGEBRA] ≡ISOL≡ | 'L=100_yd−2*W' |
| 3. Assign this value to L. | [◁] [DEF] | |
| 4. The area of the field is L times W. | 'AREA=L*W' [ENTER] | 'AREA=L*W' |
| 5. Substitute for L. | [EVAL] | 'AREA=(100_yd−2*W)*W' |
| 6. To find the maximum area, differentiate the expression. | 'W' [ENTER] [▷] [∂] | 0='−(2*W)+(100_yd−2*W)' |
| 7. Collect terms. | ≡COLCT≡ | '0=100_yd−4*W' |
| 8. Solve for W. | 'W' ≡ISOL≡ | 'W=25_yd' |

9. Assign this value to W and evaluate L. [⬏] [DEF] L [EVAL] 50_yd

Answer: The width of the field should be 25 yards, and the length 50 yards.

You can use the HP48 to formulate and solve the entire problem, incorporating the physical units directly into the expressions. With a conventional calculator, all you can do is evaluate the final numerical answer, once you have worked it out on paper; the unit conversions have to be done separately.

As another example, in section 12.11.3 we list a program SIMEQ that solves a set of simultaneous linear equations. Many other calculators provide this capability either through built-in commands or as program applications. However, without exception (including the HP48's own built-in method using matrices and vectors), these require you to enter the coefficients and constants rather than the equations themselves. In other words, you must to do the work yourself of inspecting the equations, collecting terms and rearranging if necessary, to determine the coefficients and constants. The SIMEQ program lets you enter the equations in any order, and without having to structure the individual equations in any particular way. It is the HP48's ability to deal with expressions and equations as data to be manipulated--as symbolic objects--that makes it possible for you to write a program like SIMEQ in a straightforward, compact manner. In other calculator languages, writing a program like SIMEQ would require considerable ingenuity, and would likely end up being harder to use than the usual method of entering coefficients in order.

HP48 algebraic objects (section 3.5.2) are procedures that are internally the same as programs. This means that creating any algebraic object is equivalent to writing a program. The program's "inputs" are the values stored in the variables named within the algebraic object; its "output" is the symbolic or numeric result that is returned to the stack. The beauty of an algebraic object as a program is that you can treat it as a symbolic quantity, to which you can apply additional mathematical operations, obtaining new algebraic objects--programs--automatically.

The best time to use algebraic objects as programs is when you have already defined a set of user variables, and wish to make calculations using their values. You can, of course, use the values directly by evaluating the variables as you go and using RPN commands and functions to combine the values. But if a calculation is defined in algebraic terms, you'll do better to enter the appropriate formula as an algebraic object, so that you can verify its definition before substituting specific values.

For example, to add the values of variables A and B, you can press [A] [B] [+]. Or you can type 'A+B' [EVAL]. The advantage of the latter is that you can see the entire calculation symbolically before making numerical substitutions. This advantage becomes more important as the complexity of a calculation increases. You are also relieved of the necessity for translating the calculation into RPN logic.

## 8.3 Programs

For problems for which the simplified problem solving methods that the HP 48 provides are not adequate, your final option is to write a program. There is a wide range of problems that don't fit the requirements for using other methods, including many that are mathematically very simple. For example, the three "simple" methods have the common limitation of being able to return only one result at a time. If you want to automate a process as trivial as returning the square and the cube of an argument, you must write a program. Here are three HP 48 programs that make those calculations:

```
<< DUP  SQ  SWAP  3  ^ >>

<< → x << x SQ x 3 ^ >> >>

<< → x << 'x^2' EVAL 'x^3' EVAL >> >>
```

The last two versions illustrate that you don't have to give up the advantages of the alternate problem solving methods when you create program objects; you just incorporate them into your programs. Even HP Solve's root-finding capabilities can be programmed, via the ROOT command.

The HP 48 is unusual among calculators in that it has no "program mode." In other calculators, you create a program by activating a mode where the keystrokes you press are recorded sequentially as program steps or lines. A consecutive sequence of such steps constitutes a program. To execute the program, you must leave program mode and invoke the program by means of a command like RUN or XEQ (*execute*).

Programming the HP 48 differs from manual calculating only in that you don't execute sequences of objects individually, but instead combine them into procedure objects--programs or algebraics--for later execution. You treat the procedure objects the same as any other objects: you enter and identify them by characteristic delimiters (<< >> or ' '), and you can edit, visit, store, recall, evaluate, and purge them, or just move the objects around on the stack using standard commands.

Many BASIC language computers share with the HP 48 the property of lacking a special program mode. By placing a line number at the beginning of a command line, you tell the computer to include the program line in the current program. However, that style of program entry is very context-dependent: you must be sure that the line number you assign is appropriate. It must be in the proper sequence relative to other lines, and you must have somehow established that you are adding the line to the right program. Some computers solve that problem by only holding one program in memory at a time; others permit multiple programs but you must use various means to select a particular program for editing.

Other calculator programming also uses more "program-only" concepts, like GTO (Go To), labels, line numbers, RTN (return), and commands that behave differently when used in a program than when they are executed from the keyboard. An example of the latter is the HP 41 command FS? (flag set?). From the keyboard, this command returns a temporary display of YES or NO; when executed in program, FS? acts as a "skip-if-false" operation, where the next program line is executed if the flag is set, and skipped if it is clear.

These concepts are part of what can make programming a calculator a mysterious art for many people. When you are solving a problem mentally, or with pencil and paper, you don't consider line numbers, GTO's, program modes, etc. Instead, you think in terms of a series of operations that you apply to data or symbols, which produces results that may in turn be the input for additional operations. This translates nicely to key-per-function manual use of an RPN calculator; the operations become keystrokes, and the data is kept in front of you on the stack. "Keystroke programming" on calculators originated as a process of preserving a series of keystrokes as a program. Unfortunately, as calculators became more powerful, their programming languages required you more and more to rethink a problem in order to cast it as a program.

The HP 48 is designed to minimize or eliminate the differences between interactive keystroke operations and programming. It does this in several ways:

- The command line is a program that is executed immediately; a program is a command line for which execution is deferred.
- Anything you can do in program you can do in the command line, including halting, single stepping, using local variables, branches, loops, etc.
- Commands work the same way in programs as they do when executed manually.
- Programs contain no constructs that are artificial from the standpoint of the problem being solved--no line numbers, no labels, no GTO's. The only things that appear in a program are objects and commands relevant to the calculation being performed, plus certain program structures (conditionals, loops, etc.), that are local to a particular program.

The absence of GTO's and the corresponding labels and line numbers is a manifestation of the HP 48's insistence on *structured* programming (section 9.1.3). Every program is a self-contained module, with a single "entry" and a single "exit". A program can, of course, "call" (execute by name) other programs, but only as subroutines that always return to the same point in the same program that called them. These rules promote a programming style whereby you break down a large programming task into smaller programs which are easily written and understood. As you write each "building block" program, you can test it independently before it is included in any larger program.

## 8.4 Summary

**Table 8.1. HP 48 Problem Solving Methods**

| *Method* | *Type of Problem* | *Advantages* |
|---|---|---|
| User-defined Functions | • Automatic conversion of function formulae to programs by DEFINE.<br>• Evaluation of algebraic functions, with arguments taken from the stack.<br>• Creation of new symbolic functions. | • Can be used in RPN or algebraic calculations.<br>• Does not require "permanent" user variables. |
| HP Solve | • Numerical evaluation of an algebraic expression for many values of its variables.<br>• Symbolic substitution for variables.<br>• Numerical solution of an algebraic expression, especially in combination with DRAW.<br>• "What if" problems where the independent/dependent roles of variables are interchanged. | • Automatic input prompting and labeling; automatic numerical equation solving.<br>• Lets you interchange known and unknown variables. |
| Symbolic Math | • Algebraic calculations using existing user variables.<br>• Symbolic manipulations. | • Symbolic results can be used as new programs.<br>• Calculations can be verified before they are performed. |
| Programs | All problems, especially those for which the other methods are insufficient:<br>• Multiple results.<br>• Non-mathematical problems.<br>• Special prompting or labeling.<br>• Iteration.<br>• Complicated decisions and branching. | All calculator resources are available, including the algebraic evaluation features of the other programming methods. |

## 8.5 User-Defined Functions

The archetype of a small HP 48 program is one that takes a few arguments from the stack, combines them according to some mathematical expression, and returns the computed result to the stack. For example, the distance between two points $(x_1,y_1)$ and $(x_2,y_2)$ is given by

$$[(x_2-x_1)^2 + (y_2-y_1)^2]^{1/2}.$$

This program takes the coordinates of two points from the stack, and returns the distance between the two points:

```
<< ROT - SQ 3 ROLLD - SQ + √ >>.
```

The program assumes that $x_1$, $y_1$, $x_2$ and $y_2$ have been entered onto the stack, in that order ($x_1$ in level 4). It removes the four values, and returns the computed distance to level 1.

This program is short and efficient, because you (the programmer) did the work of translating the mathematics into the HP 48's RPN logic. But writing a program this way has two shortcomings:

1. When you develop the program, you have to keep track of the stack positions of the various arguments as they are needed by the successive program commands.
2. After the program is written, it is difficult to decipher. Notice that the program objects together bear little obvious resemblance to the original distance formula.

These problems become more severe as the number of arguments and the complexity of the calculation increase. Imagine trying to alter the example program so that it works with 3-dimensional points $(x,y,z)$. Because the stack positions of all of the arguments are changed, you have to rethink all of the stack manipulations, and almost rewrite the program entirely.

The difficulty of managing stack objects is substantially reduced if your program stores the objects in named global variables, then recalls the values by name as they are needed. However, there are disadvantages to using global variables for temporary storage in a program:

- You have to choose variable names that don't conflict with those of other programs.
- The program has to purge the variables at the end to avoid leaving unneeded variables in the USER menu.

The problem of program legibility is reduced if you represent the calculations by algebraic objects. Despite the virtues of RPN for interactive calculations, by and large people are more adept at reading calculations in a form approximating conventional mathematical notation than in RPN form. With this in mind, the HP 48 provides a very simple method for creating programs that can be represented as mathematical functions, using DEFINE. For example, to create a program for the distance formula, all you need to enter is:

```
'DIST(x1,y1,x2,y2)=√(SQ(x2-x1)+SQ(y2-y1))'  [◁] [DEF]
```

If you now look in the VAR menu, you will see a variable DIST, which you can use like this:

```
1  3  4  7  DIST  ☞  5
```

If you recall the contents of DIST, you will see that DEFINE has actually stored the following program:

```
<< →  x1  y1  x2  y2  '√(SQ(x2-x1)+SQ(y2-y1))' >>
```

This program exhibits the form of a *user-defined function*, which is a program with a particular structure stored in a global variable. User-defined functions are designed to provide a simple means of programming without the problems discussed above. Specifically, they are commonly defined by algebraic expressions for easy development and modification, and they employ *local variables*, which exist only as long as the functions are executing. The local variables are used to provide names for stack arguments, and to minimize the need to manipulate lots of objects on the stack. User-defined functions are called *functions* because they act like built-in functions: you can use them like RPN commands to compute from explicit stack arguments, or as prefix functions within algebraic objects, taking arguments from within parentheses.

Looking at the example DIST, the first part of the program (→ x1 y1 x2 y2) takes four numbers from the stack and names them x1, y1, x2, and y2, by storing them in local variables with those names. The algebraic object that makes up the rest of the program computes the distance from the four stored values. You can easily modify this program for three dimensions: edit the program to add two more local names, and add a term for $(z_2 - z_1)^2$ to the algebraic expression:

```
<< →  x1  y1  z1  x2  y2  z2  '√(SQ(x2-x1)+SQ(y2-y1)+SQ(z2-z1))' >>
```

### 8.5.1 User-Defined Function Structure

In general, to create a user-defined function you store in a global variable a program object with the following structure:

$$\ll \rightarrow \ x_1 \ \ x_2 \ \cdots \ x_n \ \ 'f(x_1,x_2, \ \cdots \ ,x_n)' \ \ \gg.$$

The variable's name subsequently acts as a user-defined function. Let's look at the separate pieces of the general form, using the example DIST for illustration.

1. The first entry in the program is the symbol →. This symbol can be translated as "take arguments from the stack, and assign them the following names..." The → is always followed by a sequence of local names. The end of the sequence of names is indicated by the start of an algebraic object that must follow the names. → takes one object from the stack for each name in the sequence. In DIST, there are four names, x1, y1, x2, and y2, so DIST requires four input arguments. The objects that → takes from the stack are matched up with the names in the order in which they are entered. The first object entered onto the stack, which was in the highest numbered stack level (level 4 in DIST), is matched with the first name (x1) in the sequence.
2. The names $x_1 \ x_2 \ \cdots \ x_n$ in the series are *local* names. The combination of a local name and an object taken from the stack is called a *local* variable. Local names and variables are described in detail in section 9.7; for now, the important thing to know is that the variables exist only as long as the procedure that follows the local name list is executing. Local variables are stored in special areas of memory separate from the global variable memory; they don't appear in the VAR menu.
3. The final part of the user-defined function structure is the algebraic expression $'f(x_1,x_2, \ \cdots \ ,x_n)'$. This expression is called the *defining expression*, and constitutes the mathematical definition of the function. In the example, the defining expression is '√(SQ(x2−x1)+SQ(y2−y1))'. Within the definition of this algebraic, you can use the local names as many times as you want, just as you would global names.

When you execute the name of a global variable containing a user-defined function, the stored program is executed as follows:

1. Objects are removed from the stack and stored in local variables, one object for each variable name.
2. The defining expression in the user-defined function is evaluated.
3. The local variables are purged.

To illustrate the function behavior of a user-defined function, consider a user-defined function SEC that returns the secant of a number:

```
« → x 'INV(COS(x))' » 'SEC' STO.
```

You can execute SEC

- as an RPN command, e.g.

```
DEG 60 SEC ☞ 2.
```

- as an algebraic function, e.g.

```
'SEC(60)' EVAL ☞ 2.
```

Some other results:

| | | | |
|---|---|---|---|
| 'X' SEC | ☞ | 'INV(COS((X))' | *Symbolic arguments allowed.* |
| RAD 'SEC(X)' 'X' ∂ | ☞ | 'SIN(X)/COS(X)^2' | *Differentiation works.* |
| 'SEC(X)=Y' 'X' ISOL | ☞ | Unable to Isolate | *Error!* |

The last example shows that there is one important respect in which user-defined functions differ from built-in analytic functions. There is no inverse automatically defined for a user-defined function, so ISOL can not solve for a name that is contained in the argument of the function.

One minor note: If the HP 48 is in algebraic entry mode (section 6.4.1), pressing the VAR menu key corresponding to a user-defined function appends the function name to the command line, but does not add trailing (). Similarly, the EquationWriter does not automatically add parentheses.

### 8.5.2 User-defined Functions as Mathematical Functions

It is interesting to note the extent to which a HP 48 user-defined function is a realization of a mathematical function. That is, when you define a function such as $F(x) = 5x^2 + 2x$, you are stating that $F$ is an operator that takes a single argument, and returns a single result that is computed from the argument. The function's definition has three parts:

1. The name $F$ of the function.

2. A name $x$ used to identify the function's argument. For the purpose of the definition, $x$ does not have a value.
3. The expression in $x$ that indicates how the result is computed from $x$.

When the function is applied to a specific argument, that argument is substituted for the name $x$ in the defining expression, and the expression is evaluated. Thus

$$F(1) = 5{\cdot}1^2 + 2{\cdot}1 = 7$$

$$F(y^2) = 5(y^2)^2 + 2(y^2) = 5y^4 + 2y^2.$$

Each part of a function's definition has a corresponding representation in an HP 48 user-defined function:

1. The function's name is the name of the variable in which the user-defined function program is stored.
2. The argument name is the local name that follows the →. A *local* name is appropriate because the name is not intended to have a value except when the function is actually being evaluated.
3. The expression defining the function is represented by the defining expression.

The example function $F(x) = 5x^2 + 2x$ is created in the HP 48 as:

```
« → x '5*x^2+2*x' » 'F' STO.
```

Then

```
'F(1)' EVAL ☞ 7,
```

and

```
'F(Y^2)' EVAL ☞ '5*Y^2^2+2*Y^2'
```

In this example, we have considered a function of one variable. User-defined functions defined in terms of more than one local name naturally correspond to mathematical functions of more than one argument.

The command DEFINE makes the correspondence between user-defined functions and mathematical functions even more obvious, since DEFINE creates a user-defined function variable directly from a function definition expressed as an algebraic equation. In section 5.1.1, we described the degenerate case where the left-hand side of the equation is a name with no arguments. In symbolic execution mode (flag −3 clear--see section 3.5.5.2), *'name=expression'* DEFINE

stores *'expression'* unevaluated in a global variable *name.* In numeric evaluation mode (flag −3 set), *expression* is evaluated to a number before storing.

■ *Example:*

'A=10+10' DEFINE

stores '10+10' in variable A in symbolic execution mode; or stores 20 in variable A in numeric execution mode.

DEFINE does a more extensive conversion if the left-hand side of its argument equation is a name followed by a parenthetical list of arguments:

*'function* (*name*$_1$ ··· *name*$_N$) = *expression'* DEFINE

creates a user-defined function named *function* by storing

<< → *name*$_1$ ··· *name*$_N$ *'expression'* >>

in a global variable *function. function* and *name*$_1$ ··· *name*$_N$ must be all be global or local names. (The conversion from the right-hand side of the expression involves a reinterpretation of the expression as if you had re-entered it via the command line, so that names other than the function arguments *name*$_i$ within the expression are converted to global or local names according to the current local memories--see section 9.7.)

■ *Example:*

'F(x,y)=x+y+COS(θ)' DEFINE

stores

<< → x y 'x+y+COS(θ)' >>

in the variable F. x and y are listed as arguments on the left-side of the argument equation, so they are created as local names within the stored defining expression. θ is not listed as an argument, so it is entered as a global name (unless there is a currently existing local memory containing a local variable θ. F is thus a user-defined function of two variables, which references the global variable θ

If DEFINE's argument is not an equation with one of the forms described above, it will return the error Improper Definition. Other errors not directly associated with DEFINE may arise from the evaluation of *expression* in the *'name=expression'* case (numeric execution mode), for which the error message returned by the erring command is reported. Also, when the evaluation is successful but leaves fewer than two objects on the stack for DEFINE, Too Few Arguments is reported but no command is identified.

## 8.5.3 Defining Programs

The preceding discussion has focused on user-defined functions defined by algebraic expressions, since these are the easiest to create (with DEFINE) and correspond naturally to built-in functions. However, you can also create user-defined functions that use a *defining program* in place of the defining expression. An important use of this facility is to create function versions of various RPN commands that you can use in algebraic calculations. For example, you can define a function from HMS+:

```
<< → x y << x y HMS+ >> >> 'HMSP' STO
```

Using HMSP, you can perform hours-minutes-seconds arithmetic within algebraic objects, e.g. '5*HMSP(X,Y)'.

Note, however, that you can not evaluate user-defined functions defined this way with *symbolic* arguments, unless all of the commands in the defining program can accept symbolic arguments. For example, if you evaluate the algebraic '5*HMSP(X,Y)', both X and Y must have real-number values, since HMS+ is not a function. Also, you can not differentiate a user-defined function defined with a program.

User-defined functions defined either with expressions or programs are a special case of the more general use of local variable structures (section 9.7). To qualify at all as a user-defined function, a program must begin with →; otherwise, evaluating an expression containing the program's name with an argument list will return the Invalid User Function error.

## 8.5.4 Additional Examples: Geometric Formulae

■ VCYL($r$,$h$) returns the volume of a right-circular cylinder of radius $r$ and height $h$ from the formula $V = \pi r^2 h$:

```
'VCYL(r,h)=π*SQ(r)*h' DEFINE.
```

■ SCONE($r$,$h$) returns the curved surface area of a right cone of altitude $h$ and radius $r$ from the formula $A = \pi r(r^2 + h^2)^{1/2}$:

```
'SCONE(r,h)=π*r*√(SQ(r)+SQ(h))' DEFINE.
```

■ CSEG(*r*,*x*) returns the area of a segment of a circle, where *r* is the radius, and *x* is the perpendicular distance of the chord from the center, from the formula

$$A = \frac{\pi r^2}{2} - x\sqrt{r^2 - x^2} - r^2 \sin^{-1}\left(\frac{x}{r}\right).$$

'CSEG(r,x)=π*r^2/2−x*√(r^2−x^2)−r^2*ASIN(x/r)'  DEFINE.

■ PPER(*n*,*r*) computes the perimeter of an *n*-sided polygon inscribed in a circle of radius *r* from the formula $perimeter = 2n\,r \sin\frac{\pi}{n}$:

'PPER(n,r)=2*n*r*SIN(π/n)'  DEFINE.

These user-defined functions return symbolic results containing π, unless you clear either flag −2 or −3 (section 3.5.5.2) to cause automatic numerical evaluation of π.

# 9. Programming

Programming is the art of developing sequences of computer operations that can be "replayed" automatically. Such sequences are called *programs*; on the HP 48, programs are objects that you can use as arguments for various operations as well as executing directly. "Programming" on the HP 48 then means the creation of *program objects*, and the use of those objects to achieve various computational tasks.

Creating a program object consists of entering a sequence of objects that are to be executed in order automatically, surrounding the sequence with << >> delimiters. The delimiters identify the sequence as a program, and prevent its immediate execution by ENTER. When you name a program object by storing it in a global variable, you effectively extend the calculator's command set: you can use the variable name just as you would a built-in command. Imagine, for example, that you have created two program objects named DOTHIS and DOTHAT. Then if you want to create a program that performs both of the tasks done by DOTHIS and DOTHAT, you just enter << DOTHIS DOTHAT >>, perhaps naming it DOBOTH. This process is unlimited--you can use DOBOTH as an element of another program. DOTHIS and DOTHAT themselves may be combinations of other program names. As a matter of fact, the HP 48 commands that you use in your programs are themselves programs written the same way, stored in built-in libraries rather than in variables.

We have been using the term *sequence* to mean a series of objects (including commands) that are executed in order. However, a more general definition of sequence includes certain entries that are not objects but are used in building *program structures*. The non-object "entries," examples of which are FOR, DO, →, and END, are called *program structure words*. These are not objects, because you can't put them on the stack or execute them individually, but must use them in certain specific combinations, like FOR...NEXT, or IF...THEN...END. A complete combination, including the objects between the program structure words, is called a *program structure*.

The more complete definition of *sequence*, then, is any series of objects *and* program structures that can "stand alone," i.e. could constitute a program if surrounded by << >> delimiters. A sequence can be all of a program, or part of a program. For example, in

```
<< 1 2 IF A THEN B C END D >>
```

1 2 is a sequence, B C is a sequence, and 1 2 IF A THEN B C END D is a sequence. IF, IF A, and IF A THEN are not sequences, because the program structure is not complete--you can not enter these by themselves without obtaining an Invalid Syntax message.

## 9.1 Program Basics

The basic structure of an HP 48 program is very simple:

<< *program body* >>.

The << and >> are the program object delimiters that serve to identify this object as a program. *Program body* is the sequence of objects and program structures that make up the logical and computational definition of the program.

### 9.1.1 The << >> Delimiters

The << and >> that surround HP 48 programs serve a dual purpose. First, they are the delimiters that identify an object as a program. When you enter a program into the command line, the << tells the HP 48 to create a program object from all of the objects, commands, names, etc., that follow, up to the next matching >>. Then, when the HP 48 displays a program object after it has been created, the << and >> identify the object to you as a program.

The second role of these delimiters is to serve as logical "quotes" (see section 3.8) that postpone execution of a program sequence. When << is encountered in program or command line execution, it is interpreted by the HP 48 to mean "put the following program object on the stack." This behavior of << allows you to include programs within other programs:

<< *objects* >> EVAL

executes *objects*, but

<< << *objects* >> >> EVAL

leaves the program << *objects* >> on the stack. Notice that these are paired delimiters; for every <<, there is always a >>. The trailing >> ends the definition of the program started by the matching <<. When you enter a program into the command line, the HP 48 reminds you of this necessary pairing: pressing [⬅] [<< >>] enters both delimiters (on separate lines) with the cursor in between. The key also activates program entry mode, in which command keys echo their command names to the command line rather than executing the commands. This makes the key the HP 48's closest analog to the more traditional program mode keys you find on other calculators (such as [PRGM] on the HP 41).

### 9.1.2 The Program Body

The "body" of an HP 48 program, that is, everything between the ≪ and the ≫, can consist of any combination of objects and program structures:

- Data objects;
- Quoted names and procedures, which go on the stack like data;
- Commands--RPN commands and functions;
- Unquoted names--which act like user-defined commands;
- Program structures--loops, conditionals, and local variable structures.

To "run" a program, you *execute* the program object, either directly with EVAL, or more commonly, indirectly by executing the program's name. In general, when a program is executed, all of the items from the above list that constitute the program body are executed sequentially. The nominal order of execution is start-to-finish, or "left-to-right" in the command line order in which the program was entered originally. Within a program structure, there may be repetitive loops or conditional jumps. Of course, there's nothing remarkable about this program flow--any programming language exhibits similar orderly execution.

Creating an HP 48 program is straightforward:

1. Press $\boxed{\text{↰}}$ $\boxed{≪ ≫}$;
2. Press the keys for, or spell out, the objects you want the program to execute, in the same order used when you perform the calculation manually; then
3. End the program entry by pressing $\boxed{\text{ENTER}}$ . Alternatively, you can use the cursor keys to move the cursor past the final ≫, to continue with additional command line entries.
4. To name a program, enter a name (quoted) and press $\boxed{\text{STO}}$ . You can consider the resulting variable as a named program.

If you have a computer connected to the HP 48 via the serial port, you can also write programs (and other types of objects) on the computer. There you may use any text editor that can generate text-only (ASCII) files. When you transfer the file to the HP 48, the calculator translates the text into a program object exactly as if you had typed the text into the HP 48 command line. There are several advantages to using the computer for program development:

- The computer's keyboard provides for easier text entry.

- The larger display allows you to format your programs in a more legible manner (see section 1.4).
- Most text editors provide search-and-replace and other editing features to speed up program entry.
- The computer text file provides an automatic archive of your program.
- You can include *comments* in your program text. A comment is text that serves to annotate the program or any of its parts, but is not included in the execution action of the program. Comments, delimited by "@" characters (section 6.4.3.1), are stripped from a program by ENTER, so that they serve no real purpose when you enter a program in the command line. The comment capability was included in the HP 48 specifically for program editing on computers.

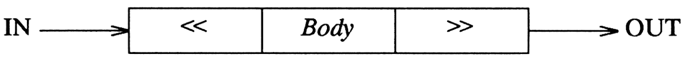
The simplest programs are those which contain no program structures. Such programs only contain objects to be executed one after the other, starting with the first object after the <<, and ending with the last object just before the >>. Examples:

1. `<< 1 2 3 >> 'P123' STO` creates a program named P123 that enters the numbers 1, 2, and 3 onto the stack.
2. `<< 2 / SIN >> 'HSIN' STO` creates a program named HSIN, that returns the sine of 1/2 of the number in level 1.
3. `<< + + SQ >> 'SUMSQ' STO` creates SUMSQ, which adds three numbers from the stack and squares the result.

You can alter the basic start-to-finish execution flow of programs by adding program structures that define branches and loops. *Branches* are forward jumps in a program, that cause program sequences to be skipped. *Loops* contain backward jumps, which cause program sequences to be repeated one or more times. These structures are described later in this chapter.

### 9.1.3 Structured Programming

A property of HP 48 programs that is common among many computer languages, but may be unfamiliar to HP 41 and other calculator language programmers, is their well-determined "entrance" and "exit." That is, in any program there is only one point--the start--where execution can begin. Similarly, there is only one exit, or point at which a program completes execution. A diagram to represent the execution flow in and out of an HP 48 program is very simple:



This diagram is elegantly simple compared with one that represents the program flow in an HP41 or BASIC program. In these languages, there is no limit on the number of entrances and exits in a single program. The principal program constructs that make this possible are labels and GOTO (go to) commands. A *GOTO* is an unconditional jump, with no return, to a label (or line number in some calculators and in BASIC). Using labels and GOTO's, program execution can jump around from program to program, in and out of portions of programs, or round and round within a single program. At first glance (and more, if you're used to programming this way), this capability seems like an advantage. You may wonder why the HP48 does not provide the same capability.

The answer is that the HP48 is designed for *structured programming*. Structured programming consists of writing small programs as building blocks, or modules, from which bigger programs are assembled as series of subroutine executions. A *subroutine* is a program that is executed, or *called*, from within another program, and which returns to the original calling program when it is finished. Bigger programs themselves may become subroutines for even bigger programs, and so on. Each program, at every level, has a single entrance and exit; there is no jumping in and out of programs at intermediate points. Structured programming has the following advantages:

- Programs are easy to write. Each program can be designed to fulfill a single task, and can thus consist of relatively few steps. If a program gets too long, you just divide it into smaller programs.
- Programs are easy to decipher. By choosing meaningful names for subprograms, you can read a program almost as text. For example, a program might look like this:

```
<<  GETINPUT  DOMATH
  IF  BIG
  THEN  IGNORE
  ELSE  SAVE
  END
>>.
```

It is easy to understand what this program does. It gets input (GETINPUT), then does some calculations (DOMATH) on that input. Next, it checks a result to see if it's too large (IF BIG); if so, it discards the result (THEN IGNORE), otherwise saves it (ELSE SAVE). At this level, you can see the overall structure of the program. To see more detail, you can examine the individual subroutines. For example, BIG must be a program that tests the results returned by DOMATH, and returns a *true* flag if the results are too big according to some criterion. BIG might be something like this:

```
<< DUP2 + LIMIT > >>.
```

This program makes copies of two numbers in levels 1 and 2, then adds them and tests to see if the sum is greater than the value of LIMIT (which might be a number, or another calculation to perform, etc.).

- Programs are easy to alter. In the above example, you can completely change the internal definition of BIG, without worrying about the main program. All you have to do is ensure that BIG works the same from an external point of view--it must take the right number of objects from the stack, and return the right number, etc. Similarly, you can change the value of LIMIT from a specific number to a program that computes a result, without any change in the design of BIG.

  In a programming language that permits GOTO's into the middle of a program, any modification of a program must ensure that the correct entry conditions are met at any point at which execution can start. This is especially difficult to manage in languages like BASIC, where a GOTO can jump to any line in a program, with no label or other indication to remind the programmer that execution may start at that line.

- Programs can be written without any regard to the internal behavior of programs that call them, or programs that they may call. All that matters about a program is its input and output, not the steps that it uses in its execution.

The last point is a key concept in HP 48 structured programming. A program is defined externally only in terms of its input and output:

1. The number and type of objects it takes from the stack;
2. The number and type of objects it returns to the stack;
3. The variables that it uses;
4. Flags that are tested or changed.

From the point of view of one program calling another as a subroutine, the first program doesn't have to care *at all* about how many stack levels or additional subroutine returns are needed by the subroutine. It just has to be sure to provide the correct inputs for the subroutine, and know where to find the results returned by the subroutine (usually on the stack). The calling program also can depend on having program execution return to it after the subroutine is finished, no matter how many other sub-subroutines are called by the subroutine.

On the HP 48, there is no structural difference between a *program* and a *subroutine*. Calling a particular program a subroutine is only a matter of convention, often deriving from the circumstance that the program uses very particular arguments or returns special results, that make it unlikely to be used as a stand-alone program.

## 9.2 Program Structures

A simple program consisting of a sequence of objects can be broken into two or more programs at any point in the sequence. For example, the program

```
« 5 * 6 + 10 - »
```

is equivalent to the two programs

```
« 5 * » « 6 + 10 - »
```

executed consecutively.

A *program structure* is a program segment that can not be broken into stand-alone sections. A user-defined function (section 8.5) is an example of a program structure; for example, the program

```
« → x '2*x+3' »
```

can *not* be divided like this:

```
« → x » « '2*x+3' ».
```

The first part would return a Invalid Syntax message when entered. Similarly, you can't break

```
« 1 5 FOR n n SQ NEXT »
```

into

```
« 1 5 FOR » « n n SQ NEXT ».
```

The FOR and the NEXT must be in the same program.

Program structures are defined by *program structure words*. These words are similar to object delimiters, in that they do not themselves represent objects, but are instructions to the HP 48 to build command line text into specific structures. As in the case of object delimiters, the structure words always appear in specific combinations and satisfy certain syntax rules.

Table 9.1 lists all of the built-in HP 48 program structures and their uses. Libraries can add additional structures to the list.

**Table 9.1. HP 48 Program Structures**

| Structure | Type | Typical Use |
|---|---|---|
| IF...THEN...ELSE...END | Conditional | Program decisions |
| CASE...THEN...END...END | Conditional | Selecting among multiple choices. |
| START...NEXT/STEP | Definite Loop | Execute a sequence a specified number of times. |
| FOR *index* ... NEXT/STEP | Indexed Definite Loop | Execute a sequence once for each value of an index. |
| DO...UNTIL...END | Indefinite Loop | Repeat a sequence until a condition is satisfied. |
| WHILE...REPEAT...END | Indefinite Loop | While a condition is satisfied, repeat a sequence. |
| → *...names... procedure* | Local Variable Structure | User-defined functions. Creating local variables. |
| IFERR...THEN...ELSE...END | Error trap | Handle expected and unexpected command errors. |

Before studying the various program structures, we need to describe HP 48 test commands, which along with the flags introduced in section 7.1, are key concepts in understanding the execution of program structures.

## 9.3 Tests and Flags

A calculator program "asks a question" by executing a *test* command. A test command is any command that in effect returns *true* or *false* as a result, which then may be used to choose a particular program branch to execute. In the HP 48, *true* and *false* are represented as stack objects by real number flags, zero for *false* and any non-zero value for *true* (when returned by a command as a result, 1 is used for *true*).

With these ideas in mind, we can make the following definitions:

| | |
|---|---|
| *Test*: | A command that returns a flag to the stack. Examples: SAME, = =, FS? |
| *Logical operator*: | A function that makes a logical combination of two flags (AND, OR, XOR), or inverts a flag (NOT), and returns a new flag. |
| *Conditional*: | A program structure that includes a structure word which uses a flag as an argument, and causes a program branch according to the flag's value. HP 48 conditionals are IF, CASE, DO, and WHILE structures. |

Notice that a test and the corresponding conditional branch are separate operations. To permit this separation, a test command returns its result in the form of a (real-number) flag on the stack, which can then be manipulated like any other stack object. Consider a typical test command, >. > compares real numbers in levels 1 and 2: if the number in level 2 is greater than that in level 1, > returns 1 (*true*); it returns 0 (*false*) if the level 2 number is equal or smaller. For example, to compare the values of X and Y in a program, you use the sequence

```
X Y >.
```

This returns 1 (*true*) if X is greater than Y, or 0 (*false*) otherwise.

In a conditional structure, one particular structure word actually makes the branch decision, taking a flag from the stack for this purpose:

- the THEN in IF...THEN...(ELSE...) END (section 9.4.1).
- each THEN in CASE...THEN...END...END (section 9.4.3)
- the END in DO...UNTIL...END (section 9.5.2.1).

- the REPEAT in WHILE...REPEAT...END (section 9.5.2.2).

But note that you can include any number of intervening objects and commands between the point at which the flag is put on the stack, and the structure word that uses the flag for a branch decision. This separation of tests and decisions makes possible the use of logical operators to combine flags. For example, the logical operator AND takes two flags from the stack and returns a true flag if both of the original flags are *true*, and a false flag otherwise. The sequence

```
X  Y  >  Y  Z  >  AND
```

returns 1 only if X is greater than Y, *and* Y is greater than Z. Furthermore, since the logical operators and most tests (except SAME) are functions, you can rewrite the above sequence in a more legible manner:

```
'X>Y  AND  Y>Z'  →NUM.
```

The →NUM converts the algebraic expression into a real number suitable for use as a flag. If the flag is intended for use in a conditional structure, you can omit the →NUM. All of the structure words listed above automatically perform a numerical evaluation on an algebraic argument. For example,

```
IF  'X>1  AND  Y>1'  THEN  ...
```

and

```
IF  X  1  >  Y  1  >  AND  THEN  ...
```

are equivalent, with the former getting better marks for legibility.

You can even store a flag value then retrieve it for later use by a conditional. Rather than using an ordinary variable, you can use a user flag as the storage location: the flag number replaces a variable name, and the number 1 or 0 is the value. FS? plays the role of RCL for a user flag--it transfers the flag value to the stack. Similarly, SF and CF store the values 1 and 0, respectively, into a user flag. There is no single command to store a stack flag directly into a user flag, but the sequence

```
IF  SWAP  THEN  SF  ELSE  CF  END
```

will accomplish that, where the flag number is in level 1 and the new flag value is in level 2.

One by-product of using real numbers as flags for conditionals is to make it easy to test a real number against zero. In the sequence

```
IF 'X≠0' THEN A ELSE B END,
```

the ≠0 is superfluous. Instead, use

```
IF X THEN A ELSE B END.
```

### 9.3.1 HP 48 Test Commands

The HP 48 test command set is as follows:

- <, >, ≤, and ≥, for comparing the numerical or lexicographical order of two objects. These operators are applicable to real numbers, binary integers, and binary integers, strings, and symbolic arguments. Strings are ordered by their character values, left to right; extra characters count as "higher," e.g. "AA" "A" > returns *true*.
- SAME, == and ≠, for testing equality and inequality. These commands may be used with any types of arguments.
- The flag test commands FS?, FC?, FS?C, FC?C, discussed in section 7.1.1.

For those commands that compare two arguments, the order of the arguments is consistent with the order for other HP 48 functions: the arguments are entered onto the stack in the same order as they appear in algebraic expressions. For example, consider the "greater-than" operator >. In an algebraic expression, "is A greater than B?" is written as "A > B" A is the first argument, reading left-to-right; B is the second. The comparison is *true* if the first argument is greater than the second. If you rewrite the infix operator > in Polish notation, the expression becomes '>(A,B)'. Converting to RPN, this becomes A B >, which indicates that A should be entered into the stack before B. When > executes, A should be in level 2, and B in level 1.

### 9.3.2 Equality

The HP 48 distinguishes two types of equality, *physical* equality and *logical* equality. SAME tests the physical equality of two objects, i.e. whether the two have the same bit pattern in memory. By contrast, for real and complex numbers, binary integers, units, and symbolic objects, == and ≠ test the logical equality of their arguments objects, using the logical values represented by the objects. In most circumstances, the two tests return the same result--if two real numbers have the same numerical value, they also have the same bit patterns. However, there are cases where the two tests will differ:

- == and ≠ can compare real and complex numbers numerically; a real and complex number can be equal if the imaginary part of the latter is zero and the real part is the same as the real number: (5,0) 5 == ☞ 1. SAME always returns *false* when comparing objects of different types.
- == is a function, and thus returns a symbolic result when applied to symbolic arguments. SAME compares the original objects themselves, always returning a flag. Thus, '1+2' 3 == returns the *expression* '1+2==3' (which evaluates to a *true* flag), whereas '1+2' 3 SAME returns a *false* flag.
- When comparing binary integers, == ignores leading zeros and compares only the numerical values, so that the relative wordsize of the two integers does not matter. For SAME to return a true flag, the two integers must have the same wordsize as well as the same value.

For other types of objects, == and ≠ test physical equality in the same manner as SAME. The interpretation of physical equality is so strict that SAME can surprise you by returning *false* in cases where two objects are identical in all outward appearances. For example, if you execute

```
1 'FOO' STO FOO 1 →LIST { 1 },
```

you obtain two lists that certainly look the same. However, SAME and == return 0 for these lists. This is because the object 1 is one of a substantial number of objects that are built into the HP 48's permanent ROM. For sake of memory efficiency, these built-in objects are not copied into RAM *except* when they are stored individually in a global or port variable. Otherwise, they are represented on the stack and in composite objects (section 3.3) by 2.5-byte pointers. In the first list in the above sequence, the 1 is converted to a RAM object (10.5 bytes) when it is stored, whereas the 1 in the second list is a pointer. SAME therefore dutifully reports that the two objects are different. BYTES (section 12.5.1) applied to the two lists also returns different sizes and checksums.

A similar analysis applies to units created by UBASE and UFACT: for example,

```
1_m DUP UBASE SAME ☞ 0
```

When UBASE rebuilds a unit object from base units, the characters (in this case, the "m") in the unit part are taken from a ROM table. A 1_m created by any other means does not contain the ROM character. In this case, however, == does return *true* for these two objects, since this function tests logical equality for unit objects.

It is also important to distinguish == and =. = is *not* a test command, so it is fundamentally different from ==, which is a test. = is a *function* that creates an equation from two expressions. Its execution does not return a flag; in symbolic execution mode, it does nothing other than evaluate its arguments. In numeric execution mode (including using →NUM) it acts the same as −, returning the numerical difference of the two sides of the equation.

= =, on the other hand, is a *test*, and always returns a flag when executed. = = is primarily intended for ordinary numerical equality comparisons. You can use = = in algebraic expressions as an infix operator, just like <, >, etc. = = and = must have different names to distinguish their quite different meanings, and to prevent ambiguity within algebraic expressions. Note that A=B is an "assertion," whereas A= =B is a "question."

## 9.4 Conditional Branches

The program decisions discussed in the preceding sections are most frequently used in conjunction with program *branches*, where execution can proceed along one of two or more paths. The HP 48 does not provide for *unconditional branches*, in which program execution jumps out of the middle of a program without any test. Such branches are used in some programming languages to minimize program size through reuse of steps common to more than one part of a program. On the HP 48, this is achieved by writing the common part as a subroutine that can be called by other programs.

A *Conditional branch* can be one of the following types:

- A *simple branch* consisting of a choice between one of two or more paths, where one or more program sequences are skipped as execution proceeds forward.
- An *iteration loop*, using backwards jumps to repeat execution of a sequence one or more times.
- An *exit* from an iteration loop.

### 9.4.1 Simple Branches: The IF structure.

The most straightforward type of branch involves a choice between executing two different program sequences. On the HP 48, this is implemented with the *IF structure*, a program structure that has the general form:

IF *test-sequence* THEN *then-sequence* ELSE *else-sequence* END

You can read this structure as "if *test-sequence* is *true* (returns a true flag), then execute *then-sequence* and jump past the END. If *false*, skip the *then-sequence* and execute *else-sequence*."

■ *Example*. If the ⬅ SIN key has a user key assignment, display the assignment; otherwise show Unassigned.

| | |
|---|---|
| RCLKEYS DUP | Get the assignment list. |
| IF 41.2 POS DUP | If keycode 41.2 is in the list... |
| THEN 1 - GET | ...then get the assigned object. |
| ELSE DROP2 "Unassigned" | ...otherwise, enter a string. |
| END | |
| 1 DISP 1 FREEZE | Display the result. |

This sequence uses the real number returned by POS both as a flag to indicate whether the search was successful, and also, if non-zero, to specify the keycode's position in the list.

The ELSE *else-sequence* portion of an IF structure is optional. For cases where the *else-sequence* is unnecessary, you can use this form:

IF *test-sequence* THEN *then-sequence* END,

which translates to "If *test-sequence* is *true*, execute then-sequence; otherwise, skip past the END."

■ *Example*. Order two numbers so that the smaller one is returned in level 1, the greater in level 2.

| | |
|---|---|
| DUP2 | Copy the two numbers. |
| IF < | Test if the first is less than the second. |
| THEN SWAP | If so, switch the numbers. |
| END | |

■ Because it is THEN that actually removes a flag from the stack and makes the branch decision, the position of the IF in the sequence that precedes THEN is unimportant:

```
1  2  IF  >  THEN  ...,  and
1  2  >  IF  THEN  ...,  and
IF  1  2  >  THEN  ...,
```

all produce the same result. You can choose to position the IF wherever you want to make a program the most readable. (The most memory-efficient form has a single object between the IF and the THEN. Thus of the three forms above, the first uses the least memory. See section 12.5.)

## 9.4.2 RPN Command Forms

An alternate means of achieving IF structure branching is provided by the IFTE and IFT commands. For these commands, the various sequences included in an IF structure are entered as stack arguments, either as single objects or as programs. That is,

*test-sequence* « *then-sequence* » « *else-sequence* » IFTE

is equivalent to

IF *test-sequence* THEN *then-sequence* ELSE *else-sequence* END.

Similarly,

*test-sequence* « *then-sequence* » IFT

is equivalent to

IF *test-sequence* THEN *then-sequence* END.

To use IFTE, you put a flag in level 3, an object (usually a program) representing the *then-sequence* in level 2, and an object representing the *else-sequence* in level 1. IFTE tests the flag; if the flag is *true* (non-zero), the *else-sequence* is dropped, and the *then-sequence* is executed. If the flag is *false* (zero), the *then-sequence* is dropped, and the *else-sequence* is executed. IFT works much the same way: the flag must be in level 2, and a *then-sequence* in level 1. If the flag is *true*, the *then-sequence* is executed, otherwise it is dropped.

▪ *Example.* Split a real or complex number into its real and imaginary parts.

| RC→R | *Real/Complex to Real* | | | 8A8F |
|---|---|---|---|---|
| | *level 1* | \| | *level 2* | *level 1* |
| | *x* | ☞ | *x* | 0 |
| | (*x*,*y*) | ☞ | *x* | *y* |

| | |
|---|---|
| « DUP TYPE | Get the input type. |
| « C→R » | Complex case (type ≠ 0). |
| 0 | Real case (type 0)--just push zero on the stack. |
| IFTE | Execute appropriate choice. |
| » | |

There is no particular advantage within a single program to using IFT or IFTE rather than the corresponding IF structure, so which form you use is mostly a matter of taste. However, the RPN command forms have one advantage for more sophisticated programming: their use allows you to place the *test-sequence*, the *then-sequence*, and the *else-sequence* in separate programs or program structures. If you use an IF structure, all must be contained in the same program.

IFTE is also a function, which means you can use it in algebraic objects as well as in programs. It is a prefix function of three arguments:

IFTE(*test-expression, then-expression, else-expression*)

Notice that the arguments are in the same order as the stack arguments when IFTE is executed as an RPN command. All three arguments are ordinary expressions. *Test-expression* is evaluated, and its value is interpreted as a flag. If the flag is *true*, *then-expression* is evaluated; if the flag is *false*, *else-expression* is evaluated. Typically, the *test-expression* contains a comparison operator, so that evaluation automatically returns a flag.

▪ *Example.* 'IFTE(X==0,1,SIN(X)/X)' computes SIN(X)/X, returning the value 1 when X is zero.

IFT has no algebraic form. This is because algebraic objects must return a result when evaluated--an algebraic conditional can't "do nothing" if the test flag is false.

### 9.4.3 The CASE Structure

The IF structures described in the previous section are convenient for branching that is based on a single test to select between two choices. While it is possible to handle any more elaborate combinations of tests and choices with "nested" IF structures, the overall structure can get rather convoluted. For more straightforward handling of multiple tests and choices, the HP 48 provides the *CASE structure*, which has the following general form:

```
CASE
    test-sequence₁ THEN then-sequence₁ END
    test-sequence₂ THEN then-sequence₂ END
    .
    .
    .
    test-sequenceₙ THEN then-sequenceₙ END
    else-sequence
END
```

You can read the CASE structure as "execute *test-sequence*$_1$, then *test-sequence*$_2$, etc., until one *test-sequence* returns *true*. Then execute the corresponding *then-sequence,* and skip to past the final END. If no *test-sequence* returns true, then execute *else-sequence*.

■ *Example*. The program COUNT4 is a simple four "bin" counting routine.

| COUNT4 | *Count in 4 Ranges* | 8F8C |
|---|---|---|
| | *level 1* \| *level 1* | |
| | $x$ ☞ | |

| | |
|---|---|
| « CASE | |
| DUP 0 < THEN DROP 1 END | Range 1 if $x<0$. |
| DUP 0 == THEN DROP 2 END | Range 2 if $x = 0$. |
| 1 ≤ THEN 3 END | Range 3 if $0<x\leq 1$. |
| 4 | Other tests failed, so $x$ must be greater than 1 (range 4). |
| END | |
| 'COUNTS' SWAP DUP2 | Make two copies of the vector name and the index. |
| GET 1 + PUT | Get the element, add 1, put it back. |
| » | |

COUNT4 tests an argument $x$ to see in which of four ranges its value lies. The total in each range is stored in the four-element vector COUNTS. The elements of the vector represent these ranges:

| Element | Range |
|---|---|
| 1 | $x<0$ |
| 2 | $x=0$ |
| 3 | $0<x\leq 1$ |
| 4 | $x>1$ |

Another way to make a multi-case choice is to create a list of programs, then select one of the programs from the list according to an index. For example, this sequence takes a real number from the stack, and executes a name corresponding to the number:

| | |
|---|---|
| { ONE TWO THREE FOUR FIVE } | List of name choices. |
| SWAP | Put the index in level 1. |
| GET | Get the indexed choice. |
| EVAL | Execute the selected name. |

## 9.5 Loops and Iteration

A *loop* is a program structure containing a sequence that is *iterated*--executed more than once. In a *definite loop*, the number of iterations is known in advance. In an *indefinite loop*, the iteration continues until some specified condition is met, after which execution exits from the loop and continues with the rest of the program.

### 9.5.1 Definite Loops

The most common form of definite loop structure is the FOR...NEXT loop. This kind of loop is appropriate when you want a program sequence to repeat several times, making use of an *index* that is incremented by 1 at each iteration of the sequence. The general form of a FOR...NEXT loop is:

*start* *stop* FOR *name* *sequence* NEXT,

where

- *start* is the (real number) initial value of the index.
- *stop* is the (real number) final value of the index.
- FOR identifies the start of the structure; it removes the start and stop values from the stack.
- *name* is the name of the (local) variable that contains the index.
- *sequence* is any program sequence, which can contain any number of uses of *name*.
- NEXT is the structure word that identifies the end of the sequence. It increments the index by one, then tests its value against the stop value to determine whether to repeat the sequence.

You can read a FOR...NEXT loop as "For each value from *start* through *stop* of an index named *name*, execute the sequence that ends with NEXT."

■ *Example*. Enter onto the stack the squares of the integers from 1 through 100.

| | |
|---|---|
| `1 100` | Start and stop values. |
| `FOR n` | Create a local variable n, with initial value 1. |
| `  n SQ` | Square the current index. |
| `NEXT` | Increment n by 1. If $n \leq 100$, loop again. |

A few ob

- *Start* and *stop* as shown above are *not* part of the FOR...NEXT program structure. FOR expects to take two real numbers from the stack, but those numbers can be entered or computed at any time in advance of the FOR, as long as they are in levels 1 and 2 when the FOR executes.
- The *start* and *stop* values are removed from the stack by FOR. They are not accessible afterwards; if a program needs their values for other purposes, it should copy them or store them in variables before executing the FOR.
- The *index* is kept in a local variable identified by the name that immediately follows FOR. You can return the current value of the index by executing its name. You can also change the value of the index after the loop has started, by storing a real number into the local variable. The naming and use of the index variable are subject to the same restrictions as local variables created by → (section 9.7). After the loop is finished, the index variable is automatically purged.
- The name following a FOR is *not* part of the sequence that is repeated. For example,

  ```
  1  10  FOR  n  n  NEXT
  ```

  puts integers 1 through 10 on the stack, but

  ```
  1  10  FOR  n  NEXT
  ```

  accomplishes nothing.
- The sequence between FOR *name* and NEXT always executes at least once, even if the specified *stop* value is less than the *start* value.
- The *start* and *stop* values don't have to be integers. NEXT always increments the index by 1; the loop will repeat as long as the index is less than or equal to the stop value.

  .5  .6  FOR  n  *sequence*  NEXT

  executes *sequence* once, with n = .5.
- The combination FOR *name* acts like a single operation when you single-step (section 12.2.2) the FOR.
- *Start*, *stop*, and *step* can be algebraic objects, as long as they evaluate to real numbers.

### 9.5.1.1 Summations

A common form of iteration is a *summation*, in which successive values are accumulated to a total. To add the squared integers computed in the previous section, we can modify the example as follows:

| | |
|---|---|
| 0 | Initialize the total. |
| 1 100 | Start and stop values. |
| FOR n | Create a local variable n, with initial value 1. |
| n SQ + | Square the current index, and add to the total. |
| NEXT | Increment n by 1. If n≤100, loop again. |

Executing this sequence returns 338350.

For cases where the successive terms in the sum can be represented by algebraic expressions, the HP 48 provides the summation function Σ. Σ takes four arguments:

*index start stop summand* Σ ☞ *sum*.

*Index* must be a name, and the other three arguments can be algebraic objects or any objects permitted within an algebraic object. *Summand*, of course, is usually a function of *index*.

As well as being more compact and legible compared to the FOR...NEXT form, Σ is also a function; used itself in an algebraic object, it is a prefix function with this syntax:

Σ(*index=start,stop,summand*).

(Notice the required = sign). In standard mathematical notation, and in the Equation-Writer display, this translates to

$$\sum_{index=start}^{stop} summand.$$

*Summand* is usually an expression containing *index*. Using the summation function, the sum of squares computed above can be obtained by evaluating

Σ(n=1,100,n^2).

When Σ is evaluated, it evaluates *start* and *stop*, then returns:

- The same sum except with the evaluated limits, if either of the evaluated limits is still symbolic;
- A sum of symbolic terms, if both limits evaluate to numbers and the summand contains symbolic arguments other than the index, thus

'Σ(I=1,2,I+A)' EVAL ☞ '1+A+(2+A)';

- A numeric sum, if the limits and the summand all evaluate to numbers, thus

'Σ(I=1,2,I+1)' EVAL ☞ 5.

A sum can be differentiated:

'Σ(I=A,B,F(X,I))' 'X' ∂ ☞ 'Σ(I=A,B,∂X(F(X,I))'

A sum may also be integrated symbolically. When the integral is evaluated, if the summand is an integrable pattern, the result is the (unevaluated) sum with the summand replaced by its definite integral. If the summand is not integrable, the result retains the sum as the integrand (i.e. the integral is not pushed inside the sum).

#### 9.5.1.2 Varying the Step Size

The FOR...STEP program structure is a variation of FOR...NEXT, which allows you to increment the loop index by amounts other than one, including negative values. A FOR...STEP structure looks like this:

*start* *stop* FOR *name* *sequence* STEP.

*Start*, *stop*, *name*, and *sequence* play the same roles as in FOR...NEXT loops. The structure word STEP plays a similar role to NEXT, but allows you to control the amount by which the index is incremented (or decremented). STEP takes a real number from level 1, and adds it to the current value of the index. Then:

- If the *step* value is *positive*, the loop repeats if the index is less than (more negative) or equal to the stop value.
- If the *step* value is *negative*, the loop repeats if the index is greater than (more positive) or equal to the stop value.

Note that since STEP takes a number from the stack, *sequence* must end with the step value on the stack (the step value doesn't have to be the same each time).

▪ *Example*. The program DFACT computes the double factorial $n!! \equiv n(n-2)(n-4)...1$, where $n$ is an integer.

| DFACT | *Double Factorial* | 605F |
|---|---|---|
| | *level 1* \| *level 1* | |
| | *n* ☞ *n!!* | |

| | |
|---|---|
| « 1 | Initialize the product. |
| SWAP 2 | Loop from *n* down to 2. |
| FOR m | m is the index. |
| m * | Multiply the product by m. |
| -2 STEP | Decrement m by 2. Repeat if m≥2. |
| » | |

### 9.5.1.3 Looping with No Index

In some circumstances, there is no need for an index when a program sequence is to be repeated a fixed number of times. In such cases, you can use START in place of FOR. START...NEXT and START...STEP are the same as FOR...NEXT and FOR...STEP, respectively, except that the loop index is not accessible. The index name that must follow FOR is not used with START (if a name does follow START, it is just treated as part of the loop sequence, and has nothing to do with the loop index).

■ *Example.* The program VSUM sums the *n* elements of a vector.

| VSUM | *Sum Vector Elements* | ACD8 |
|---|---|---|
| | *level 1* \| *level 1* | |
| | [ *vector* ] ☞ *sum* | |

| | |
|---|---|
| « OBJ→ OBJ→ | Put the elements on the stack, with the number of elements in level 2, and a 1 in level 1. |
| SWAP OVER - | Loop start and stop values for $n-1$ additions. |
| START + NEXT | Execute + $n-1$ times. |
| » | |

### 9.5.1.4 Exiting from a Definite Loop

Definite loop structures are designed to repeat a predetermined number of times. There is no "exit" command that can cause program execution to jump out of a loop before it has completed the specified number of iterations. Ordinarily, you should use an indefinite loop (section 9.5.2) for calculations where you don't know in advance how many iterations are needed. However, indefinite loop structures don't provide an automatic index like that in FOR...NEXT/STEP loops, so for some problems you may find it more convenient to use a definite loop with a contrived exit rather than an indefinite loop where you have to provide your own index.

All you have to do to cause a loop to exit before the prescribed number of iterations is to store a number greater than or equal to the stop index value into the index variable. In loops with a positive *step* size, an obvious choice for an exit value is MAXR, the largest number that the HP 48 can represent, although you have to be sure to convert the symbolic constant into a real number. For loops with a negative *step*, you can use -MAXR.

Typically, the exit from a definite loop is taken as the result of a test. The general form of such a loop is as follows:

```
start  stop
FOR  n  sequence
  IF  test
  THEN  MAXR  →NUM  'n'  STO
  END
NEXT
```

This structure executes *sequence* for every value of *n* starting with *start*, and ends when either *n* is greater than *stop*, or *test* returns a true flag.

- *Example*. Determine the value of N for which $\sum_{n=1}^{N} n^2 \geq 1000$.

| | |
|---|---|
| `0  1  10000` | Initial value of sum; *start* and *stop* values. |
| `FOR  n` | Loop index is n. |
| `  n  SQ  +` | Increment the sum. |
| `  IF  DUP  1000  >` | Is the sum ≥1000? |
| `  THEN  n` | The current value of the index is *N*. |
| `    MAXR  →NUM  'n'  STO` | Set the index past the stop value. |
| `  END` | |
| `NEXT` | |

Executing this sequence returns the sum 1015, and the value 14 for *N*.

## 9.5.2 Indefinite Loops

An *indefinite loop* is a loop where the number of iterations is not determined in advance. Instead, the loop repeats indefinitely until some exit condition is satisfied.

The HP 48 provides two program structures for indefinite looping, the *DO loop* and the *WHILE loop*. The primary difference between the two structures is the relative order of the test and the loop sequence. In a DO loop, the sequence is performed first, then the test; in a WHILE loop, the test is performed first.

### 9.5.2.1 DO Loops

The basic form of a DO loop structure is:

DO *loop-sequence* UNTIL *test-sequence* END.

*Loop-sequence* is any program sequence. *Test-sequence* is a second program sequence, which must end with a flag on the stack. END removes the flag; if the flag is *false* (zero), execution jumps back to the start of *loop-sequence*. If the flag is *true* (non-zero), execution proceeds with the remainder of the program after the END. You can read a DO loop as:

"Do *loop-sequence* repeatedly, until *test-sequence* is *true*."

▪ *Example*. Compute $\sum_{n=1}^{\infty} \frac{1}{n^5}$.

▪ *Solution*: The sequence below sums terms of the form $n^{-5}$, until two consecutive sums are equal. Executing the sequence returns 1.03692775496, after 184 iterations.

| | |
|---|---|
| `1 'N' STO` | Initialize a variable N as a counter. |
| `0` | Initialize the sum. |
| `DO` | Start of loop. |
| `  DUP` | Copy the old sum. |
| `  N -5 ^ +` | Add $n^{-5}$. |
| `  1 'N' STO+` | Increment the counter. |
| `  SWAP` | New sum in level 2, old in level 1. |
| `UNTIL` | Start test-sequence. |
| `  OVER ==` | True if old sum = new sum (leaves only new sum in level 1). |
| `END` | Repeat if test was *true*, otherwise done. |

The position of the UNTIL between DO and END is unimportant. That is, the division of the program steps into *loop-sequence* and *test-sequence* is only a matter of program legibility. Both *loop-sequence* and *test-sequence* are executed at each iteration of the loop, so it doesn't matter where you put the UNTIL. We recommend that you use the UNTIL to isolate that portion of the program that constitutes the logical test--the program steps which produce the flag that determines whether or not to repeat. The portion that precedes the UNTIL should be the part of the loop that computes the results used by the remainder of the program after the END.

To reverse the sense of the test, that is, to make a loop that repeats until a test is *false*, you can either substitute an opposite test command (> for <, FC? for FS?, etc.), or insert a NOT immediately before the END:

DO *loop-sequence* UNTIL *test-sequence* NOT END.

In the example above, we used a global variable N to hold the summation index. It is not uncommon to have an indefinite loop that uses an index or a counter similar to that used in definite loops. For simple incrementing by one, you may find it convenient to use INCR, which takes a global or local name as an argument and executes the equivalent of DUP 1 STO+ RCL, using fast "in-place" arithmetic. DECR serves a similar purpose when you want to decrement by one. These commands perform a final RCL expressly so that the index value is available for testing for a loop exit condition. For example, the following sequence is equivalent to a FOR...NEXT loop:

| | |
|---|---|
| → index stop | Initialize the *index* and save the *stop* value. |
| « DO | |
| *loop-sequence* | |
| UNTIL | |
| 'index' INCR stop == | Iterate until the incremented index is equal to the *stop* value. |
| END | |
| » | |

Here we have used local variables to hold the index and stop values. The point of the example is not to suggest replacing FOR...NEXT loops, but to show how you might write a loop that combines features of indefinite loops and definite loops. Such a loop can use INCR or DECR to maintain an index, while using a more elaborate exit condition than is convenient with FOR...NEXT loops.

### 9.5.2.2 WHILE Loops

In a WHILE loop, a test sequence is defined in the first part of the structure:

WHILE *test-sequence* REPEAT *loop-sequence* END.

Here again *loop-sequence* is any program sequence, and *test-sequence* is any sequence that returns a flag. REPEAT removes the flag; if the flag is *true*, the program executes *loop-sequence*, then loops back to *test*. If the flag is *false*, *loop-sequence* is skipped, and execution proceeds with the remainder of the program after the END. You can read a WHILE loop like this:

"As long as *test-sequence* is *true*, keep repeating *loop-sequence*."

▪ *Example.* The program GCD finds the greatest common divisor (GCD) of two integers $m$ and $n$. GCD repeatedly computes $r = m \bmod n$; if each successive $r$ is nonzero, it replaces $n$ with $r$, $m$ with $n$, and repeats. When $r$ is finally zero, the value of $n$ is the GCD.

| GCD | *Greatest Common Divisor* | | | E895 |
|---|---|---|---|---|
| *level 2* | *level 1* | | *level 1* | |
| $n$ | $m$ | ☞ | GCD($n$,$m$) | |

| | |
|---|---|
| « WHILE | Beginning of test-sequence. |
| DUP2 | Make 2 copies of $m$ and $n$. |
| MOD | Compute $r = m \bmod n$ |
| DUP 0 ≠ | Test $r \neq 0$. |
| REPEAT | If true, do the following: |
| ROT DROP | Replace $m$ and $n$ by new values. |
| END | Loop back and repeat the test-sequence. |
| ROT DROP2 | Leave $n$ in level 1. |
| » | |

To reverse the sense of the test, that is, to make a loop that repeats while a test is *false*, you can either substitute an opposite test (> for <, FC? for FS?, etc.), or insert a NOT immediately before the REPEAT:

WHILE *test-sequence* NOT REPEAT *loop-sequence* END.

### 9.5.2.3 DO vs. WHILE

DO loops and WHILE loops are very similar in purpose, and often you can use either form for a programming problem. Here is a summary of the differences between the two structures:

- In a DO loop, the test for looping is made *after* the *loop-sequence* is executed. In a WHILE loop, the test is made before the *loop-sequence*.
- In a DO loop, the *loop-sequence* is executed at least once, and again at every iteration. In a WHILE loop, the *loop-sequence* may not be executed at all. In general,

the WHILE loop *loop-sequence* is executed one time fewer than the *test-sequence*.

- The position of UNTIL between DO and END is arbitrary, and has no effect on results. The position of REPEAT between WHILE and END is significant.

## 9.6 Error Handling

An HP 48 *error* is a circumstance in which normal execution is stopped because the HP 48 is unable to proceed without your intervention. Errors range from simple cases, such as DROP executed with an empty stack, to the extreme case where there is so little free memory that the HP 48 is unable even to display the stack contents. When an error occurs, the HP 48 normally stops all current execution, beeps, and displays an error message. Usually, if the error occurs during execution of a command, the error display also identifies the erring command.

Whether a particular circumstance is an error or not is a matter of design and convention. On most calculators, taking the square root of $-1$ is an error; the HP 48 is designed instead to return a complex number result. The calculator could similarly return some sort of default result in almost any situation. The Invalid Syntax error, for example, could be eliminated by having ENTER return the command line as a string when the object syntax in the command line is incorrect. That, however, would generally be more misleading and inconvenient than the immediate error signal that requires you to fix a bad entry. This is the general philosophy behind all of the HP 48 error conditions--the calculator would rather stop and have you take action than to proceed with a possibly inappropriate action of its own.

Most HP 48 capabilities are programmable, and error handling is no exception. By using the *IFERR structure*, a program can intercept any or all errors (except Out of Memory) and supply its own corrective action. The structure is also called an *error trap*, since it "traps" an error before it can interrupt the overall program execution. The IFERR structure has the following general form:

IFERR *error-sequence* THEN *then-sequence* ELSE *normal-sequence* END,

where the three sequences are arbitrary program sequences. You can read an IFERR structure as:

"If any error occurs during the execution of *error sequence*, then execute *then-sequence* and continue execution after the END. If no error occurs, skip *then-sequence* and execute *normal-sequence*, and continue on after the END."

There does not have to be a *normal-sequence*--the ELSE *normal-sequence* is optional.

IFERR *error sequence* THEN *then-sequence* END

executes *then-sequence* if an error occurs during *error sequence*, but does nothing special otherwise.

■ *Example*. Compute sin$x/x$, where $x$ is a stack argument, using an IFERR structure to handle the undefined result error condition at $x = 0$.

```
DUP  SIN  SWAP  IFERR  /  THEN  DROP2  1  END
```

This sequence returns 1 for an argument of zero.

The position of the IF structure word in the sequence preceding THEN in an IF structure is unimportant because it is THEN that actually makes the branch decision. However, the position of IFERR in an IFERR structure *is* significant; the IFERR and the succeeding THEN define the extent of the sequence for which errors are trapped. IFERR A B THEN intercepts errors in A and B, whereas A IFERR B THEN traps errors occurring only in B. The jump to the *then-sequence* happens immediately upon the error; any remaining steps preceding the THEN are skipped. Thus if an error occurs in A in the structure IFERR A B C THEN D END, B and C are not executed--execution jumps from the point in A where the error occurred directly to D.

Because the reaction to an error is usually specific to a particular error, it is generally a good idea to keep the *error-sequence* short, containing as few as one object if possible. Then there is no ambiguity about which object caused the error, and no part of the sequence that will be skipped. Of course, even a single object may cause different types of errors. To sort out such possibilities, you can use the ERRN command (in the third page of the program control menu) to return the error number of the most recent error, and ERRM to return the text of the error message. For example, suppose that a program adds two arguments. The addition can fail either because the stack is empty, or because the arguments are of the wrong type. The following error trap can deal with either problem:

| | |
|---|---|
| IFERR + | |
| THEN ERRN | Get the error number. |
| IF #201h == | Is it error 201 (Too Few Arguments)? |
| THEN GETMORE | Use GETMORE to get more arguments. |
| ELSE ERRM ABORT | If the arguments are the wrong type, return the error message as a string. |
| END | |
| END | |

The number returned by ERRN, expressed in hexadecimal, is a (up to) five-digit number. The first three digits are the library number of the library containing the command that reported the most recent error. The last two digits are just the number of the error message in the library's message table. Built-in libraries have single-digit library numbers; for example, the Too Few Arguments error illustrated in the preceding example is the first error in library 2, which contains all of the error messages related to generic stack operations.

It is sometimes useful for a program to determine whether a particular error has occurred, after any trapping of that error has taken place. It is not always sufficient just to check the last error number using ERRN, since that value might have been established prior to the execution of the error trap. To prevent this ambiguity, you can use ERR0 to reset the error number to zero prior to an error trap. ERR0 also resets the error message returned by ERRM to an empty string.

## 9.6.1 The ATTN Key

Pressing ATTN normally aborts current procedure execution and returns the HP48 to manual operation (see also section 6.2.3). ATTN thus behaves similarly to an error, except that there is no beep or message display. In all other respects, you can treat ATTN as an ordinary error that has error number zero and a null error message. In particular, you can trap ATTN with an IFERR structure. You might do this in order for a program that is interrupted by ATTN to have a chance to "clean up" before terminating execution, or to prevent termination entirely.

Examples of both of these uses of ATTN trapping are given in the program ASN41 in section 7.2.1.1. In that program, the first error trap, around INPUT, lets you abort the assignment but discards the three INPUT arguments before quitting. The second error trap, around 0 WAIT, allows you to make an assignment to ATTN by pressing it--without the trap, pressing ATTN would abort the program. Notice that in both cases, ATTN is the only error possible, so the error trap does not need to check the error number.

In order to intrude less on program error handling, ATTN does not change the error number and message returned by ERRN and ERRM when it is pressed during the execution of non-programmable operations such as the EquationWriter, the interactive stack (section 4.5) or any of the catalogs.

## 9.6.2 Custom Errors

An error trap lets you prevent an ordinary error from interrupting program execution. However, the reverse situation may also arise: you would like to abort program execution and report an error when nothing has occurred that the calculator recognizes as an error. This also includes cases where an error trap has intercepted an error then decides to go ahead and report the error anyway. These purposes are accomplished by

DOERR (*DO ERRor*).

You can create a *custom error* by executing DOERR with a string argument. "*Message*" DOERR generates an error condition just like a command error:

- Procedure execution aborts, and the calculator beeps.
- The text of "*message*" is displayed in line 1 of the display. You can also create a two-line message by including a newline character (10) in the message. Each line should be 22 characters or fewer to fit on the display.
- Subsequent execution of ERRN and ERRM return #70000h and "*message*", respectively. #70000h is a special error number reserved for DOERR.
- You can trap DOERR like any other error.

DOERR will reproduce an ordinary error condition when it is used with a numerical argument. The number, which may be either a real number or a binary integer, should be the error number of a built-in or library error (DOERR does not display any command name along with Error:). If there is no message corresponding to the number, the display will show Error: with no additional text. 0 DOERR is a programmable equivalent of $\boxed{\text{ATTN}}$ ; execution causes a program to abort with no beep or error message.

You can observe that the errors listed in the HP48 owner's manuals are not always numbered consecutively. There are, for example, apparently no errors between 106h and 123h. However, if you execute #107h DOERR, the HP48 will beep and display Error: Real Number. The explanation is that all of the text used in HP48 displays is entered in the various libraries' message tables, along with the error messages. Messages 106h-122h happen to be the object type text that the HP48 uses to display the stack contents in low memory situations. DOERR does not attempt to distinguish which messages correspond to normal errors. The program MSGSHOW in section 12.6.4.4 lets you review all HP48 messages.

### 9.6.3 Error Handling and Argument Recovery

The design of an error trap must take into account whether last arguments recovery (section 4.3) is active at the time an error occurs. If argument recovery is enabled, the arguments of the command that errors are restored to the stack. If recovery is disabled, the arguments are discarded. This difference obviously can have an effect on error traps, which may need to take into account the contents of the stack after an error. The $\sin x/x$ example at the beginning of section 9.6 assumes that argument recovery is enabled. The DROP2 in the *then-sequence* is intended to discard the two zeros that cause the division error, and which are restored by the error system. If recovery is disabled, the DROP2 is inappropriate because the two zeros are not returned after the error.

A well-designed program, including its error traps, should work correctly regardless of whether argument recovery is enabled or disabled. There are two general approaches:

1. Set or clear flag −55 in the program before an error trap, then write the IFERR structure accordingly. Returning to the $\sin x/x$ example, either

   ```
   -55  CF  DUP  SIN  SWAP  IFERR  /  THEN  DROP2  1  END
   ```

   or

   ```
   -55  SF  DUP  SIN  SWAP  IFERR  /  THEN  1  END
   ```

   will work. This method has the disadvantage that it may alter the state of flag −55 and thus affect other programs that may depend on the flag. As a rule, any program that does depend on flag −55 or any other flag should itself set the flag the way it wants, so this should not be a major limitation.

2. Include a conditional in the *then-sequence* that can react to the current state of flag −55 without altering it. For example,

   ```
   DUP  SIN  SWAP
   IFERR  /
   THEN
     IF  -55  FC?
     THEN  DROP2
     END
     1
   END
   ```

## 9.6.4 Exceptions

A mathematical *exception* is an error condition encountered in the execution of certain functions, for which the HP 48 has a built-in error trap that lets you control how the condition is handled. You can treat an exception as an execution-halting error, or have the calculator supply a default result and continue normally. You make your choice by means of the three exception action flags (−20, −21, and −22).

A typical exception is division by zero. The behavior of / when the divisor is zero i controlled by flag −22, the *infinite result action* flag. If flag −22 is clear (the default) division by zero is treated as an error, causing the Infinite Result error. However, if fla

−22 is set, no error is reported, and one of the values ±9.99999999999E499 (±MAXR) is returned, which are the HP 48's best representations of ±∞. The sign of the result is determined by the sign of the dividend.

The choice to error or to supply a default generally depends on whether the exceptional condition is expected. For example, if you don't anticipate that a program might cause a division by zero, it is better to clear flag −22 so that the program will halt and report the error. On the other hand, if you know that the divide-by-zero situation can happen, and that ±MAXR is a good approximate result that lets a calculation proceed to meaningful results, then setting flag −22 is a good choice.

A program can detect when an exception occurs even if the action flag is cleared to prevent an execution halt. Flags −23 through −26 act as *signal* flags--when an exception occurs, the corresponding signal flag is set automatically. For example, flag −26 is set by an infinite result exception. Therefore, a program can clear flag −26, carry out a calculation with flag −22 set, and still determine if a division by zero occurred by testing flag −26.

In addition to the infinite result exception, the HP 48 also recognizes two other exceptions:

- *Overflow* (action flag −21, signal flag −25). Overflow occurs when a function returns a result that is finite, but larger than the HP 48 can represent, such as FACT(2000). With flag −21 clear (the default setting), overflowing functions return ±9.99999999999E499. Setting flag −21 causes an overflow to return an error. An overflow is *not* the same as an infinite result, for which the correct value is ±∞ rather than a too-large finite number.
- *Underflow* (action flag −20, signal flags −23 and −24). Underflow occurs when a function returns a result that is not zero but is smaller in absolute value than 1E−499 (MINR), the smallest non-zero number that the HP 48 can represent. If flag −20 is clear (the default setting), any underflowing function returns zero as its default result. Since zero has no sign, two signal flags are used: flag −23 is set to indicate that the function underflowed from the negative side of zero; flag −24 is set to indicate underflow to a small positive number.

Notice that the sense of the underflow and overflow flags is reversed from that of the infinite result flag. That is, you *set* flag −22 to prevent an error, whereas you *clear* flag −20 or −21. HP 48 mode flags are clear in the default state, and the defaults are that an infinite result is an error but overflow and underflow are not.

0÷0 is *not* an exception. That quantity is mathematically undefined--it is neither an overflow nor an infinite result. There is no appropriate default result to supply, so the

HP 48 always reports the Undefined Result error and halts execution. You can, of course, create your own exception handing by using an IFERR structure to trap this error.

## 9.7 Local Variables

The variables that you see cataloged in the VAR menu are called *global* variables because they are accessible from any procedure, and remain in memory until you specifically remove them. However, the HP 48 also provides *local* variables that are associated only with individual procedures. The use of these variables and the corresponding *local name objects* is a very useful and powerful programming technique.

It is possible, with the "unlimited" stack provided by the HP 48, to carry out an arbitrarily complicated calculation on the stack without any use of variables to store inputs, intermediate results, or final outputs. The fastest and most efficient computation is usually achieved in this manner.

A language like BASIC, which has no stack at all, requires that all input, output, and intermediate results must be stored in variables. This makes individual BASIC statements easy to read, but not particularly efficient. Nevertheless, the popularity of BASIC suggests that it is not always program execution efficiency that is paramount, but rather the overall "throughput" of the problem solving process. If a calculator is easy to program, you can usually get a result in less total time even if the program itself may execute more slowly than if you developed a solution in an efficient but arcane language. Thus while you *can* write a HP 48 program that is a marvel of structure and efficiency by using only stack objects, the time and skill required for you to keep track of everything on the stack during program development may be too high a price for the result. In short, there is often a compelling advantage to assigning names to objects to simplify the programming process.

At first glance this seems to imply the use of global variables, which are always accessible and appear automatically in the VAR menu. However, while global variables are fine for "permanent" data and procedures, they are not as attractive for storing intermediate results. They stay around indefinitely, so that you have to remember to purge them to avoid cluttering up the VAR menu and to conserve memory. Furthermore, you have to be careful when you create a variable in one program to avoid using the same name as that used by another program, unless you deliberately intend the two programs to share a common variable.

HP 48 *local* variables are a means for saving intermediate data and results that is intermediate between using the stack exclusively and using global variables. Local variables exist in *local memories*, which are portions of RAM temporarily allocated for the local variables. A local memory is accessible only within a context defined by the program structure that creates it. This means that there cannot be any name conflicts with global variables or other procedures' local variables. Also, when the defining structure has completed its execution, its local memory with all of its local variables is automatically deleted.

There are two methods by which you can create local variables. The primary method is by means of *local variable structures*, which use the program structure word → to create local variables. In addition, the FOR...NEXT/STEP loops described in section 9.5.1 use local variables to store the current values of their loop indices. Although the index variable is used for this special purpose, it is otherwise the same as a local variable created by →, with the same applicable commands and restrictions. In the remainder of this section, we will concentrate on local variable structures.

A *local variable structure* starts with the structure word → (called "arrow," "bind," or just "to") followed by one or more local names, and then by a program or an algebraic object referred to as the *defining procedure*. The closing delimiter (' or >>) that ends the defining procedure also marks the end of the structure:

$$\rightarrow\ name_1\ \ name_2\ \ \cdots\ \ name_n\ \ \ll program \gg, \quad \text{or}$$

$$\rightarrow\ name_1\ \ name_2\ \ \cdots\ \ name_n\ \ 'algebraic'.$$

The user-defined functions described in section 8.5 are a special case of local variable structures. A user-defined function is a program containing one local variable structure, with no additional objects before the → or after the defining procedure.

The primary purpose of local variables is to provide a means of manipulating by name the stack arguments used by a procedure. You can think of the → as meaning "take objects from the stack and give them the following names; then evaluate a procedure defined using the names." Note that the procedure *is* evaluated, even though it is entered between quote delimiters ' ' or << >>.

→ takes objects from the stack and matches them each with one of the names that follows the →. The number of objects taken is determined by the number of names that are specified. The end of the series of names is marked by the delimiter ' or << that starts the defining procedure. The objects are matched in the order in which they appear in the stack; the object in the highest stack level goes with the first name; the object in level 1 is matched with the last name. A local variable is created for each of the names, with the local name as its variable name, and the matching object as its value. For example,

```
1 2 3 4 → a b c d
```

creates the local variables a with the value 1, b with value 2, c with value 3, and d with value 4.

▪ *Example.* Compute the five integer powers $x$ through $x^5$ of a number $x$ in level 1. This first method does not use any variables except a loop index:

```
«  2  5                         Powers 2 through 5.
  FOR  n                        Loop with index n.
    n  1  -  PICK               Get a copy of the number.
    n  ^                        Raise to the nth power.
  NEXT
»
```

This is not a very complicated program. It is fast and efficient, because it uses only stack operations to obtain copies of the input number. The sequence n 1 - PICK is needed to return a new copy each time around because when the index is $n$, the original number has been pushed to level $n-1$ by the growing stack of computed powers.

The program looks easy to write, but you do need a little thought to figure out where the input number will be on the stack at each iteration, and what stack operations are required to return a copy of the number. You can avoid the mental gymnastics by writing the program to remove the number from the stack at the outset, and name it with a local name:

```
«  →  x                         Store the number as x.
  «  x  2  5                    Powers 1 through 5.
    FOR  n                      Loop with index n.
      x  n  ^                   Compute x^n.
    NEXT                        Repeat.
  »
»
```

The latter program is slightly longer than the previous version, but the time it takes you to write it should be less because there is no effort required to keep track of the input number on the stack. Any time the program needs the number, it just executes the local name. The lesson of this simple example becomes more important as the complexity of the programmed calculation increases, to the point where using local variables can make the difference between success and failure in the development of a program.

You can use local variable structures at any point in a program, not just at the beginning as in the case of user-defined functions. The program CINT illustrates the use of a local variable to name an *intermediate* result. CINT computes the radius of a circle inscribed in a triangle, where the lengths of the sides of the triangle are specified on the stack. The formula is:

$$r = \frac{[s(s-a)(s-b)(s-c)]^{½}}{s}$$

where $a$, $b$, and $c$ are the lengths of the sides, and $s = ½(a + b + c)$.

| CINT | | *Circle in a Triangle* | | | 3EBE |
|---|---|---|---|---|---|
| | *level 3* | *level 2* | *level 1* | \| | *level 1* |
| | $a$ | $b$ | $c$ | ☞ | $r$ |

| | |
|---|---|
| `« → a b c` | Name the lengths of the sides. |
| `  « '(a+b+c)/2' EVAL → s` | Compute and save $s$. |
| `    '√(s*(s-a)*(s-b)*(s-c))/s'` | Compute $r$. |
| `  »` | End of local variable structure. |
| `»` | |

There are numerous additional examples of the use of local variables in programs throughout this book. In the remainder of this section, we will review some of the idiosyncrasies of local names and variables, and local variable structures.

## 9.7.1 Comparison of Local and Global Variables and Names

Local names and variables are very similar to ordinary names and variables, but there are some important differences:

- *Global* variables are "permanent," remaining in user memory until you explicitly purge them. *Local* variables are stored in dynamically created *local memories*, which are segments of memory associated with individual procedures. When a procedure has finished evaluation, its local memory (if it has one) is deleted, including all of its local variables.
- *Local* names are a different object type (7) from *global* names (6). This is how the HP 48 system knows whether to find the variable corresponding to the name in VAR memory (global variables) or in a temporary local memory. When the HP 48 attempts to find a local variable, it searches the most recently created local memory first, then previous ones in reverse chronological order, until it finds a local variable matching the specified name.

- Executing a local name recalls to level 1 the object stored in the corresponding local variable, *without* executing the object. This means that when you store a program in a local variable, to execute that program you must execute the variable name and then the recalled program separately, usually with EVAL (or →NUM). The EVAL is not necessary for programs stored in *global* variables, since execution of a *global* name automatically executes the stored object.
- ISOL, QUAD, and TAYLR, which are designed to work with *formal* global variables (names with no associated variables) do *not* accept local names as arguments. Also, the independent variable used for plotting (DRAW) and solving (ROOT) must be specified with a global name.
- You can not delete a local variable with PURGE.
- *Local* names can be the same as HP48 command names (except for single-character algebraic operator names like +, −, *, etc.). Notice that you can have local names i and e, but you should be careful not to use these names when you also want to use the symbolic constants i and e.

Occasionally you may encounter a local name for which the associated local variable no longer exists. This is not a problem for global names, because of their role as formal variables (section 3.6.1). However, executing a local name with no local variable is an error. For example, a defining procedure may leave the name of a local variable on the stack after it completes evaluation.

```
<< 1 → x << 'x' >> >>
```

leaves the local name 'x' on the stack after evaluation, but the corresponding local variable x that was given the value 1 is gone. You can not successfully execute this "formal local variable"--EVAL returns the Undefined Local Name error. You should try to avoid leaving left-over local names on the stack or in algebraic objects that result from symbolic calculations, to avoid confusion later.

## 9.8 Local Name Resolution

The general topic of name resolution was discussed in section 5.5. However, there are a few details that are worth adding now in light of the more extensive treatment of local names in the preceding sections. When ENTER processes a name in the command line, it normally interprets the name as a global name. However, if the name follows a FOR or an →, then ENTER treats the name as a local name while it is handling the rest of the structure that follows. After the subsequent >>, ', or NEXT that terminates the structure, further instances of the same names are again interpreted as global names. Thus in

```
<< → X << X >> X >>,
```

the X in the inner program (<< X >>) is a *local* name, but the final X is a *global* name. To help you keep track of which names are which type, we recommend that you adopt a naming convention, such as using lower-case letters for local names, and upper-case letters for global names. The above program then looks like this:

```
<< → x << x >> X >>,
```

making it clear that the global X is not to be confused with the two local x's. We will follow this convention in this book, except in certain examples in this section where we are illustrating possible confusions between global and local names.

The resolution of names as global or local can be complicated when you nest local variable structures. "Inner" structures can access the local variables of the "outer" structures that contain them, but not vice-versa. For example,

```
1 → x << 2 → y << x y + >> x + y + >>
```

returns '4+y' (not 6), as follows:

| | |
|---|---|
| `1 → x` | Store 1 in local variable x. |
| `  <<` | Start of program in which x is recognized. |
| `    2 → y` | Store 2 in local variable y. |
| `    <<` | Start of program in which y is recognized. |
| `      x y +` | Add x from "outer" program to y from "inner" program, returning 3. |
| `    >>` | End of inner program where y is recognized. |
| `    x +` | Add x to 3, returning 4 |
| `    y +` | This y is *not* a local name, because it is outside of the program where y is local. It therefore names a global variable, which we are here assuming to have no current value. The sum is therefore '4+y'. |
| `  >>` | End of outer program where x is a local variable. |

If you rewrite the above sequence as

```
1 → x << 2 → y << x y + x + y + >> >>,
```

moving the final y back inside the program where the local variable y is defined, the sequence then returns the value 6.

When two nested local variable structures define local variables with the same name, two separate local variables are created. Any use of the name refers to the most recently created local variable. The fact that there is another local variable with the same name in a previously created local memory does not matter. Thus

```
1 → x << 2 → x << x >> >>
```

returns 2, whereas

```
1 → x << 2 → x << >> x >>
```

returns 1.

It is important to note that a procedure represented by a *name* (rather than the procedure itself) within a local variable structure can not access the local variables defined by that structure (unless the procedure is created while the structure is evaluating or suspended; see below). For example, if you create the program A:

```
<< x y + >> 'A' STO,
```

and invoke it in another program like this:

```
<< 1 2 → x y << A >> >>,
```

then executing the latter program returns 'x+y' (global x and y), not 3. When you enter the program A, x and y are created as global names. The search for their values when A is executed in the second program therefore is made in VAR memory, even though there are identically named local variables at the time of the search.

This property of local variables, which makes it possible for each program to define its own variables without name conflicts with those of other programs, has the disadvantage that you can't always easily break a program containing a local variable structure into smaller programs. For example, you can't rewrite

<< → x y << *sequence*$_1$ *sequence*$_2$ >> >>

as two programs

<< *sequence*$_1$ >> 'SEQ1' STO

<< → x y << SEQ1 *sequence*$_2$ >> >>,

if *sequence*$_1$ contains either of the names x or y. There are several approaches that you can use instead:

- Use global variables. Rewrite the second program as

  ```
  << 'y' STO 'x' STO SEQ1 sequence₂ { x y } PURGE >>.
  ```

  In this case you might keep the lower-case names x and y for the global variables, to avoid editing *sequence*$_1$ and *sequence*$_2$.

- Use the stack to pass the values from one program to the other. Rewrite the programs as:

  ```
  << → x y << sequence₁ >> >> 'SEQ1' STO

  << → x y << x y SEQ1 sequence₂ >> >>
  ```

  The latter program puts the values of x and y back on the stack, where SEQ1 can store them in its own local variables x and y. This approach requires no change to *sequence*$_1$.

- Force x and y in SEQ1 to be created as local variables. You can achieve this by entering the SEQ1 program while there is an existing local memory containing local variables x and y.

  1. Type

     ```
     0 0 → x y << HALT >> [ENTER]
     ```

     You will see the suspended program annunciator turn on. Because the local variable structure is executing when the program halts, the local memory containing local variables x and y is still present.

  2. Enter the program SEQ1:

     ```
     << sequence₁ >> 'SEQ1' STO.
     ```

     All instances of x and y in *sequence*$_1$ are treated as local names.

  3. Now, when you execute the main program

     ```
     << → x y << SEQ1 sequence₂ >> >>,
     ```

     execution of the names x and y in SEQ1 returns the values stored at the start of the main program.

     This method, although it solves the problem with no rewriting, is dangerous because if you later edit SEQ1, you must remember to create again the halted program local memory. Otherwise, the command line reentry converts x and y back into global names. Also, you won't be able to use SEQ1 as a subroutine for other programs unless those programs also define local variables x and y.

## 9.8.1 Resolution Speed

Because typical procedures use relatively few local variables compared to the number of global variables that might be in the current path, local name resolution is often significantly faster than that of global names. This speed difference can be important when you have, for example, a program loop that executes at each iteration a global name that resolves to a global variable in the home directory, which might be several levels above the current directory. In such cases, you may find you can improve the program's performance by having it recall the object in the global variable at the outset, and storing it in a local variable. Then all uses of the global name within the program should be replaced by the local name (with EVAL if needed).

Local variables are also preferable to global variables for temporary result storage for performance reasons as well as because of their automatic deletion. When you store an object in a global variable, room must be made for the variable in user memory by moving some or all of the current variables. The time it takes for this is roughly proportional to the total memory size of existing global variables, which can be as much as a second or more when user memory exceeds 100 Kbytes. By contrast, storing an object in a local variable takes on the order of .01 seconds.

# 10. Display Operations and Graphics

In mechanical terms, the HP 48 display is a liquid-crystal display (LCD), containing a matrix of square picture elements, or *pixels*. The pixels are arranged in 131 horizontal rows and 64 vertical columns. The individual pixels can be in two states, which we will call *light* and *dark*, or *off* and *on*. A blank display has all pixels off; turning various pixels on forms characters and other patterns that make up the information content of the display.

The logical capability of the HP 48 display goes well beyond its simple mechanical description. The HP 48 CPU has the ability to deal with display information up to 2048 pixels wide, and indefinitely high, so that the pictures you can create on the HP 48 are not limited to the ordinary LCD dimensions. You can observe this capability when you use the EquationWriter; if a formula display becomes too large for the LCD, you can use the cursor keys to *scroll* the picture around in the display. Since the picture moves to the left when you press the right cursor arrow, the appropriate model you can visualize is that the physical display is a "window" through which you can view the picture. Pressing a cursor key moves the window in the indicated direction.

Since the logical size of the HP 48 display is not fixed, the calculator does not have memory specifically dedicated to the display. Rather, display memory is allocated from ordinary RAM, sharing that memory with the stacks, user memory, and all of the other memory-consuming HP 48 systems. The maximum size of the pictures you can display thus depends on the amount of current free memory, at (roughly) 1 bit of memory per display pixel. By *picture* we mean the visual image represented by a pattern of pixels, as distinguished from the actual pixels or display.

Furthermore, the HP 48 actually defines three separate memory regions for display purposes. We will call these regions *screens*, deriving from their roles as media upon which you can show various pictures. The screens are:

- The *menu screen*, which is permanently allocated memory for the menu labels, 131 pixels wide by 7 pixels high.
- The *text screen* is an expandable memory region a minimum of 131×57 pixels in size. The text screen is not limited to the display of text, but it is most commonly used for displaying the stack and status information, in its minimum size configuration. However, the text screen is also used by the EquationWriter, for which it expands as needed to accommodate the EquationWriter pictures. When you exit from the EquationWriter, the text screen automatically collapses back to its default size.
- The *graph screen* is used by the plotting system and for program graphics. It does not exist until needed by any plotting operation, when it is created if necessary with

a size of 131×64 or larger. If you check free memory with MEM before and after viewing the graph screen for the first time, you will find that free memory has decreased by over 1000 bytes; that is the memory assigned for the graph screen. Be aware that the graph screen is deleted by a system halt (section 5.8); you may want to save its contents in a variable before doing anything that requires a system halt, such as storing a library in a port or inserting or removing a memory card.

In addition to the three dedicated display screens, the HP 48 also provides for storing and manipulating an indefinite number of pictures as *graphics objects* (section 3.4.7). The three screens are actually specially stored graphics objects. The HP 48 has a number of operations for creating graphics objects and displaying them on its screens. In the remainder of this chapter we will study the programmable commands that are available for prompting and presenting graphical and textual information.

## 10.1 Controlling the Display

Ordinarily, after completing any current and pending operations, the HP 48 reverts to its *standard display*, which consists of the simultaneous display of the text screen and the menu screen. Here the text screen is divided into two regions: the status area at the top, and the stack area that is shared by the stack display and the command line. Frequently, however, you can see the standard display superseded temporarily or indefinitely by some form of special display. Such displays ranging from the use of the status area to show error messages, which persist only until the next key press, to a change an environment like the plot environment, which remains with its own display you deliberately exit the environment. Environments may also use their own menus. This ability to supplant the standard display is also available to programs through various display commands.

The most frequent manual display change is switching between the text screen and the graph screen. To activate the graph screen from the standard display, you execute GRAPH, usually by pressing [↰] [GRAPH] --or just [◁] when no command line is present. GRAPH displays the graph screen with the menu screen superimposed upon it (with the plot environment menu). You can switch the menu screen on and off by pressing [+] and [−] ; or by pressing [↰] [GRAPH] , which also allows you to scroll the display window around on the graph screen if it is larger than 131×64. To return to the standard display, press [ATTN] .

Executing GRAPH from a program activates the plot environment while suspending further program execution. When you next press [ATTN] , the text screen is redisplayed (the plot menu remains in the menu screen) and the program resumes execution. If you returned any data from the graph screen to the stack, such as coordinates, graphic objects, or solved results from the ≡FCN≡ menu, that data is then available for the program's use as it resumes execution.

You may also wish to make the graph screen visible while a program is running, but without activating the plot environment. This is accomplished with PVIEW (*Plot VIEW*). PVIEW requires an argument that specifies the position of the screen relative to the display; in particular, you must enter the coordinates of the pixel in the graph screen that you want displayed in the upper left corner of the display. The coordinates may be expressed as a list of two binary integers { *#m #n* }, or as a complex number (*x*,*y*) that specifies a point in logical coordinates (section 10.3.4). PVIEW allows you to watch the graph screen while you change it, to help you monitor the progress of an ongoing plot, or to present any kind of varying graphics display (see, for example, the program GSAMP listed in section 10.3.1). The graph screen remains visible until the program ends, or until you execute TEXT. This command returns the text and menu screens to the display. Note, however, that TEXT does not try to display the current stack contents--it merely redisplays whatever was on the text screen at the point when PVIEW was executed.

As one more alternative, you can execute PVIEW with an empty list as its argument. In that case, PVIEW is equivalent to executing GRAPH followed by pressing [◁] [GRAPH] immediately. Program execution is suspended, and the graph screen is displayed without the plot menu or the cursor--the cursor keys scroll the entire window. { } PVIEW is useful when you want to display a picture that is larger than the display, but you don't need any of the interactive plotting facilities. Again, when you press [ATTN] to exit from the graph screen display, program execution resumes normally.

## 10.1.1 Postponing the Standard Display

While a program is running, it can use display commands to show special text or pictures. However, once the program finishes, the standard display takes over unless the program specifically prevents it. For example, if you execute

"Hi There!" 1 [PRG] ≣DSPL≣ [◁] [PREV] ≣DISP≣ ,

you will see "Hi There!" displayed in the top line of the display. However, if you press [▷] [ENTRY] to turn on program entry mode, then execute "Hi There!" 1 DISP [ENTER] , the text is only flashed momentarily and then is replaced by the standard status display. The difference here is that the ≣DISP≣ *key* executes the command DISP then "freezes" the display. The current picture is not replaced by the standard display until you press another key. The command DISP itself does not include the freeze step. To keep a special display visible after a program stops, you must use an additional command, appropriately named FREEZE. (The ≣→LCD≣ and ≣CLLCD≣ keys also have an automatic freeze built into their definitions, while the programmable commands do not.)

For the purposes of FREEZE, the three nominal areas of the HP 48 standard display are numbered with powers of two: 1 for the status area, 2 for the stack area, and 4 for the menu labels. To freeze one display area, execute FREEZE with a real number argument equal to the desired display area number, e.g. 2 FREEZE preserves the stack area display while the status and menu areas are updated. To freeze more than one area, FREEZE's argument is the sum of the display area numbers (hence the use of powers of two): 3 FREEZE freezes the status and stack areas; 5 FREEZE affects the status and menu areas; and so forth, up to 7 FREEZE, which freezes the entire display.

## 10.2 Text Displays

One of the most common program display tasks is to show one or more lines of text. This is accomplished by means of DISP, which displays text in the medium font in any of the top seven display lines. Here's a simple example:

| CHARDISP | *Display HP 48 Characters* | D423 |
|---|---|---|

```
« CLLCD                               Clear the status and stack areas.
  0 11                                Need a total of 12 lines.
  FOR i ""                            Initialize each text line.
    0 21                              22 characters per line.
    FOR j
      '22*i+j' →NUM                   Next character number.
      CHR + DUP                       Add the character to the line string.
      '(i MOD 7)+1' →NUM DISP         Display in the current line.
    NEXT DROP
  NEXT
»
```

There are several things to notice in this program:

- CHARDISP starts with CLLCD. This command blanks the status and stack areas. You might omit this command from the program if you want to see how DISP overwrites the existing (standard) display.
- DISP takes two arguments: a string from level 2 and a real number from level 1, where the latter can be from 1 to 7 (hence the (i MOD 7)+1) to indicate the desired display line. In the medium font, the display has eight lines; DISP can display in any of the top seven but will not overwrite the menu labels in line 8.

- DISP displays an entire line at once, starting at the edge; you can not use it to display part of a line. If the string argument is shorter than 22 characters, the remainder of the display line is blanked.
- When CHARDISP starts, you see 10 " ▪ " characters displayed in line 1, then the next characters appear in line 2. This is because character 10 is the newline character. You can use DISP to display multi-line messages by including one or more newlines in the display string. The displayed text will start on the line specified by DISP's number argument, and jump to the next line below after each newline character in the string argument. Without newlines, only the first 21 characters of strings longer than 22 characters can be displayed, with ellipses "..." in the rightmost character position to indicate missing characters.
- CHARDISP does not include FREEZE, so the character display disappears as soon as the program is finished.

The string manipulation commands described in section 3.4.3 are the basic tools for creating text displays. For example, a very common task is creating a display string from an object and text that labels the object. The program OLABEL below illustrates this process. OLABEL displays an object (taken from level 2) by converting the object into a string, and appending it (with an "=") to a string provided in level 1. If the label plus object does not fit in a single line, then the label and object are displayed on separate lines. A copy of the object is left in level 1.

| OLABEL | *Output Labeling Utility* | | | E3C1 |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | *object* | *"label"* | ☞ | *object* |

| | |
|---|---|
| `<< "=" +` | Append " = " to the label. |
| `OVER DUP2` | Copy the string and object. |
| `+` | Append the object string to the label string. |
| `IF DUP SIZE 22 >` | If the string is too long, |
| `THEN DROP SWAP` | |
| `10 CHR + SWAP +` | then insert a newline. |
| `ELSE 3 ROLLD DROP2` | Otherwise, discard the extra copies. |
| `END` | |
| `CLLCD 1 DISP` | Clear the LCD and display the string. |
| `>>` | |

You may want to include FREEZE at the end of OLABEL to preserve the object display.

## 10.3 Graphics Displays

To go beyond simple, line/character-oriented text displays, or to use the small and large character fonts, you must create *graphics displays*. Here the key element is the *graphics object*, or *grob*, which is the building block of graphics displays, analogous to string objects for character displays. The HP 48's text and graph screens provide the viewing mechanisms for graphics objects. For simple prompt and information displays, you will most likely use the text screen, so that normal calculator keyboard operations are available. Also, using the text screen for temporary graphics displays does not disturb a plot or other picture currently on the graph screen.

The primary vehicle for viewing graphics on the text screen is the command →LCD. →LCD stores a grob into the top 56 pixel rows of the text screen, with the upper left corner of the grob in the upper left corner of the screen. If the grob is smaller in either dimension than 131×56, the remainder of the screen (other than the menu area, which is not affected by →LCD) is blank. If it is larger than 131×56, only the upper left 131×56 portion of the grob is used. Several examples of using →LCD are given in the next section.

The counterpart of →LCD is LCD→, which returns the current combined text and menu screen picture as a 131×64 graphics object. Notice that the LCD→ grob includes the menu labels, even though →LCD does not overwrite the menu label display area. However, you can use SUB to extract the menu label picture from the LCD→ grob for other purposes, and you can view the entire grob by displaying it on the graph screen.

### 10.3.1 Graphics Object Operations

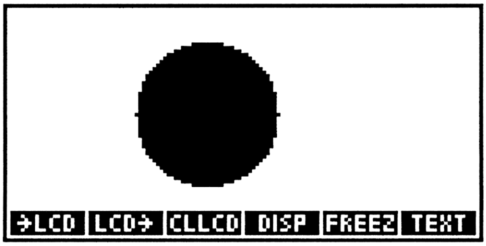
Graphics objects are the object representations of display pictures. They are characterized by their dimensions *width* × *height*, measured in pixels, and by the pixel data that they contain. An individual pixel or position within a grob is specified by coordinates expressed as a list of two binary integers: { #*n* #*m* }, where *n* is the column number, counting right from column 0 at the left edge; and *m* is the row number, counting down from row 0 at the top edge. These binary integers are interpreted as 20-bit signed integers, so that only the least-significant 20 bits are used, and a number #*n* greater than #80000h represents a negative number with absolute value #100000h−#n. (Negative coordinates may be used with line, box, and arc drawing--see section 10.3.5.1).

HP48 commands that apply to graphics objects are found in the third page of the program display menu ( PRG ≡DSPL≡ ), plus SIZE from the third page of the program object menu ( PRG ≡OBJ≡ ), and + and NEG ( +/− ) from the main keyboard. To illustrate the use of these commands, it is helpful to make two sample graphics objects, which is accomplished by the program GSAMP. The program stores a grob containing a filled circle in the variable SPOT, and another with a filled square in GBOX.

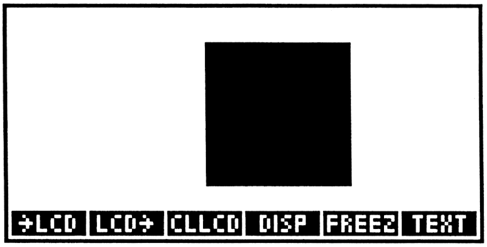
| GSAMP | *Graphics Samples* | A694 |
|---|---|---|

| Program | Comments |
|---|---|
| `« 'PPAR' PURGE` | |
| `ERASE { #0 #0 } PVIEW` | Initialize and view the graph screen. |
| `-3 1` | Draw a filled circle as a series of lines: |
| `FOR x` | |
| `'(x,√(4-SQ(x+1)))'` | |
| `→NUM DUP CONJ LINE` | |
| `.1` | |
| `STEP` | |
| `PICT {#0, #3} {#131d #56d} SUB` | Store the picture in SPOT. |
| `'SPOT' STO` | |
| `ERASE` | |
| `-1 3` | |
| `FOR x` | Now draw a filled box: |
| `'(x,2)' →NUM DUP CONJ LINE` | |
| `.1` | |
| `STEP` | |
| `PICT {#0, #3} {#131d #56d} SUB` | |
| `'GBOX' STO` | Store this picture in GBOX. |
| `»` | |

After executing GSAMP, you can try out the various graphics commands, starting by looking at the grobs made by GSAMP. SPOT ≡→LCD≡ yields this picture:



And GBOX ≡→LCD≡ shows the other picture:



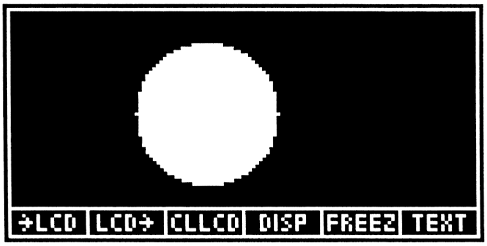
Here if you execute →LCD by means of its menu key, the grob display remains visible until the next keystroke.

• SIZE returns the dimensions of a graphics object as two binary integers, with the width in level 2 and the height in level 1:

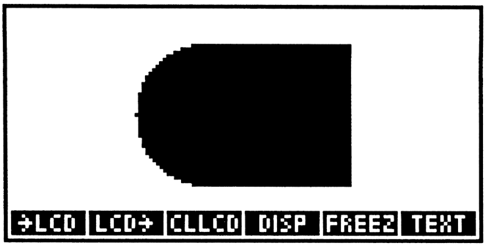
SPOT SIZE ☞ #131d #56d.

• BLANK creates new blank grobs, taking as arguments two binary integers that specify in pixels the width (level 2) and height. #20d #30d BLANK makes a grob 20 pixels wide by 30 pixels high.

• NEG inverts all of a grob's pixels, turning dark into light and vice versa. For example, SPOT NEG →LCD shows:



• + "adds" two grobs together. Specifically, + combines two grobs of the same dimensions into a new grob also of that size, where the result has all pixels turned on that were turned on in either of the original grobs. In effect, one picture is superposed on the other. Thus SPOT GBOX + →LCD yields:

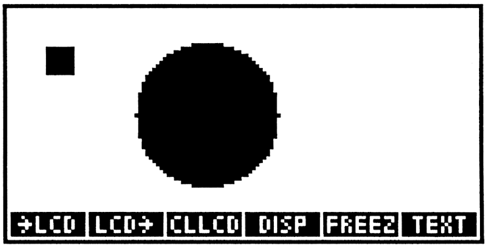


• GOR (*Graphics OR*) is a generalized form of + for graphics objects, for which the two argument grobs do not have to be the same size. Its name derives from logical OR, which returns *true* if either of two arguments are *true*, and *false* otherwise. GOR works as follows:

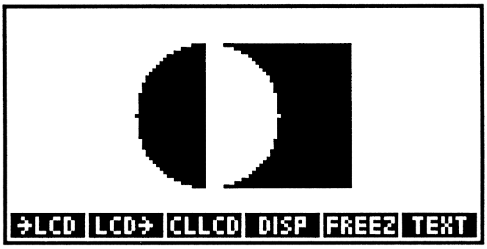
*grob*$_1$ { *#m* *#n* } *grob*$_2$ GOR ☞ *grob*$_3$

where *grob*$_2$ is superposed onto *grob*$_1$, with the upper-left corner of *grob*$_2$ positioned at the { *#m* *#n* } pixel in *grob*$_1$ (you can also use a complex number to represent the pixel position--see section 10.3.4). The result *grob*$_3$ is the same size as *grob*$_1$; any portions of *grob*$_2$ that do not fit within the dimensions of *grob*$_1$ are clipped off. Example:

SPOT { #10d #10d } #8 #8 BLANK NEG GOR →LCD ☞



• GXOR (*Graphics eXclusive OR*) is modeled upon logical XOR, which returns *true* if either of two arguments is *true*, and *false* if both are *true* or both are *false*. For graphics objects, the result picture is a superposition of the argument grobs, except that it will be light where dark regions from both arguments overlap. GXOR's argument order is the same as GOR's; for example, SPOT { #0 #0 } GBOX GXOR →LCD ☞



An important use of GXOR is for placing temporary visible marks (such as a cursor) on a picture that you can easily remove later. That is,

$$grob_1 \quad \{ \#m \quad \#n \} \quad grob_2 \quad \text{GXOR}$$

puts a mark represented by $grob_2$ on $grob_1$; then with the result still on the stack,

$$\{ \#m \quad \#n \} \quad grob_2 \quad \text{GXOR}$$

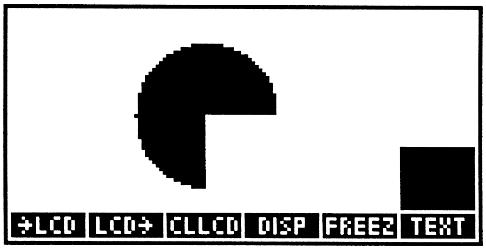
removes the mark and restores $grob_1$. You can observe the action of GXOR by executing the following program:

| AGXOR | *Animate with* GXOR | 0D7A |
|---|---|---|

| | |
|---|---|
| « SPOT PICT STO | Store the spot on the graph screen. |
| # Ah # Ah BLANK NEG → s | Make a black square. |
| « { # 0h # 0h } PVIEW | View the graph screen. |
| 0 130 | |
| FOR x PICT x R→B #14h 2 →LIST | For each *x* position: |
| s 3 DUPN GXOR GXOR | Turn the square on and off. |
| NEXT | |
| » | |
| » | |

• REPL provides a third method of combining two graphics objects, using the same arguments as GOR and GXOR. In this case a region in $grob_1$ starting from { $\#m$ $\#n$ } is *replaced* by $grob_2$. Thus

SPOT { #55d #29d } GBOX REPL →LCD ☞



• SUB is a counterpart of REPL that allows you to extract a portion of a graphics object as a separate, smaller grob. SUB is useful when you want to trim a grob to a smaller size, or to use part of a grob for building other pictures. SUB takes a grob from level three, and two coordinate lists that specify the pixel positions of the corners of the region to be extracted:

SPOT { #35d #9d } { #75d #49d } SUB

creates a 41×41 grob that contains the black spot from the SPOT grob.

## 10.3.2 Graphical Text

A very useful command for the development of graphical displays is the object-to-grob conversion command →GROB. Not only does this command simplify converting objects to graphical text, but it gives you access to all three display fonts, plus the Equation-Writer display.

→GROB requires two arguments: from level 2, the object to be imaged, and from level one a real integer from 0 to 3 to specify the display font. For fonts 1, 2, and 3, the object picture is a one-line text string like that obtained in a single line stack display, respecting the real number and binary integer display modes, and the coordinate mode for complex numbers and vectors. Unlike a stack display, however, the →GROB result is not truncated at the display width--this is because the grob may be intended for display on the graph screen, which can be up to 2048 pixels wide.

Font numbers 1, 2, and 3 represent the small (variable width × 6 pixels ), medium (6×8), and large (6×10) character fonts, respectively. (The width of a character cell given here includes the blank column at the right edge of a character that separates successive characters). Font 0 is intended for algebraic and unit objects, for which →GROB's results are the EquationWriter pictures of the objects (for other object types, font 0 is the same as font 3). Since the EquationWriter uses the active display to build its picture, you will see the EquationWriter "in action" during 0 →GROB execution, and the display is blanked afterwards. Also, the grob returned is always at least 131×64, so you may wish to trim the grob to a smaller size by using SUB.

A nice example of the use of →GROB is provided by the program MINISTK listed below. This program is handy when you want to view more than four stack levels simultaneously. It uses the small font (1) to display up to nine stack objects in single line format. If you store « DROP MINISTK » in the global variable βENTER, and set flags −62 and −63, then the HP48 will use MINISTK in lieu of the normal stack display after every ENTER (see section 7.4).

| MINISTK | *Small-font Stack Display* | F76F |
|---|---|---|

```
«  #131d  #56d  BLANK                      Create a blank display-sized grob.
  DEPTH  1  -  9  MIN                      Make up to nine object grobs.
  IF  DUP                                  If the stack is not empty...
  THEN  #50d  →  y                         Start in row 50.
    «  1  SWAP                             From 1 to depth...
      FOR  n  #0d  y  2  →LIST             Coordinates of level number.
        n  →STR  1  DUP  SUB               Convert the level number to a string.
        ":"  +  1  →GROB  REPL             Add ":", convert to a grob, add to pic-
                                           ture.
        n  1  +  PICK  1  →GROB            Make the nth object into a grob.
        { #0d  #0d }  { #120d  #5d }  SUB  Clip to 121 columns, if necessary.
        #131d  OVER  SIZE  DROP  -         Right-justified position.
        y  2  →LIST  SWAP  REPL            Add the object to the picture.
        'y'  #6d  STO-                     Decrement the vertical position.
      NEXT  →LCD  3  FREEZE                Display the picture.
    »
  ELSE  DROP2                              Do nothing if the stack is empty.
  END
»
```

## 10.3.3 Displays on the Graph Screen

The text screen is adequate for many graphical display purposes. However, you must use the graph screen in the following circumstances:

- You don't want the menu to be visible.
- You want to work with graphics objects larger in either dimension than 131×56.
- You want "animation," or to watch a display continuously as it is being created.
- You want to use any of the automated plotting or drawing facilities.

The graph screen is also more convenient than the text screen, because you can use the pseudo-name PICT to manipulate the graph screen like an ordinary graphics object. PICT is actually a command (type 19), but you can use it in two ways:

1. As a graphics object. PICT can be used as an argument for commands that work with graphics objects: SIZE, SUB, GOR, GXOR, and REPL. For the last three commands, PICT may only be used as the first (level 3) argument. With that argument, the three commands return no result to the stack--the result becomes the new graph screen. Furthermore, there are operations on the PICT grob that are not provided for other grobs: line, box, and arc drawing, and the ability to control and test individual pixels in the grob.
2. As a "variable." Using PICT like a quoted name allows you to treat the graph screen like a variable, where the current value is a grob corresponding to the graph screen picture. Specifically, *grob* PICT STO stores *grob* into the graph screen, replacing the current contents; PICT RCL returns the current contents of the graph screen to the stack as a graphics object, and PICT PURGE deletes the graph screen and recovers the associated memory. Note: you do not need ' ' quotes aroung PICT.

There are several ways to create and dimension the graph screen. Any time you use any plotting or drawing commands, the graph screen is automatically created with a size of 131×64, if it does not already exist. This also occurs if you use GXOR, GOR, or REPL with PICT. To create a new graph screen, you can:

- Store a grob with PICT STO. If you store a grob smaller than 131×64 into the graph screen, it will occupy the upper left corner of the graph screen, with the remainder of the screen blank, but the graph screen will be at least 131×64.
- Execute *#m #n* PDIM. This creates an $m \times n$ graph screen (again with a minimum size of 131×64).

To observe some of these PICT operations in action, try executing the following three programs (which use SPOT and GBOX from section 10.3.1):

ASTO *Animation with* STO 92F5

```
<< SPOT PICT STO                    Store the SPOT grob in PICT.
  { #0 #0 } PVIEW                   View the graph screen.
  1 10                              Repeat 10 times:
  START GBOX PICT STO               View the square.
    SPOT PICT STO                   View the circle.
  NEXT
>>
```

AREPL *Animation with* REPL 5F31

```
<< SPOT PICT STO                    Store the SPOT grob in PICT.
  { #0 #0 } PVIEW                   View the graph screen.
  1 10                              Repeat 10 times:
  START PICT { #0 #0 } GBOX REPL    View the square.
    PICT { #0 #0 } SPOT REPL        View the circle.
  NEXT
>>
```

APVIEW *Animation with* PVIEW EA43

```
<< #131d #128d PDIM                 Create a 131×128 graph screen.
  PICT { #0d #0d } SPOT REPL        Store the circle in the top half.
  PICT { #0d #64d } GBOX REPL       Store the square in the bottom half.
  1 10                              Repeat 10 times:
  START { #0d #0d } PVIEW           View the circle.
  { #0d #64d } PVIEW                View the square.
  NEXT
>>
```

All three programs demonstrate a simple kind of animation on the graph screen, where the picture alternates between a circle and a square. ASTO and AREPL achieve this by changing the actual contents of the graph screen. You can observe that using REPL produces a faster and smoother animation than using STO. This is because STO actually replaces the graph screen grob, whereas REPL merely rewrites the pixels in the existing grob. The "noise" you see between frames in ASTO occurs when the HP 48 is moving the new grob into place, causing the temporary display of random memory bits.

The fastest animation is exhibited by APVIEW, since both frames of the picture are stored in the graph screen in advance. All that is necessary then is to alternate which half of the screen is shown, which can be done quite rapidly. Another variation on this theme is illustrated in the program BOUNCE, where the appearance that the spot is bouncing around the screen is actually achieved by moving the window rather than changing the picture.

| | | |
|---|---|---|
| BOUNCE | *Bouncing Ball Demo* | 4CD8 |

```
« #222d #88d PDIM                  Dimension the graph screen.
  PICT { #56d #14d } SPOT REPL     Put the spot in the center of the
                                   screen.
  #45d #11d #1d DUP
  → x y vx vy                      Initial values for window position
                                   and increments.
  « DO                             Repeat the following:
       x y 2 →LIST PVIEW           View the graph screen.
       vx 'x' STO+                 Increment window x position.
       IF x #91d ==                If at the left edge,
         x #1d == OR               or the right edge,
       THEN 'vx' SNEG              then negate the x increment.
       END
       vy 'y' STO+                 Increment window y position.
       IF y #23d ==                If at the top edge,
         y #0d == OR               or the bottom edge,
       THEN 'vy' SNEG              then negate the y increment.
       END
     UNTIL KEY                     Quit when a key is pressed.
     END DROP                      Discard the key code.
  »
»
```

## 10.3.4 Logical Coordinates

All of the positions within graphics objects that we have specified so far have been expressed as pixel numbers. However, when you refer to positions in the graph screen, you also have the option of using *logical coordinates*. These are floating point numbers derived from a coordinate system imposed upon the graph screen according to the plot parameters in the variable PPAR. The first two elements in the list stored in PPAR are complex numbers $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$, which respectively specify the logical coordinates of the bottom left pixel and the upper right pixel of the graph screen. (Here $x$ represents the horizontal direction, positive to the right, and $y$ the vertical direction, positive upward.)

The conversion between pixel numbers and logical coordinates is as follows. A position $(x,y)$ falls on the $m$-$n$ pixel ({ #$m$ #$n$ }), where

$$m = \mathrm{RND}\left[(M-1)\frac{x - x_{\min}}{x_{\max} - x_{\min}},0\right]$$

$$n = \mathrm{RND}\left[(N-1)\frac{y_{\max} - y}{y_{\max} - y_{\min}},0\right]$$

$M$ is the width of the graph screen in pixels, and $N$ is the height. RND is the HP 48 function RND. Conversely, the $x$-$y$ coordinates center of the $m$-$n$ pixel are:

$$x = x_{\min} + (x_{\max} - x_{\min})\frac{m}{M-1}$$

$$y = y_{\max} - (y_{\max} - y_{\min})\frac{n}{N-1}$$

These formulae are implemented in the commands C→PX (*Coordinates-to-PiXels*) and PX→C (*PiXels-to-Coordinates*). C→PX takes a complex number representing coordinates $(x,y)$ on the graph screen, and returns a list { #$m$ #$n$ } containing the corresponding pixel numbers. PX→C is the inverse, converting pixel coordinates to logical coordinates. These commands are only relevant to the graph screen, or stack grobs that happen to have the same dimensions as the current graph screen. The logical coordinate system is always determined by the values in PPAR, which also are intended for use with the graph screen.

The commands that can accept logical coordinates are GOR, GXOR, REPL, and PVIEW, plus the pixel drawing commands described in the next section. Logical coordinates are often more convenient for mathematical function graphics, whereas pixel coordinates are preferable for making prompting displays and drawing simple geometric figures. Arithmetic with binary integers is also faster than with floating-point complex numbers.

### 10.3.4 Pixel Drawing

The first two pages of the program display menu ( PRG ≡DSPL≡ ) provide several drawing tools for producing simple graphics on the graph screen. These commands do not work with stack grobs; if you want, for example, to draw a line in any grob you must first store it into the graph screen.

The most basic tools are commands that turn individual pixels on and off. PIXON turns on the pixel specified by its coordinates, entered either as logical coordinates (complex number) or pixel coordinates. As an example of using PIXON, the program DRAWPIX imitates the command DRAW. As listed, DRAWPIX is only a slower substitute for DRAW, but you can use it as a starting point for creating modified plotting programs to obtain results you can't get with DRAW.

| DRAWPIX | *DRAW using PIXEL* | 2641 |
|---|---|---|

```
≪ PICT  SIZE  #64d  -  2  /
  SWAP  #131d  -  2  /  SWAP
  2  →LIST  PVIEW                    View the center of the graph screen.
  PPAR  1  GET  RE                   Get x_min.
  PPAR  2  GET  RE                   Get x_max.
  PPAR  4  GET                       Get the resolution.
  IF  DUP  0  SAME
  THEN  DROP  #1                     Default real case.
  END
  IF  DUP  TYPE                      If it's not a real number,
  THEN
    IF  DUP  #0  SAME
    THEN  DROP  #1                   Default binary case.
    END  B→R  OVER  4  PICK  -
    PICT  SIZE  DROP  B→R  /  *      then compute the step size.
  END
  PPAR  3  GET                       Get the independent variable name x.
  →  step  indep
  ≪  IF  indep  VTYPE  1  +          If the independent variable exists,
    THEN  indep  RCL  3  ROLLD  1    then keep its value and true.
    ELSE  0                          Otherwise false.
    END  3  ROLLD                    Save the flag.
    FOR  x                           Loop from x_min to x_max:
      x  indep  STO                  Store the current value of x in the independent
                                     variable.
      EQ  →NUM                       Evaluate the current equation (y).
      x  SWAP  R→C                   Combine the coordinates into a complex
                                     number.
      PIXON                          Plot the point.
    step  STEP                       Increment x and repeat.
    IF  THEN  indep  STO  END        Restore the original value.
  ≫
≫
```

The counterpart of PIXON is PIXOFF, which turns off a pixel specified by its coordinates. You can also test whether a pixel is currently turned on by executing PIX?, which returns *true* if the specified pixel is on, and *false* if it is off. It is a simple matter also to reverse the state of a pixel, using the program TPIX:

| TPIX | *Toggle a Pixel* | 7259 |
|---|---|---|
| | *level 1* | |
| | { *#m #n* } | ☞ |
| | (*x*,*y*) | ☞ |

```
« DUP                              Copy the coordinates.
  IF  PIX?                         If the pixel is on,
  THEN  PIXOFF                     then turn it off.
  ELSE  PIXON                      Otherwise turn it on.
  END
»
```

LINE and TLINE allow you to draw straight lines much more rapidly than you can using PIXON and PIXOFF. Both require two arguments, which specify the start and end points of a line. The arguments can be either complex numbers or lists of binary integers, but both must be the same type. LINE draws by turning on all of the pixels on a straight line (allowing for the finite size of the pixels) between and including the start and end point. TLINE reverses the pixels along the line, which is useful when you are drawing lines across dark areas. The use of LINE is illustrated in the next two programs. STAR draw a five-pointed star, using the second program SKETCH. The latter takes a list of coordinates and draws lines between each successive pair of points.

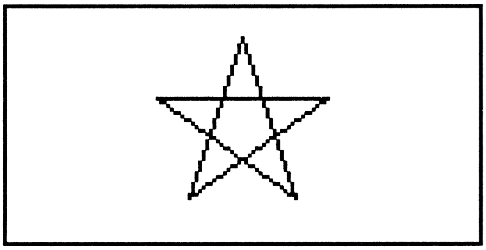
| STAR | *Draw a Star* | F19D |
|---|---|---|

```
«  RCLF  -16  SF  DEG  -19  SF     Polar mode, degrees, V2 complex.
   'PPAR'  PURGE                   Initialize.
   (0,2.5)  DUP                    Start at (0,2.5).
   0  4
   START  V→ 144 + →V2             Rotate by 144°.
      DUP  SWAP                    Add the point to the stack.
   NEXT
   DROP  6  →LIST                  Combine into a list.
   {#0  #0}  PVIEW                 Omit this if you don't want to watch.
   SKETCH                          Connect the dots.
   STOF                            Restore previous modes.
»
```

| SKETCH | *Sketch Lines* | C1A7 |
|---|---|---|
| | *level 1* | |
| | *{ list of points }* ☞ | |

| | |
|---|---|
| `« → points` | Store the list. |
| `  « 1 points SIZE 1 -` | One fewer lines than points. |
| `    FOR n` | |
| `      points n GETI 3 ROLLD GET` | Get the next pair of points. |
| `      LINE` | Use TLINE to toggle the lines. |
| `    NEXT` | |
| `  »` | |
| `»` | |

Executing STAR yields this picture:



The command BOX provides an easy method for drawing rectangular boxes, specified by two sets of coordinates (pixel or logical). For example, to draw a simple frame around the graph screen, execute FRAME:

| FRAME | *Frame the Graph Screen* | 2C4F |
|---|---|---|

| | |
|---|---|
| `« { #0 #0 }` | Upper-left corner. |
| `  PICT SIZE` | Screen dimensions. |
| `  #1 - SWAP #1 - SWAP 2 →LIST` | Lower-right corner. |
| `  BOX` | Draw the box. |
| `»` | |

The final built-in drawing command is ARC, which draws circular arcs on the graph screens. ARC uses four arguments, either

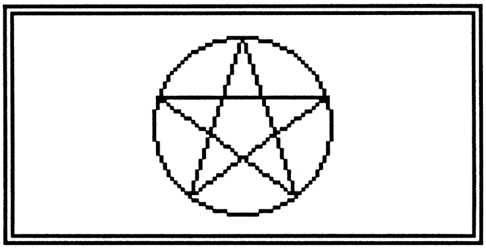
$$(x,y) \quad r \quad \theta_1 \quad \theta_2$$

or

$$\{\ \#x\ \ \#y\ \}\quad \#r \quad \theta_1 \quad \theta_2,$$

where $x$ and $y$ are the coordinates of the center of the arc, expressed either as a complex number or as a list of binary integers; $r$ is the radius in logical coordinates or pixels; and $\theta_1$ and $\theta_2$ are the starting and ending angular positions of the arc. The following sequence uses ARC and the other programs listed in this section to draw a circle around a star, framing the whole graph screen for good measure:

```
STAR  (0,0)  2.5  0  -1  ACOS  2  *  ARC  FRAME  7  FREEZE
```



ARC does not attempt to compensate for differing plot scales in the vertical and horizontal directions--it will not draw an ellipse. It always draws an arc of constant radius *in pixels*. The pixel specified by the coordinates $(x,y)$ + $(r,\measuredangle\theta_1)$ is taken as the starting point of the arc; the distance in pixels from that point to the center pixel $(x,y)$ is then used as the actual radius $r'$, where $r'$ has the same sign as $r$. The arc drawing stops at the pixel specified by $(r', \measuredangle\theta_2)$. Note also that

- If $\theta_1 = \theta_2$, one pixel is turned on, at $(x,y)+(r,\measuredangle\theta_1)$.
- If $|\,\theta_2 - \theta_1\,| > 360°$, then the drawing stops after one full circle is drawn from $\theta_1$.

**10.3.5.1 Off-Screen Coordinates**

The drawing commands PIXON, PIXOFF, PIX?, LINE, TLINE, BOX, and ARC, and the coordinate conversions C→PX and PX→C, all accept coordinate arguments that correspond to pixel positions that do not fall on the current graph screen. This includes *negative* pixel numbers in the range #80000h to #FFFFFh, which represent pixels that are above or to the left of the screen. While you can never view pixels that are off-screen, their coordinates may be useful:

- When you are doing any kind of iterative plotting, you don't have to check each set of coordinates for PIXON or PIXOFF to verify that it falls on the graph screen. The checking is done automatically by the commands, which just do nothing for off-screen pixels. PIX? always returns *false* for off-screen positions.
- You can use LINE, TLINE, BOX, and ARC when their position arguments are off-screen. This allows you to draw parts of figures too large for the screen by drawing the entire figure without regard to off-screen portions. In particular, the center of an arc drawn by ARC does not have to lie within the graph screen, nor do the start and end points of the arc.

LINE, TLINE, and BOX are smart enough to "clip" any portions of lines that are off-screen, so that they will not spend unnecessary time plotting invisible points. ARC is not so enlightened; if you use ARC to draw a circle that only partially fits on the graph screen, ARC takes just as long to execute as it would if the screen were large enough to contain the entire circle.

For all four commands, keep in mind that the dynamic range of pixel coordinates is limited to #80001h-#7FFFFh ($\pm$524287); if you use logical coordinates that correspond to pixel numbers out of this range, the coordinates are truncated to the allowed range. You can see this when you use LINE, for example, to draw a line between two points along the $-45°$ line. With the default plot parameters, arguments of (-100,100) and (100,-100) yield a line through the origin (0,0), as you would expect. But if you increase the coordinates to (-100000,100000) and (100000,-100000), the line is drawn at $-45°$ through pixel {#0 #0}, passing below the logical origin. This is because the larger arguments are truncated to {#80001h #80001h} and {#7FFFFh #7FFFFh}, respectively.

# 11. Arrays and Lists

The HP 48 *array* and *list* object types allow you to deal with collections of numbers or other objects as single units, as well as to access the individual objects in the collections. You are probably familiar with one-dimensional arrays--*vectors*--and two-dimensional arrays--matrices--from mathematics. These are one-dimensional (vectors) or two-dimensional (matrices) ordered sets of numbers that satisfy certain rules of arithmetic and transformation properties. However, you may find the idea of a *list* as a useful computational tool to be a new concept, since other calculator languages and most computer languages have no equivalents. (Lists will be very familiar to you if you have studied LISP, or a similar computer language.) In mathematics the closest counterpart is the *set*, usually a collection of objects with some common property.

## 11.1 Arrays

The most important feature of the HP 48 related to array computation is the calculator's ability to manipulate arrays as self-contained units--as objects. This means, for example, that you can perform array arithmetic on the stack using the same steps and commands as you would for real number arithmetic. Programs can use arrays as input and return arrays as output; the arrays themselves contain all of the dimensional information that the programs need to deal with the data in the arrays. The mathematical operations that the HP 48 provides for matrices and vectors are not remarkable; it is the ease with which you can apply the operations to arrays that is the strength of the HP 48. We will not dwell on the mathematical commands here, since they are described adequately in the owner's manuals. Instead we will focus on the array manipulation commands and methods.

- To assemble a series of numbers on the stack into an array, use →ARRY.

```
1 2 3 4 4 →ARRY ☞ [ 1 2 3 4 ]

1 2 3 4 { 4 } →ARRY ☞ [ 1 2 3 4 ]

                           [ [ 1 2 ]
1 2 3 4 { 2 2 } →ARRY ☞      [ 3 4 ] ].
```

  The level 1 argument of →ARRY determines how many numbers are taken from higher stack levels to form the array, and the dimensions of the array. When the argument is a real number $n$, or a list { $n$ }, then $n$ additional numbers are taken from the stack to form an $n$-element vector. If the argument is a two-element list, e.g. { $n$ $m$ }, $n \cdot m$ numbers are combined into an $n \times m$ matrix. The order in which the array elements are taken from the stack is called *row-order*. This order has element 1 or 1-1 in the highest stack level, followed by the elements of the first row in left-to-right order, then by the row 2 elements, if any, and so forth, ending in level 2 with the last element in row *n*.

- To take an array apart, use OBJ→ (you can also substitute ARRY→). Reversing the previous example:

```
[1  2  3  4]  OBJ→  ☞  1  2  3  4  {4}

[[1 2]
 [3 4]]  OBJ→  ☞  1  2  3  4  {2  2}
```

OBJ→ returns the elements of an array as individual numbers in row order, and leaves the dimension list in level 1. Notice that OBJ→ always returns the dimension(s) in a list, even when its argument is a vector.

- To determine the dimensions of an array, use SIZE.

```
[[1 2]
 [3 4]   SIZE  ☞  {3 2}
 [5 6]]
```

- To extract individual numbers from an array, use GET or GETI (section 5.7.1).

```
[[1 2]
 [3 4]]  {2 1}  GET  ☞  3.
```

- To substitute numbers into an array, use PUT or PUTI (section 5.7.2.3).

```
[[1 2]                            [[1 2]
 [3 4]]  {2 1}  8  PUTI  ☞   [8 4]]  {2 2}
```

the { 2 1 } element in the array is replaced with a new value 8, and the next index is returned.

- To reorganize the elements of an array into an array with different dimensions, while preserving the row order of the elements, use RDM (*ReDiMension*). The arguments for this command are the original array in level 2 and the dimension list for the new array in level 1:

```
                                         [[1 2]
[[1 2 3 4]                                [3 4]
 [5 6 7 8]]  {4  2}  RDM  ☞        [5 6]
                                          [7 8]]
```

If the dimension list { $m$ $n$ } specifies a new array with fewer elements than the original, RDM uses only the first $m \cdot n$ elements of the original and discards the remainder. If the new array requires more elements than the original, the missing elements are filled by zeros.

You can also apply RDM to an array stored in a global or local variable by substituting the variable's name for the argument array. The result array replaces the original array in the variable.

- TRN (*TRaNspose*) replaces a matrix by its (conjugate) transpose, where the matrix can be on the stack itself, or represented by a variable name:

```
                                [[ 1 5 ]
[[ 1 2 3 4 ]     TRN   ☞        [ 2 6 ]
 [ 5 6 7 8 ]]                   [ 3 7 ]
                                [ 4 8 ]]
```

TRN does not work with vectors: if you want to transform a vector into a single-row matrix, use this sequence:

```
OBJ→  1  SWAP  +  →ARRY.
```

- Two commands are available for creating constant arrays. IDN (*IDeNtity*) creates an $n \times n$ identity matrix specified by a real number argument $n$:

```
                [[ 1 0 0 ]
3  IDN  ☞        [ 0 1 0 ]
                 [ 0 0 1 ]]
```

IDN can also change an existing array (on the stack or specified by name) into an identity matrix of the same size. In that case, of course, you don't need to specify the dimension of the matrix. If the initial matrix is complex, the resulting matrix will also be complex, with diagonal elements (1,0).

CON (*CONstant array*) creates an array dimensioned according to a list in level 2, where all of the elements have the same value, specified by a real or complex number in level 1. Like IDN, CON will also use an array (or its name) as its level 2 argument. If the initial array is real, then the new constant value in level 1 must also be real. For an initial complex array, the constant value can be real or complex; for a real number $x$, the result array will remain complex, with elements ($x$,0).

The program MINOR listed below illustrates the use of array manipulation commands. MINOR computes the *nm minor* of a matrix, which is defined as the original matrix with its *n*th row and *m*th column removed. Assuming that MINOR should start with the original matrix in level 3, the row number *r* in level 2, and the column number *c* in level 1, we can sketch a preliminary version of MINOR:

```
<< 3 ROLLD DELROW SWAP DELCOLUMN >>
```

DELROW must be a subroutine that removes the *r*th (level 1) row of a matrix (level 2); DELCOLUMN removes the *c*th column. However, you can observe that removing a column of a matrix is the same as removing a row of the transposed matrix, so that a program that removes a row can do both jobs if combined with the transpose command TRN. We choose to work with rows rather than columns because OBJ→ puts elements on the stack in row order, making it easier to delete the elements of a row than a column.

| MINOR | *Minor of a Determinant* | | | | 4C4A |
|---|---|---|---|---|---|
| | *level 3* | *level 2* | *level 1* | \| | *level 1* |
| | [[ *matrix* ]] | *r* | *c* | ☞ | [[ *matrix'* ]] |

| | |
|---|---|
| << 3 ROLLD | \| *c* [[ *matrix* ]] *r* \| |
| DELROW | Remove the *r*th row. |
| TRN SWAP DELROW | Remove the *c* column. |
| TRN | Transpose back again. |
| >> | |

| DELROW | *Delete a Matrix Row* | | | | 0FC6 |
|---|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* | |
| | [[ *matrix* ]] | *n* | ☞ | [[ *matrix'* ]] | |

| | |
|---|---|
| << → r | Store the row number. |
| << OBJ→ | Put the array elements individually on the stack. |
| OBJ→ DROP → n m | Save the dimensions in r and c. |
| << n r - m * →LIST | |
| → s | Save the last $(n-r)m$ elements in a list s. |
| << m DROPN | Discard *m* elements. |
| s OBJ→ DROP | Recover the saved elements. |
| >> | |
| n 1 - m 2 →LIST | The new array has dimensions $(n-1)\times m$. |
| →ARRY | Make the result array. |
| >> | |
| >> | |
| >> | |

Programs in subsequent sections of this chapter contain several additional examples of the uses of array manipulation commands.

## 11.2 Arrays and Algebraic Objects

Although algebraic objects may not directly contain arrays, you can nevertheless use algebraic objects to represent various array operations symbolically. Any name appearing in an algebraic expression can refer to a variable containing arrays. You can apply any symbolic manipulation to such an expression; but when you evaluate an expression, the result can not be symbolic. For example, if variable A contains the vector [ 1 2 ], and B is [ 3 4 ], then evaluation of 'A+B' returns [ 4 6 ]. However, if B is undefined, then evaluation of the expression returns the Bad Argument Type error.

In addition to this ordinary use of names within algebraic expressions to refer to stored arrays, you can use a function-like syntax to access individual array elements. Consider subtracting the second column from the first in a ten-row matrix 'MAT', replacing the first column with the difference. Using GET and PUT explicitly, this is accomplished by

```
1 10 FOR n MAT 1 n GET MAT 2 n GET - MAT 1 n PUT NEXT.
```

The following sequence accomplishes the same thing, but it is rather more readable:

```
1 10 FOR n 'MAT(n,1)-MAT(n,2)' EVAL 'MAT(n,1)' STO NEXT.
```

The general syntax for recalling an *n*th element is *name(n)* for vectors, and *name(m,n)* for matrices, where *name* is the name of a variable containing an array, and *m* and *n* are element numbers. For example, with variable A and B defined as above,

```
'A(1)+B(2)' EVAL ☞ 5.
```

(The vector syntax with a single element number also works when the named variable is a list). Evaluating expressions like these actually invokes GET, e.g. 'M(1,2)' EVAL is equivalent to 'M' { 1 2 } GET when variable M contains a matrix. (If the evaluation fails because the element index is out of range, the error message will specify a GET error.) To execute PUT in a similar manner, you can use an algebraic object as an argument for STO:

*object* 'X(*n*)' STO

stores *object* as the *n*th element of the array or list X. If X contains a list, *object* may be of any type. If X contains a vector or a matrix, *object* must be a number. In the case of a matrix, X can have one or two indices; either

```
25  'X(3)'  STO
```

or

```
25  'X(2,1)'  STO
```

stores 25 into the 2-1 element of a 2×2 matrix stored in X.

The program SUBCOL demonstrates the use of this type of indexing. It replaces a matrix with a new version in which the elements in column *i* have been replaced by their original values minus the corresponding elements in column *j*.

| SUBCOL | *Subtract Columns* | | | | 0AD5 |
|---|---|---|---|---|---|
| *level 3* | *level 2* | *level 1* | \| | *level 1* | |
| [[ *x* ]] | *i* | *j* | ☞ | [[ *x'* ]] | |

```
<<  →  mat i  j                     Store column numbers.
  <<  1  mat  SIZE  1  GET          Number of rows.
    FOR  n
      'mat(n,i)-mat(n,j)'  EVAL     Compute the difference.
      'mat(n,i)'  STO               Replace the value.
    NEXT
    mat                             Return the matrix.
  >>
>>
```

## 11.3 Vectors and Coordinate Systems

HP48 vectors are one-dimensional arrays represented as a series of real or complex numbers enclosed in single brackets [ ]. When a vector is entered or displayed, the elements are shown in a horizontal format suggesting a row (covariant) vector. However, HP48 vectors actually have the mathematical properties of column (*contravariant*) vectors. This means, for example, that an *n*-element vector $\vec{v}$ is conformable for premultiplication ($\mathbf{A}\cdot\vec{v}$) by an $m \times n$ matrix **A**. The vectors are displayed horizontally in order to show as many elements as possible on the display. You can represent row (*covariant*) vectors as $1 \times n$ matrices.

The HP48 provides two commands for computing vector products. DOT computes the dot, or inner, product of two vectors of the same dimension: if $x_i$ and $y_i$ are the $i$th elements of two vectors of size $N$, then the dot product is defined as

$$\sum_{i=1}^{N} x_i \cdot y_i .$$

ABS applied to a vector returns

$$\left| \sum_{i=1}^{N} x_i^2 \right|^{1/2},$$

which is equivalent to the square root of the absolute value of the dot product of the vector with itself. The following program uses ABS to compute the angle between two vectors

| VANGLE | *Angle Between Two Vectors* | | | F518 |
|---|---|---|---|---|
| | *level 2* | *level 1* | | *level 1* |
| | [ $x_i$ ] | [ $y_i$ ] | ☞ | $\theta$ |

| | |
|---|---|
| « DUP2 DOT | $\vec{x}\vec{y}$ |
| SWAP ABS / | $\vec{x}\vec{y}/\lvert\vec{x}\rvert$ |
| SWAP ABS / | $\vec{x}\vec{y}/(\lvert\vec{x}\rvert\ \lvert\vec{y}\rvert)$ |
| ACOS | $\theta$ |
| » | |

For two- and three-dimensional vectors, CROSS computes the cross-product $\vec{z} = \vec{x} \times \vec{y}$ of two vectors, where

$$z_i = \sum_{j,k} x_j y_k \epsilon_{ijk}$$

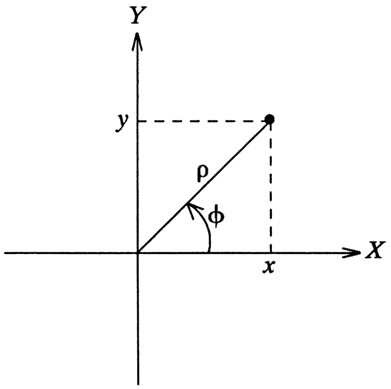
$$\epsilon_{ijk} = \begin{cases} 0 \text{ if } i=j,\ j=k, \text{ or } i=k \\ +1 \text{ if } i,\ j,\ k \text{ are in cyclic order} \\ -1 \text{ otherwise.} \end{cases}$$

CROSS's result is always a three-element vector. A two-element vector used as an argument is treated as a three-element vector. If both arguments are two-element vectors, then the result is a vector of the form [ 0 0 $z$ ].

## 11.3.1 Coordinate Systems

Because two- and three-element real vectors are common in engineering and physics, the HP 48 provides special capabilities associated with this class of objects. In particular, the vectors can be entered, displayed, and analyzed in polar coordinates as well as in rectangular coordinate systems.

In two dimensions, the position of a point is represented by a radial coordinate ρ and a polar angle ϕ:



The conversions between polar coordinates (ρ,ϕ) and rectangular coordinates (*x*,*y*) are given by:

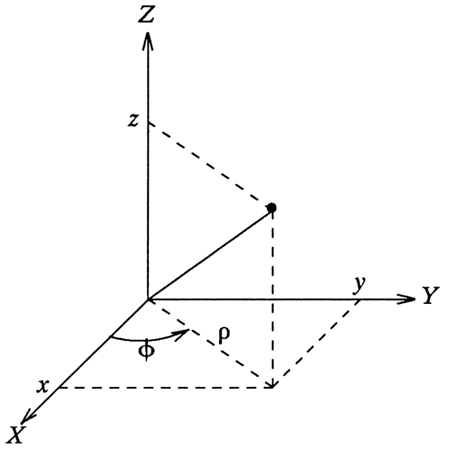
$$\rho = (x^2 + y^2)^{1/2} \qquad x = \rho\cos\phi$$
$$\phi = \tan^{-1}(y/x) \qquad y = \rho\sin\phi$$

In three dimensions, two types of polar coordinates are used, cylindrical and spherical. The conversions between rectangular and cylindrical polar coordinates are the same as for the two-dimensional case, with the *z*-coordinate the same in both systems. For spherical polar coordinates (r,ϕ,θ), the conversions with rectangular coordinates (*x*,*y*,*z*) are

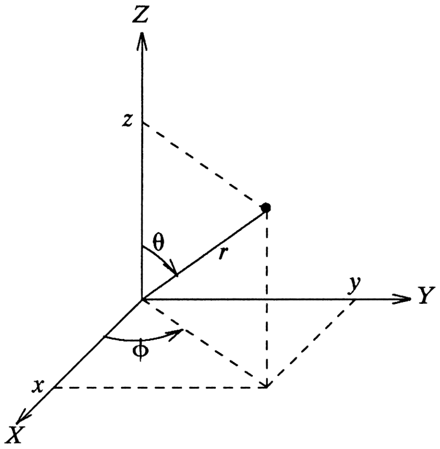
$$r = (x^2 + y^2 + z^2)^{½} \qquad x = r \sin\theta \cos\phi$$

$$\phi = \tan^{-1}(y/x) \qquad y = r \sin\theta \sin\phi$$

$$\theta = \tan^{-1}\frac{(x^2 + y^2)^{½}}{z} \qquad z = r\cos\theta$$



**Cylindrical Polar Coordinates**



**Spherical Polar Coordinates**

HP 48 vectors are always stored in memory in rectangular coordinates. However, you can choose to display vectors in polar form, and to create and take apart vectors using polar values. The display of vectors is controlled by the *coordinate mode*, controlled by flags −15 and −16. You can change modes manually by pressing ⮣ POLAR , or by using one of the menu keys XYZ (rectangular), R∡Z (cylindrical), and R∡∡ (spherical) found on the first page of the MTH VECTR menu and the third page of the ⮢ MODES menu. The ⮣ POLAR key switches back and forth between rectangular mode, and whichever of the two polar modes was last selected by one of the menu keys or by setting or clearing flag −15. The current coordinate mode is indicated in the status area of the display, where the symbol R∡Z means cylindrical polar mode, and R∡∡ means spherical polar mode. If neither symbol is visible, rectangular mode is active.

In either polar mode, the HP 48 displays a two-dimensional vector in the form [ρ, ∡φ], where the angle symbol ∡ indicates that the latter number is to be interpreted as the polar angle. (This discussion also applies to complex numbers, using parentheses (ρ,∡φ) rather than vector delimiters [ρ, ∡φ].) The numerical value of the angle also depends on the current angle mode, which can be degrees, radians, or grads:

```
[⬑] [MODES] ≡STD≡
[NXT] [NXT] ≡DEG≡
≡XYZ≡    [ 1  1 ]    ☞   [ 1  1 ]
≡R∡Z≡                ☞   [ 1.41421356237   ∡45 ]
≡RAD≡                ☞   [ 1.41421356237   ∡.785398163397 ]
≡GRAD≡               ☞   [ 1.41421356237   ∡50 ]
```

You can also enter a two-element vector in polar form by including an angle symbol ∡ in front of the second number, which is interpreted as an angle according to the current angle mode. Notice, however, that a vector entered in polar form may not keep exactly the values that you enter:

```
DEG  [ 25  ∡225 ]  [ENTER]  ☞  [ 25.0000000001   ∡-135 ]
```

This is because the polar coordinates are converted to rectangular coordinates to store the vector, then are converted back to polar form for display. The finite precision conversions may introduce changes in the twelfth decimal place. Also, since ATAN is used in the conversion, the displayed polar angle will always have a value between −180° and +180°.

The fact that all vectors are stored in the same rectangular format regardless of display mode means that they are suitable for various operations without needing any preliminary conversions. You can perform vector arithmetic, for example, directly in polar form:

```
[ 1  ∡45 ]  [ 1  ∡-45 ]  +  ☞  [ 1.41421356237   ∡0 ]
```

All of the properties described for two-dimensional vectors apply as well to three-dimensional vectors, in which the second and third elements may include angle symbols to indicate polar coordinates. If the second element (only) of a three-element vector is entered or displayed with a leading angle symbol, the vector is being represented in cylindrical polar coordinates [ $\rho$ ∡$\phi$ $z$ ]. A vector with the second and third elements starting with angle symbols is interpreted in spherical polar coordinates [ $r$ ∡$\phi$ ∡$\theta$ ].

```
[◁] [MODES]    ≡STD≡
 [NXT] [NXT]   ≡DEG≡
≡XYZ≡   [1  1  1]      ☞   [1   1   1]
≡R∡Z≡                  ☞   [ 1.41421356237   ∡45     1 ]
≡R∡∡≡                  ☞   [ 1.73205080757   ∡45    ∡54.7356103172 ]
```

You can not use the MatrixWriter to enter or edit vectors in polar form; there is no provision for including the angle symbol in any element field. However, you can enter two- or three-element vectors without using [ ] delimiters or angle symbols by using the 2D and 3D operations ([◁] [2D] and [▷] [3D]). Both operations are "toggles" that convert a vector to and from its elements. 2D, for example, converts two real numbers to a vector:

```
≡XYZ≡   1.23   4.56  [◁] [2D]   ☞  [ 1.23   4.56 ]
```

On the other hand, if level one contains a real vector, 2D takes it apart into separate real numbers:

```
[ 1.23   4.56 ]  [◁] [2D]   ☞  1.23   4.56.
```

3D works similarly for three-dimensional vectors. Both 2D and 3D are sensitive to the coordinate mode and angle mode:

```
≡R∡∡≡     ≡DEG≡
1   45   50  [▷] [3D]   ☞   [ 1   ∡45   ∡50 ]

≡R∡Z≡     ≡RAD≡
[▷] [3D]                ☞   .766044443119   .785398163397   .642787609687
```

Actually, when the argument is a vector, 2D and 3D are equivalent, and both will decompose vectors of any size. However, for vectors with more than three elements, only rectangular coordinates are returned.

The operations 2D and 3D are combinations of the programmable commands V→, which takes a vector apart, and →V2 and →V3, which respectively assemble two- and three-element vectors from real numbers. (If there is an error during execution of 2D or 3D, one of these commands will be identified by the error message.) V→ breaks a vector of any size into its component elements, following the current angle and coordinate modes for two- and three-element vectors:

[ 1 1 1 ] R∡∡ DEG V→ ☞ [ 1.73205080757 ∡45 ∡54.7356103172 ]

(Unlike OBJ→ and ARRY→, which also decompose vectors, V→ does not return a dimension list.)

If you are combining individual coordinates into a vector, →V2 and →V3 are equivalent to 2D and 3D, respectively. Of course, you can also use →ARRY, but →V2 and →V3 allow you to express their arguments as polar coordinates, following the current angle and coordinate modes.

### 11.3.2 Example: Coordinate Transformations

Consider two coordinate systems, where the second is derived from the first by 1) displacing the origin by an amount given by the vector $\vec{T}$, and 2) rotating the axes through an angle α about the direction specified by a (unit) vector $\vec{N}$. A vector $\vec{P}'$ expressed in the new system are obtained from the coordinates $\vec{P}$ of the same vector in the original system using the following formula:

$$\vec{P}' = \left[(\vec{P}-\vec{T})\cdot\vec{N}\right](1-\cos\theta)\vec{N} + (\vec{P}-\vec{T})\cos\theta + \left[(\vec{P}-\vec{T})\times\vec{N}\right]\sin\theta.$$

An easy way to render this formula into a program is to store the four arguments $\vec{P}$, $\vec{T}$, $\vec{N}$, and α as local variables, then evaluate an algebraic object matching the formula. This is not immediately possible, however, since DOT and CROSS are not functions in the HP 48 sense. But we can fix that problem by creating user-defined functions (section 8.5) as follows:

| DOTF | *DOT Function* | | | | 60EC |
|---|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* | |
| | $\vec{x}$ | $\vec{y}$ | ☞ | $\vec{x}\cdot\vec{y}$ | |

```
≪ → A B
  ≪ A B DOT
  ≫
≫
```

| CROSSF | *CROSS Function* | | | 4732 |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | $\vec{x}$ | $\vec{y}$ | ☞ | $\vec{x}\times\vec{y}$ |

```
<< → A B
  << A B CROSS
  >>
>>
```

With these two programs in hand, we can write the transformation program:

| XFORM | *Coordinate Transformation* | | | | | 0D94 |
|---|---|---|---|---|---|---|
| | *level 4* | *level 3* | *level 2* | *level 1* | \| | *level 1* |
| | $\vec{P}$ | $\vec{T}$ | $\vec{N}$ | $\theta$ | ☞ | $\vec{P}'$ |

| | |
|---|---|
| << SWAP DUP ABS / | Unit vector. |
| 4 ROLL 4 ROLL - | $\vec{PT} = \vec{P}-\vec{T}$. |
| → α N PT | Save the parameters as local variables. |
| 'DOTF(PT,N)*(1-COS(α))*N | |
| +(PT)*COS(α)+CROSSF(PT,N)*SIN(α)' | Evaluate transformation formula. |
| >> | |

■ *Example*. A coordinate system is translated a distance 3 in the direction specified by the spherical polar angles $\phi=30°$ and $\theta=60°$, then rotated through 45° about the z-axis. What are the coordinates of the vector [1 1 1 ] in the new system?

■ *Solution*. In this problem, $\vec{P}$ = [ 1 1 1 ], $\vec{T}$ = [3 ∡30 ∡60 ], $\vec{N}$ = [0 0 1], and $\alpha$ = 45°. Thus

```
DEG 3 FIX [1 1 1] [3 ∡30 ∡60] [0 0 1] 45

                        XFORM ☞ [ -1.095 0.672 -.500 ]
```

## 11.4 Lists

In this section, we will review the general ideas of list objects, and study their application by means of examples.

A list is a *composite* object (section 3.3) made up of a series of other objects. It is similar to a program in this respect, but whereas a program is primarily intended for execution, a list is usually used as data. This difference is reflected in the execution actions of the two types of objects: executing a program automatically executes the objects that make up the program, but executing a list merely returns the list to the stack.

*Evaluating* a list by means of EVAL treats a list as a program, and successively executes the objects in the list.

There are other distinctions between lists and programs:

- You can take a list apart into its component objects, or combine objects into a list.
- You can extract and replace objects within a list, but not within a program (other than by editing the program).
- The local variable command → can not be used within a list.
- Names and programs in a list can not be "quoted" (section 3.8). For example,

  ```
  { << 1 >> } EVAL ☞ 1,
  ```

  compared with

  ```
  << << 1 >> >> EVAL ☞ << 1 >>.
  ```

  Similarly, names entered in lists are not quoted--if you enter a name with ' ' quotes in a list, the quotes are not retained. Therefore, to prevent the execution of a name or a program in a list, you must embed it within another set of program delimiters, e.g. { << << 1 >> >> } or { << 'ABC' >> }.
- You can not single-step through a list. If you evaluate a list containing a HALT or PROMPT, execution will suspend at the appropriate place, but SST in this case is equivalent to CONT.

Lists also resemble vectors, since they are both one-dimensional arrays of objects. You can create either a list or a vector out of a series of numbers (using →LIST or →ARRY). The difference is that in a list, the numbers do not necessarily have any particular association, whereas in a vector, they may be considered as the coordinates of a geometrical point, and hence are subject to various arithmetic operations and transformation rules.

### 11.4.1 List Operations

The HP 48 provides several commands that enable you to manipulate lists and their elements. The commands are quite similar to those used for array operations.

- To assemble objects into a list, use →LIST.

  ```
  1 (1,2) 'A+B' 3 →LIST ☞ { 1 (1,2) 'A+B' }.
  ```

  Note that the level 1 argument of →LIST (the 3 in this example) determines how many objects are taken from the stack to be combined into the list.

- To take a list apart, use OBJ→, or the equivalent for lists, LIST→.

  { 1 (1,2) 'A+B' } OBJ→ ☞ 1 (1,2) 'A+B' 3

  OBJ→ returns the elements of the list as separate stack objects, and leaves the number of elements in level 1.

- To determine the number of elements in a list, use SIZE.

  { 1 (1,2) 'A+B' } SIZE ☞ 3.

- To substitute objects into a list, use PUT or PUTI.

  { 1 (1,2) 'A+B' } 2 "ABC" PUT ☞ { 1 "ABC" 'A+B' },

  where the second element (1,2) in the initial list is replaced with the string "ABC". PUTI makes a substitution like PUT, but also leaves the index of the next element in level 1.

- To pull individual objects out of a list, use GET or GETI.

  { 1 (1,2) 'A+B' } 2 GET ☞ (1,2).

- To combine (concatenate) lists, use +.

  {1 2 3 4} {5 6 7 8} + ☞ {1 2 3 4 5 6 7 8}.

- + also add a stack object to a list, automatically applying 1 →LIST to the object:

  {2 3 4} 1 + ☞ {2 3 4 1}

  This is similar to the concatenation of objects to a string (section 3.4.3.1), where a non-string object is automatically converted to a string. If + is applied to a string and a list together, precedence is given to the list operation:

  "123" { 456 } + ☞ { "123" 456 }

  There is also an ambiguity when both objects are lists, which is resolved by giving precedence to concatenation. Thus if you want to add a list itself as an object to another list, you need an extra set of list delimiters:

```
{1  2  3  4}  {{5  6  7  8}}  +  ☞  {1  2  3  4  { 5  6  7  8}}.
```

- Since a list is an object, you can include lists within other lists. Notice the distinction between

```
{1 2 3 4}  {5 6 7 8}  +  ☞ {1 2 3 4 5 6 7 8}
```

  and

```
{1  2  3  4}  {5  6  7  8}  1  →LIST  +  ☞ {1  2  3  4 {5  6  7  8}}.
```

- To extract sublists from a list, use SUB. For example, the sequence

```
2  OVER  SIZE  SUB
```

  takes a list from level 1 and returns a shorter list consisting of the original list minus its first element (like the LISP function CDR). Thus,

```
{A  B  C}  2  OVER  SIZE  SUB  ☞  {B  C}.
```

- To replace several consecutive objects in a list, use REPL. REPL takes three arguments: the target list in level 3, the first substitution position in level 2, and the replacement list in level 1. The rules for use of REPL with lists are similar to those for use with strings (section 3.4.3.3). Assume that the target list contains $l_1$ elements, the replacement list has $l_2$ elements, and the substitution position is $n$. Then for

  $n>l_1$, the two lists are concatenated:

```
{A  B  C  D  E}  10  {F  G}  REPL
                  ☞  {A  B  C  D  E  F  G}
```

  $n+l_2-1>l_1$, elements $n$ through $l_1$ are replaced, and the leftover $l_2-(l_1-n)$ objects from the end of the replacement list are concatenated, so that the result list has $n+l_2-1$ elements:

```
{A  B  C  D}  4  {E  F}  REPL  ☞  {A  B  C  E  F}
```

  $n+l_2-1 \le l_1$, elements $n$ through $n+l_2-1$ are replaced in the target list; the remaining $l_1-l_2$ objects are unchanged:

```
{A  B  C  D}  2  {E  F}  REPL  ☞  {A  E  F  D}
```

$n=0$, the Bad Argument Value error is reported.

- You can find an object in a list by using POS:

  { A B C } 'B' POS ☞ 2.

  The number returned is the element number in the list of the search object, or 0 if the object is not contained in the list.

## 11.5 List Applications

The basic ideas of the use of the HP 48 object stack carry over into the principles and applications of list objects. A list is like an auxiliary stack, in which you can store and retrieve an indefinite number of objects, with no restrictions on the order or type of objects in the list. To illustrate this point, try the following:

1. Enter several objects of any types onto the stack.
2. Now use the interactive stack to combine all of the stack objects into a list:

   [△] [↱][△] ≡→LIST≡

   (In a program, you can obtain the same result with DEPTH →LIST.) Note that the objects are present in the list in the same order in which they were originally entered into the stack. The object that was in the highest stack level is the first element in the list; the object that was in level 1 is the last element. The list thus preserves an image of the original stack.
3. Save the list: 'OLD' [STO]. The stack is now empty.
4. Carry out any number of new calculations, leaving various objects on the stack. Discard these objects with [↱][CLR] (CLEAR), then enter

   OLD LIST→ DROP.

   This restores the stack as it was after step 1.

The ability to "freeze" a copy of the stack, store it away, then retrieve it later, is a useful list application in itself. But the main point of the example is to bring out the similarities between the stack and a list object, which suggests how you might use lists. The stack provides a medium for the ordered presentation of objects as input arguments for procedures (built-in or user-created), and for receiving the result objects. Lists can be used for the same purposes, especially for cases where juggling mixtures of input, intermediate, and output objects during the course of a calculation can become complicated.

To summarize, lists are a valuable programming tool for any situation in which the number of objects with which a program has to deal is not specified at the time the program is written. When a program works with a definite number of objects, it is appropriate to store those objects in variables, or to manipulate them on the stack as individual objects. But when you don't know in advance how many objects are to be handled, the best approach by far is to manage the objects together in a list. We will give some examples of this concept in the next sections.

### 11.5.1 Input Lists

Certain HP48 commands provide examples of the use of lists to combine several input objects into a single argument. There are two basic reasons for this approach:

1. *To provide flexibility along with uniformity.* For example, consider the command CON, which creates an array in which all elements have the same value. CON requires two pieces of information: 1) the common value for the elements, and 2) the dimensions of the array. The first is easy; the value is specified by a real or complex number in level 1. The second is a little more difficult, since an array can either be a one-dimensional vector, or a two-dimensional matrix. The use of a list as the level 2 argument for CON allows CON to handle both matrices and vectors. If the level 2 list contains one number, CON creates a vector; if the list contains two numbers, CON creates a matrix. If the dimensions were not combined into a list, there would have to be two versions of CON: one that takes two real numbers as arguments--the value and the vector dimension; and one that takes three numbers--the value and two matrix dimensions.

2. *To reduce the number of separate arguments.* Many graphics commands such as GXOR (section 10.3.1), use either complex numbers or binary integers to specify pixel coordinates. If the binary integers were entered as separate arguments, then these commands would violate the usual HP48 convention that any particular command uses the same number of arguments for each of its allowed argument type combinations. Instead, each pair of binary integers is combined as a list, to match one-for-one the uses of complex numbers.

Of these two reasons, the first is the only one of significance as a model for the use of lists as input arguments for user programs. That is, lists are ideal for situations where you have an indefinite number of inputs. An example of this is provided by the program MINL (section 12.3), which finds the minimum among a series of numbers in a list. The program is written for series of any length--it has only to execute SIZE on the input list to determine how many numbers it needs to compare. Furthermore, during its execution, the numbers remain in the list, except for when they are extracted one-by-one from the list for the comparisons. Keeping track of that single list, which could be stored in a global or local variable if necessary, is much simpler than trying to maintain the series of numbers as separate stack objects. If you are not yet convinced of the utility of lists, try writing a version of MINL that uses no lists (or arrays). See also the recursive program RMINL, in section 12.10.

#### 11.5.1.1 Index List Arguments

Commands such as PUT and GET that use argument lists containing one or more real numbers also allow you to substitute other types of objects for the numbers. The substitute objects must evaluate (by means of →NUM) to real number values. In particular, this means you can use symbolic values (names or expressions), or even programs, rather than specific numerical values. For example, the sequence

```
→ m  « 1  m  SIZE  2  GET  FOR  n  m  { 3  n }  GET  NEXT »
```

returns in order all of the numbers from the third row of a matrix. This capability can lead to some convoluted executions when argument lists contain (directly or indirectly) programs that manipulate the stack. You can predict the execution in such cases as follows:

1. Empty lists cause the Bad Argument Value error.
2. Lists containing only real numbers go directly on to the computation part of the command.
3. When a list contains elements other than real numbers:
    a. The stack depth (less the list) is recorded.
    b. Each non-real number list element is evaluated numerically (→NUM). After each evaluation, if the resulting stack is empty, the error Too Few Arguments is reported. If the resulting level 1 object is not a real number, the Bad Argument Type error is reported.
    c. If the stack depth has decreased, the Too Few Arguments is returned. Otherwise, the new objects, plus any excess, are combined back into a list.
    d. The command execution is started over again with the new list.

Command errors that occur during evaluation of procedures within the argument list identify the guilty command and return its arguments as usual. However, other errors that occur in step 2 do not identify any command.

If a non-numeric list is used as the index argument for GETI or PUTI, the incremented index list is returned with real number indices.

## 11.5.2 Output Lists

Just as you can use a list to combine an indefinite number of *input* objects into a single argument, you can use a list to receive the multiple-object *output* of a program. This approach makes it easy to manipulate a program's output--either to save it in a variable, or to use it as the input for another program.

■ *Example*. For any integer $n$, compute the first $n+1$ terms $F_n$ of the Fibonacci series. This series is defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

| FIB | *Fibonacci Series Generator* | | ED29 |
|---|---|---|---|
| | *level 1* | \| | *level 1* |
| | $n$ | ☞ | $\{0\ 1\ \cdots\ f_n\}$ |

```
<< {0 1}                          Start the list with F0 and F1.
  SWAP DUP 1
  IF >                            If n is ≤ 2, quit.
  THEN 0 1                        Initial values Fn-2 and Fn-1.
    3 4 ROLL 1 +                  From 3 to n...
    START DUP ROT +               Fn-2 + Fn-1.
      ROT OVER +                  Add Fn to the output list.
      3 ROLLD                     | { ... Fn } Fn-2 Fn-1 |
    NEXT DROP2
  ELSE DROP
  END
>>
```

## 11.5.3 Lists of Intermediate Results

When a program contains loop structures, or is written recursively, it is usually necessary to ensure that the stack has the same configuration at each iteration. A particularly convenient means of achieving this is to use a list as an auxiliary data stack, to hold an indefinite number of intermediate results in a constant position on the stack.

The program SORT illustrates the use of lists of intermediate results. SORT orders a list of numbers (or strings), so that the smallest (most negative, or alphabetically first) object is moved to the start of the list, and so on to the largest (alphabetically last) object as the last element. SORT uses a recursive algorithm that can be summarized as:

1. Remove an object from the middle of the list and separate the remaining objects into two lists, one containing objects that are smaller than the middle object, and the other containing larger objects.
2. Sort the two lists using the same algorithm.
3. Combine the results back into a single list, with the sorted "smaller" objects first, followed by the original middle object, then the sorted "larger" objects.

| SORT | *Sort a List in Increasing Order* | | | A1BE |
|---|---|---|---|---|
| | *level 1* | \| | *level 1* | |
| | { *list* } | ☞ | { *ordered list* } | |

```
« IF  DUP  SIZE  1  >              If the list has fewer than 2 elements, just
                                   return.
  THEN  OBJ→                       Put the objects on the stack.
  DUP  2  /  1  +  ROLL            Get the middle object.
  NEWOB  →  x                      Save the object as x.†
    «  { }  { }                    Initialize "less" and "greater" lists.
      2  4  ROLL                   Iterate for n-1 elements:
      START  ROT                   Get the next element.
        DUP  x
        IF  <                      If the element is < x,
        THEN  ROT  +  SWAP         add to first list.
        ELSE  +                    Element ≥ x, so add to second list.
        END
      NEXT
      SORT                         Sort the first list.
      SWAP  SORT                   Sort the second list.
      x  +  SWAP  +                Combine the lists.
    »
  END
»
```

† NEWOB saves memory by separating *x* from the original list. See section 11.6.

▪ *Example.*

{ 5 1 2 4 3 } SORT ☞ { 1 2 3 4 5 }.

The algorithm used by SORT is not specific to numerical ordering; you can rewrite SORT for other types of sorting by replacing the < comparison with any other test or sequence of tests. A more general approach is taken by the program GSORT, which sorts a list of objects according to a test defined by another program that is supplied as a second argument to GSORT.

| GSORT | *General-purpose Sort* | | | EFFC |
|---|---|---|---|---|
| | *level 2* | *level 1* | | *level 1* |
| | { *list* } | << *test* >> | ☞ | { *list* } |

```
<< → test                          Save test program as test.
  << IF DUP SIZE 1 >               If the list has fewer than 2 elements, just
                                   return.
    THEN OBJ→                      Put the objects on the stack.
    DUP 2 / 1 + ROLL               Get the middle object.
    NEWOB → x                      Save the object as x.
      << {} {}                     Initialize "true" and "false" lists.
        2 4 ROLL                   Iterate for n-1 elements:
        START ROT                  Get the next element.
          IF DUP x test EVAL       If test is true,
          THEN ROT + SWAP          add element to first list.
          ELSE +                   Otherwise, add element to second list.
          END
        NEXT
        test GSORT                 Sort the first list.
        SWAP test GSORT            Sort the second list.
        x + SWAP +                 Combine the lists.
      >>
    END
  >>
>>
```

To use GSORT, enter an unsorted list of objects, followed by a program *test-program* that represents a logical test. *Test-program* should work like this:

$object_1$ $object_2$ *test-program* ☞ *flag.*

*Flag* should be *true* if $object_1$ is to precede $object_2$, or *false* otherwise. GSORT sorts the list so that the sequence

*object*$_n$ *object*$_{n+1}$ *test-program*

will return a *true* flag for any two consecutive objects *object*$_n$ and *object*$_{n+1}$ in the list (unless the order is ambiguous). For example, the numerical ordering performed by SORT is represented by the program « < »; therefore « < » GSORT is equivalent to SORT. Other examples:

- « > » GSORT sorts numbers or strings in decreasing numerical or alphabetical order.
- « ABS SWAP ABS > » GSORT sorts in order of increasing absolute value.
- « SIZE SWAP SIZE > » GSORT sorts strings or lists in order of increasing length.
- To sort complex numbers in order of increasing polar angle from 0° to 360°:

```
« « ARG DUP 0 IF < THEN -1 ACOS 2 * + END »
ROT OVER EVAL ROT ROT EVAL < » GSORT
```

## 11.6 Composite Objects and Memory

There is a subtlety in the management of composite objects--lists, algebraics, and programs--that you should keep in mind when programming with these objects. When an object originates in a composite object, such as when GET extracts an object from a list, or when executing a program leaves an object from the program on the stack, the composite object remains in memory as long as any of its component objects remains on the stack or is otherwise in use. If the composite object itself is stored in a global or port variable (or is part of a program or another list in a variable), this point is unimportant, since the memory used by the object is accounted for in the variable. However, if the composite object has not been stored, the memory it uses will not be recovered until it *and* any objects that have been extracted from it are removed from the stack. For the individual objects, "removed" means dropped, stored in a global or port variable (not a local variable), or combined into a vector or another list.

To see this effect, disable the argument, stack, and command recovery systems so that they will use no memory, and execute

```
1 50 FOR n n NEXT 50 →LIST
```

to create a list of 50 numbers. Now execute 50 GET, so that the number 50 (from the list) is left on the stack. Next, execute MEM to determine how much memory is available. Use SWAP DROP to drop the 50, then execute MEM again. Notice that the difference is 447.5 bytes--far more memory than you would expect to be recovered by dropping the single real number 50. The large difference between the successive MEM's actually arises because the removal of the 50 allowed the HP48 to delete the copy of the list that it had been preserving.

As mentioned above, you can "uncouple" an object from the list from which it came by either storing the object in a global variable, or by including it in another list (or an array, if the object is a number). An even simpler method is to execute NEWOB (*NEW OBject*). NEWOB may not appear to do anything, since the object it returns matches the original, but in fact NEWOB creates a new independent copy of an object that is disassociated from any other object. Using NEWOB in the SORT and GSORT programs listed in the preceding section enables those programs to sort lists substantially larger than they could if NEWOB were omitted.

One additional note: if you are dealing only with a collection of numbers (all real or all complex), you can often use a vector (or a matrix, if you want a rows-and-columns type of organization) to store the numbers, instead of a list. For storing more than a few numbers, a vector is more memory-efficient than a list, and you can perform many of the same operations to assemble and disassemble vectors as you can with lists. The main disadvantage of using a vector in place of a list is that there is no built-in command for adding (concatenating) numbers to vectors, or combining two vectors into a longer one. The following program provides list-like concatenation for vectors:.

| ADDV | *Concatenate Vectors* | 9645 |
|---|---|---|
| *level 2* *level 1* | | *level 1* |
| [ *vector*$_1$ ] [ *vector*$_2$ ] | ☞ | [ *vector*$_3$ ] |

```
«
  « DUP TYPE                     Program to apply to both vectors.
    IF 1 ≤                       Is the object a number?
    THEN 1                       Then treat as a one-element vector.
    ELSE OBJ→ OBJ→ DROP          For a vector, put its elements on the stack.
    END
    DUP 2 + ROLL                 Get the object above the vector.
  » → s                          Store the program as a subroutine s.
  « SWAP s EVAL s EVAL           Apply s to both vectors.
    +                            Total number of elements.
    1 →LIST →ARRY                Combine the numbers into the result vector.
  »
»
```

## 11.7 Symbolic Arrays

HP 48 array objects are designed for the efficient storage of real and complex numbers, and can not contain symbolic elements. Nevertheless, it is possible to deal with symbolic arrays on the HP 48 by using the more flexible list objects to represent the arrays. In this section, we will present several programs for symbolic array calculations, which also serve as examples of the use of lists and arrays, and other programming techniques. These programs obviously do not exhaust the subject of symbolic array manipulations, but you can use them as a basis for developing additional programs.

All of the programs follow the convention that a symbolic array is represented by a list of lists. An $n \times m$ array is represented as a list containing $n$ $m$-element lists. For example, the list {{ $a$ $b$ } { $c$ $d$ } { $e$ $f$ }} stands for the matrix

$$\begin{vmatrix} a & b \\ c & d \\ e & f \end{vmatrix}.$$

There is no special provision for vectors, which may be represented as $1\times n$ or $n\times 1$ arrays in this system. Since all of the arrays are two-dimensional, we will always use two separate (i.e. not in a list) real numbers to specify elements or dimensions.

The programs do not check for the integrity of the lists you may enter--they presume that all of the inner lists in a particular symbolic array list have the same number of elements, that all of the elements are either names, numbers, or algebraic expressions, and that there are no extraneous elements in any of the lists. If the programs are applied to lists that violate any of these assumptions, they may error or return nonsensical results. If this is not satisfactory, you can easily revise the programs to include more argument testing.

### 11.7.1 Utilities

To start with, here are several utility programs for symbolic arrays that are analogous to various HP 48 array commands:

| | |
|---|---|
| DIM | returns the dimensions $n$ (rows) and $m$ (columns) of a symbolic array. |
| SA→ | unpacks a symbolic array into separate stack objects. |
| →SA | combines stack objects into a symbolic array. |
| N→S | converts an ordinary numerical array into a symbolic array. Vectors are converted into $n\times 1$ symbolic arrays. |
| S→N | attempts to evaluate all elements in a symbolic array into numbers. If successful, it then converts the symbolic array into a numeric array. |

| | |
|---|---|
| APLY1 | applies a program to each element of a symbolic array. |
| APLY2 | combines two symbolic arrays by applying a program to pairs of elements. |
| STRN | transposes a symbolic array. |

| DIM | *Symbolic Array Dimensions* | | | 2BA8 |
|---|---|---|---|---|
| *level 1* | | *level 2* | *level 1* | |
| {{ *array* }} | ☞ | *n* | *m* | |

```
« DUP SIZE SWAP 1 GET SIZE
»
```

| SA→ | *Symbolic Array to Stack* | | | | 28D2 |
|---|---|---|---|---|---|
| *level 1* | | ... | *level 2* | *level 1* | |
| {{ *array* }} | ☞ | ...*elements*... | *n* | *m* | |

```
« OBJ→ OVER SIZE → n m          Store dimensions.
  « 1 n
    FOR i
      '(i-1)*m+n-i+1' EVAL ROLL  Get the ith row.
      OBJ→ DROP                  Put its elements on the stack.
    NEXT
    n m                          Return the dimensions.
  »
»
```

| →SA | *Stack to Symbolic Array* | | | | 98FD |
|---|---|---|---|---|---|
| ... | *level 2* | *level 1* | | *level 1* | |
| ...*elements*... | *n* | *m* | ☞ | {{ *array* }} | |

```
« → n m                          Save the dimensions.
  « 1 n
    FOR i
      m →LIST                    Make the ith row.
      'm*(n-i)+i' EVAL ROLLD     Put it at the end.
    NEXT
    n →LIST                      Combine the rows.
  »
»
```

| N→S | *Numeric to Symbolic* | B665 |
|---|---|---|
| | *level 1* \| *level 1* | |
| | [[ *array* ]] ☞ {{ *array* }} | |

| | |
|---|---|
| `<< OBJ→ OBJ→` | Put elements on the stack. |
| `IF 1 ==` | Is this a vector? |
| `THEN 1` | Then add the other dimension. |
| `END` | |
| `→SA` | Combine into a symbolic array. |
| `>>` | |

```
<<  OBJ→  OBJ→
  IF  1  ==
  THEN  1
  END
  →SA
>>
```

| S→N | *Symbolic to Numeric* | 0AC1 |
|---|---|---|
| | *level 1* \| *level 1* | |
| | {{ *array* }} ☞ [[ *array* ]] | |

```
<<  SA→                                   Put elements on the stack.
  DUP2  *  →  n  m  p                     Save dimensions and number of elements.
  <<  1  SF                               Flag 1 clear will indicate a non-number.
    1  p
    START  p  ROLL                        Get the next element.
      IFERR  DUP  →NUM                    Convert it to a number.
      THEN  DEPTH  p  -  DROPN            If →NUM fails, discard any partial results.
        1  CF                             Remember the failure.
      ELSE  SWAP  DROP
        IF  DUP  TYPE                     If the result is not a number...
        THEN  1  CF                       ...clear flag 1.
        END
      END
    NEXT
    n  m                                  Dimensions for result array.
    IF  1  FC?
    THEN  →SA                             If there are non-numbers, return a symbolic
                                          array.
    ELSE  2  →LIST  →ARRY                 Otherwise, return a numeric array.
    END
  >>
>>
```

S→N sets flag 1 to indicate a successful conversion, and clears it otherwise.

| APLY1 | *Apply Program to 1 Symbolic Array* | 5D68 |
|---|---|---|

| *level 2* | *level 1* | | *level 1* |
|---|---|---|---|
| {{ *array* }} | << *program* >> | ☞ | {{ *array'* }} |

| | |
|---|---|
| `<< OVER DIM → a f n m` | Store the array, program and dimensions. |
| `<< 1 n` | |
| `FOR i` | |
| `1 m` | |
| `FOR j` | |
| `a i GET j GET` | Get the *ij* element. |
| `f EVAL` | Apply the program. |
| `NEXT` | |
| `m →LIST` | Pack up the *i*th row. |
| `NEXT` | |
| `n →LIST` | Pack up the array. |
| `>>` | |
| `>>` | |

```
<<  OVER  DIM  →  a  f  n  m

  <<  1  n
    FOR  i
      1  m
      FOR  j
        a  i  GET  j  GET
        f  EVAL
      NEXT
      m  →LIST
    NEXT
    n  →LIST
  >>
>>
```

| APLY2 | *Apply Program to 2 Symbolic Arrays* | 9815 |
|---|---|---|

| *level 3* | *level 2* | *level 1* | | *level 1* |
|---|---|---|---|---|
| {{*array*$_1$}} | {{*array*$_2$}} | << *program* >> | ☞ | {{*array*$_2$}} |

| | |
|---|---|
| `<< ROT DUP DIM → a2 f a1 n m` | Save the arrays, the program, and the dimensions |
| `<< 1 n` | |
| `FOR i` | |
| `1 m` | |
| `FOR j` | |
| `a1 i GET j GET` | Get a1$_{ij}$. |
| `a2 i GET j GET` | Get a2$_{ij}$. |
| `f EVAL` | Execute the program. |
| `NEXT` | |
| `m →LIST` | Pack up the *i*th row. |
| `NEXT` | |
| `n →LIST` | Pack up the result array. |
| `>>` | |
| `>>` | |

```
<<  ROT  DUP  DIM  →  a2  f  a1  n  m

  <<  1  n
    FOR  i
      1  m
      FOR  j
        a1  i  GET  j  GET
        a2  i  GET  j  GET
        f  EVAL
      NEXT
      m  →LIST
    NEXT
    n  →LIST
  >>
>>
```

| STRN | *Transpose Symbolic Array* | A128 |
|---|---|---|
| | *level 1* \| *level 1* | |
| | {{ $A_{ij}$ }} ☞ {{ $A_{ji}$ }} | |

| | |
|---|---|
| « DUP DIM → a n m | Save array and dimensions. |
| « 1 m | |
| FOR j 1 n | |
| FOR i a i GET j GET | $A_{ij}$ |
| NEXT | |
| NEXT | Elements are now in transposed order. |
| m n | |
| » | Discard the original array. |
| →SA | Pack up the new array. |
| » | |

## 11.7.2 Symbolic Array Arithmetic

Using the APLY1 and APLY2 utilities listed in the preceding section, it is straightforward to create programs for simple symbolic array arithmetic.

| | |
|---|---|
| SADD | adds two symbolic arrays. |
| SSUB | subtracts two symbolic arrays. |
| SMS | multiplies a symbolic array by a scalar (number, name, or algebraic). |
| SMUL | multiplies two symbolic arrays. |

| SADD | *Add Symbolic Arrays* | E3E4 |
|---|---|---|
| | *level 2* *level 1* \| *level 1* | |
| | {{ $A_{ij}$ }} {{ $B_{ij}$ }} ☞ {{ $A_{ij}+B_{ij}$ }} | |

```
« « + COLCT » APLY2
»
```

| SSUB | *Subtract Symbolic Arrays* | 87B2 |
|---|---|---|
| | *level 2* *level 1* \| *level 1* | |
| | {{ $A_{ij}$ }} {{ $B_{ij}$ }} ☞ {{ $A_{ij}-B_{ij}$ }} | |

```
« « - COLCT » APLY2
»
```

You may wish to omit COLCT from SADD or SSUB, to speed up execution or to prevent an unwanted rearrangement. You can execute << COLCT >> APLY1 on an array to collect terms once after a series of calculations.

| SMS | *Scalar Multiply Symbolic Arrays* | | | C58A |
|---|---|---|---|---|
| | *level 2* | *level 1* | | *level 1* |
| | $\{\{ A_{ij} \}\}$ | $z$† | ☞ | $\{\{ z{\cdot}A_{ij} \}\}$ |
| | $z$† | $\{\{ A_{ij} \}\}$ | ☞ | $\{\{ z{\cdot}A_{ij} \}\}$ |

```
<<  IF  DUP  TYPE  5 ==                  Put the array in level 2.
  THEN  SWAP
  END
  →  z                                   Save the scalar.
  <<  <<  z  *  >>                       Program for APLY1.
    APLY1
  >>
>>
```

†*z* can be a number, a name, or an algebraic expression.

| SMUL | *Multiply Symbolic Arrays* | | | 12A9 |
|---|---|---|---|---|
| | *level 2* | *level 1* | | *level 1* |
| | $\{\{ A_{ij} \}\}$ | $\{\{ B_{ij} \}\}$ | ☞ | $\{\{ (AB)_{ij} \}\}$ |

```
<<  DUP2  DIM  ROT  DIM
  →  a1  a2  n2  m2  n1  m1             Save the arrays and dimensions.
  <<  1  n1
    FOR  i  1  m2
      FOR  j  0  1  m1
        FOR  k                           Compute ∑_k A_ik B_kj:

          a1  i  GET  k  GET             A_ik
          a2  k  GET  j  GET             A_kj
          *  +
        NEXT
      NEXT
      m2  →LIST                          Pack up the ith row.
    NEXT
    n1  →LIST                            Pack up the result array.
  >>
>>
```

## 11.7.3 Determinants and Characteristic Equations

In this section, we develop a program DETM that computes the determinant of a symbolic matrix from the formula

$$\mathrm{DET}\,\mathbf{A} = \sum_{i=1}^{n}(-1)^{i+1}\mathbf{A}_{i1}\,\mathbf{A}^{C}_{i1},$$

where $\mathbf{A}^{C}_{ij}$ is the *ij cofactor* (unsigned) of element $\mathbf{A}_{ij}$, and $n$ is the number of rows or columns in the (square) matrix. This is a recursive form of the definition of DET, since the cofactor of an element is the determinant of its minor:

$$\mathbf{A}^{C}_{ij} = \mathrm{DET}\,\mathbf{A}^{M}_{ij}.$$

(The minor $\mathbf{A}^{M}_{ij}$ is defined in section 11.1. Note that some textbooks may give different definitions for the terms *minor* and *cofactor.* )

The programs to compute determinants of symbolic matrices, SDET (*symbolic determinant*), SCOF (*symbolic cofactor*), and SMINOR (*symbolic minor*), are straightforward realizations of the above definitions, including the recursion. They are presented in an order (SDET first, SMINOR last) that demonstrates a "top-down" programming approach, where you write a program before writing the subroutines that it calls. This kind of approach lets you concentrate on the essential main logic flow of a program, before worrying about the details. Also, when you come to write the subroutines (the "details"), you know exactly what the stack use of the subroutines should be. Note, however, that the opposite, "bottom-up" order is usually more convenient for actually entering the programs into the HP48. By entering the subroutines first, you can then enter their names into other programs by pressing the appropriate VAR menu keys.

SDET computes the determinant of a matrix as a sum along the first column, of elements times their respective signed cofactors. (The sign $-1^{i+1}$ is computed explicitly in this program, rather than as part of the cofactor program, so that the row and column numbers that determine the sign don't have to be passed along down through all of the levels of recursion.) The unsigned cofactor of a matrix element is the determinant of the corresponding minor; for a $1\times1$ matrix, the cofactor is 1. The program SCOF called by SDET embodies these points. At the point in SDET where SCOF is executed, the stack contains a matrix and the row and column number of the desired cofactor.

The two programs SDET and SCOF call each other back and forth--each is a subroutine of the other. The calculation proceeds the same way it would if you were computing the determinant by hand, where you use cofactors to compute the determinants and determinants to compute cofactors.

| SDET | *Symbolic Determinant of a Matrix* | D39C |
|---|---|---|
| | *level 1* \| *level 1* | |
| | {{ *matrix* }} ☞ *determinant* | |

| | |
|---|---|
| `<< DUP DIM DROP → a n` | Save the matrix (a) and its dimension. |
| `<< 0` | Initialize the sum. |
| `1 n` | |
| `FOR i` | For each element in column 1... |
| `a i GET 1 GET` | Get the element. |
| `a i 1 SCOF *` | Multiply by the (unsigned) cofactor. |
| `-1 i 1 + ^ *` | Multiply by $(-1)^{i+1}$ |
| `+` | Add to the current sum. |
| `NEXT` | |
| `>>` | |
| `>>` | |

| SCOF | *(Unsigned) Symbolic Cofactor* | 5785 |
|---|---|---|
| | *level 3* *level 2* *level 1* \| *level 1* | |
| | {{ *matrix* }} *r* *c* ☞ *cofactor* | |

| | |
|---|---|
| `<< 3 PICK DIM DROP` | Get the dimension of the matrix. |
| `IF 1 ==` | If it's a 1×1 matrix... |
| `THEN 3 DROPN 1` | ...then just return 1. |
| `ELSE SMINOR SDET` | ...else, return the determinant of the cofactor. |
| `END` | |
| `>>` | |

SCOF uses a subprogram SMINOR to compute the *nm* minor of a symbolic matrix. It would be straightforward to modify the programs MINOR and DELROW from section 12.1 to work with symbolic matrices; however, because the structure we are using for symbolic arrays makes it easy to break an array into rows, we use a different approach and write SMINOR as a single program.

| SMINOR | *Minor of a Symbolic Matrix* | | | | D352 |
|---|---|---|---|---|---|
| | *level 3* | *level 2* | *level 1* | | *level 1* |
| | {{ *matrix* }} | *r* | *c* | ☞ | {{ *minor* }} |

| | |
|---|---|
| `<< → r c` | Save the row and column number. |
| `  << OBJ→` | Put the rows on the stack. |
| `    OVER SIZE OVER 1 - → m n` | Save the (final) dimensions. |
| `    << r - 1 + ROLL DROP` | Discard the *r*th row. |
| `      1 n` | For each remaining row: |
| `      START n ROLL` | Get the next row. |
| `        IF c 1 -` | $r=1$ is a special case. |
| `        THEN DUP 1 c 1 - SUB` | Elements in columns $< r$. |
| `          SWAP c 1 + m SUB` | Columns $> r$. |
| `          +` | New row. |
| `        ELSE 2 m SUB` | $r=1$ case. |
| `        END` | |
| `      NEXT` | |
| `      n →LIST` | Pack up the result. |
| `    >>` | |
| `  >>` | |
| `>>` | |

■ *Example.* Compute the determinant of the matrix $\begin{vmatrix} A & B \\ C & D \end{vmatrix}$.

■ *Solution.*

```
{{ A  B }{ C  D }}  SDET  ☞  'A*D-C*B'
```

You might note that for purely numeric matrices, SDET can occasionally produce more accurate results than you obtain by applying the HP 48 command DET to the same matrix. For example, applying SDET to the matrix

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

returns 0, which is exactly correct, whereas using the command DET returns 2.14259999999E-10. This happens because SDET actually carries out all of the matrix element multiplications explicitly, whereas, except for 2 × 2 matrices, DET does not. DET uses more advanced numerical methods to speed up calculation and minimize memory use for large matrices, and to insure a reliable answer even for matrices with elements of widely varying values.

An excellent application of the symbolic array capabilities presented here is the computation of the characteristic equation of a matrix, which is used in the determination of eigenvalues. The characteristic equation of a matrix **A** is defined as

$$\mathrm{DET}(\mathbf{A} - x\mathbf{I}) = 0,$$

where $x$ is an eigenvalue, and **I** is the identity matrix. The program CEQN returns the characteristic equation of a symbolic or numeric matrix, where you specify the matrix in level 2, and the name to be used for the eigenvalue variable in level 1. [Note: the sequence x n TAYLR is used in CEQN to simplify the result (see section 9.8.3). You can omit this sequence for faster execution of CEQN, which will then return an equivalent but longer form of the equation.]

| CEQN | *Characteristic Equation* | | | 1831 |
|---|---|---|---|---|
| | *level 1* | *level 2* | | *level 1* |
| | {{*matrix*}} | *'name'* | ☞ | *'equation'* |
| | [[*matrix*]] | *'name'* | ☞ | *'equation'* |

| | |
|---|---|
| « IF OVER TYPE 5 ≠ | If it's a numeric matrix... |
| THEN SWAP N→S SWAP | ...make it symbolic. |
| END | |
| OVER SIZE → x n | Save the name and dimension. |
| « n IDN | Make an identity matrix. |
| N→S | Make it symbolic. |
| x SMS | Multiply by X |
| SSUB | Subtract from the original matrix. |
| SDET | Determinant |
| x n TAYLR | Simplify the expression. |
| 0 = | Make into an equation. |
| » | |
| » | |

▪ *Example*. Find the characteristic equation in X of $\begin{vmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 1 & 2 \end{vmatrix}$.

▪ *Solution*:

```
[[ 1 0 2 ] [ 0 1 4 ] [ 0 1 2 ]]  'X'  CEQN  ☞  '-2-X+4*X^2-6/3!*X^3=0'.
```

# 12. Program Development

Program development is the process of transforming a computation problem into a calculator program. No two problems are identical, of course, but in this chapter we will consider certain elements that involve a common approach from program to program. Such elements include program techniques for obtaining input from a program's user and presenting the program's results in a manner that the user can interpret. There are also mechanical aspects such as editing, debugging and program optimization--altering a program to improve its speed or to minimize memory use. In all of these matters there are elements of art, and of personal preferences and style, that preclude a authoritative prescriptions. It is not even easy to define what distinguishes a good program from a bad one. For example, one program might require less memory, or run faster, or have fewer steps than another. But perhaps you can develop the less efficient program and use it to obtain results in less time than it takes just to design the other; which, then, is the "better" program?

In this chapter, we will study some general-purpose topics in HP48 program development, with examples to illustrate each topic. From these and other examples throughout this book, you will see how various HP48 programming tools and techniques can be combined. You can remember those methods that appeal to you, and through practice, develop your own methodology.

## 12.1 Program Editing

To make any alteration to an existing program in order to correct an error, optimize execution, or add features, you must *edit* the program. Because HP48 programs are objects, you edit a program the same way you edit any other object. That is, you use EDIT or VISIT to create a text version of the program in the command line, use the facilities of the command line to make the alterations you desire, then execute ENTER to replace the old copy of the program with the new one. Re-entering the entire program this way ensures that objects and program structures are entered correctly. Even if you develop or edit a program as text on a computer, as you transfer it to the HP48 it is subjected to the same syntax checking as it would had it been entered into the command line.

When an object is copied into the command line by EDIT or VISIT, any numbers in the object are shown to their full precision, regardless of the current number display mode. That is, floating-point numbers are shown in STD format, and binary integers with a wordsize of 64 bits. This prevents the accidental changing of numbers during editing. Also, binary integers are shown with an identifying character (b, d, h, or o), so that reentering a binary integer will not change its base regardless of the current mode.

The advantages of the HP 48 program editing approach are:

- The same editing methods apply to all HP 48 object types, so that you don't have to learn special techniques for each object type.
- No changes you make during an edit are "final" until you press ENTER . If you change your mind while you are editing a program, you can just press ON to cancel the edit and leave the program intact.

On the other hand, there are two important disadvantages:

- For a large program, it can take a substantial amount of time for the HP 48 to translate the entire program object into its text form, and, when you're done editing, to build the new program from the command line text.
- During the execution of ENTER, there must be memory available for as many as three versions of the program (the original, the command line text, and the new version) simultaneously. This restricts the size of the program that can be edited.

The latter disadvantage is the most serious, because it can happen that there isn't enough memory to permit any changes to an existing program, even if the changes don't increase the final size of the program. Both disadvantages dictate that you keep programs small, typically less than a few dozen objects. If a program starts to get too big as you develop it, break it up into smaller subprograms that are executed by a short main program. Even though this costs a little more memory for the subprogram names and variables, the smaller programs will be editable when a big single program is not.

### 12.1.1 Low Memory Editing Strategies

When the HP 48 runs out of memory as you try to enter an edited program (or any other object), you can use the following steps to increase the available memory:

1. Remove any unwanted objects--clear the stack, kill any suspended programs (section 12.2.1), and purge unneeded variables from user memory.
2. Disable last arguments and stack recovery: ◁ MODES NXT STK□ ARG□ .
3. Recall the object you want to edit to level 1. If the object is stored in a variable, purge the variable to save the memory used for the variable.
4. Press ◁ EDIT .
5. Press CMD□ CMD . This empties the command stack, but leaves command recovery enabled.
6. Make your changes, and press ENTER . If there still is insufficient memory, press ◁ LAST CMD to return the object to the command line, CMD□ to disable and

clear the command stack, then ENTER. This step is risky, because if there is still not enough room, you will have lost the edited version of the object.

If the preceding steps fail, you can take the more drastic step of purging the object you are trying to edit. That is,

1. With the object in level 1, press CMD to reactivate the command stack.
2. Press ◁ EDIT to copy the object to the command line; make your changes.
3. Press ENTER. This will presumably fail due to insufficient memory.
4. Press ⇐ to discard the object from level 1.
5. Press ◁ LAST CMD to recover the command line with the altered text version of the object.
6. Try ENTER again. If there is no error message, you're finished. But if ENTER fails again, then...
7. Press ◁ LAST CMD to retrieve the command line one more time. Now press CMD□ to disable the command stack. Press ENTER. If this fails, you're out of options, and out of luck--all copies of the object are gone. Generally, however, this process will succeed unless you are making major additions to the edited object.

## 12.2 Starting and Stopping

As we have discussed in previous sections, HP48 programs are highly structured, and each has only a single entrance and exit. This fact makes starting and stopping an HP48 program a different proposition from the simple run/stop capability of calculators (like the HP 41, for instance) that use a keystroke programming language.

In the HP48, a program that has stopped execution at some point but can be restarted from there is said to be *suspended*. This is different from a program that is terminated while running by ATTN, which abandons all pending execution in the currently executing program and cancels pending returns to any other programs that may have called that program. (In more precise terms, the return stack is cleared, and the normal stack display and keyboard are reactivated.) A program can suspend itself by including HALT or PROMPT in its definition, or you can suspend it manually by using the debug and single-step keys in the program control menu. For sake of illustration here we will concentrate on HALT, but the discussion generally applies to the other methods as well.

When one or more programs are suspended by any means, the HALT annunciator is displayed in the status area. The keyboard is activated, and all calculator operations work normally. The HP 48 can maintain this state indefinitely--it behaves as if you had started up another calculator "inside" the halted program. This suspended program environment has its own local memory containing a new recovery stack, independent of the usual saved stack that was present before the suspended program was started. The calculator operates in the suspended environment until you execute CONT, whereupon the suspended program resumes execution at the point at which it was stopped.

You can "nest" suspended program environments one within another without limit (other than available memory). While one program is halted, you can run another program that is suspended in turn, with another local memory for a recovery stack, and so on. Each time you execute CONT, the latest suspended environment is deleted, including its recovery stack. If you press [◁] [LAST STACK] immediately after a program completes execution, the stack that was saved by the ENTER that started the program is restored. To demonstrate this, enter the following program and name it A:

```
« CLEAR 1 2 HALT 3 4 » 'A' STO
```

Then:

| **Keystrokes:** | **Results:** | | |
|---|---|---|---|
| [▷] [CLR] 'X' 'Y' | | | |
| [ENTER] | 2: | 'X' | |
| | 1: | 'Y' | |
| A [ENTER] | 2: | 1 | |
| | 1: | 2 | HALT annunciator is on. The program has put 1 and 2 on the stack, and halted. |
| [▷] [CLR] | 2: | | |
| | 1: | | |
| [◁] [LAST STACK] | 2: | 1 | |
| | 1: | 2 | [◁] [LAST STACK] restores the stack from prior to the previous [▷] [CLR]. |
| [◁] [CONT] | 4: | 1 | |
| | 3: | 2 | |
| | 2: | 3 | |

| | | | |
|---|---|---|---|
| | 1: | 4 | The program A resumes, pushes 3 and 4 onto the stack, and is finished. |
| [⬑] [LAST STACK] | 2: | 'X' | |
| | 1: | 'Y' | Back to the original environment; the last ENTER in this environment was the one that started the program A. [⬑] [LAST STACK] restored the stack as it was before that ENTER. |

Since the command line itself is a program, you can include a HALT or a PROMPT in the command line even if it is not explicitly contained in a program object delimited by << >>. When you press [ENTER] , the command line is executed up to the HALT or PROMPT. Then you can perform any normal operations; when you finally press [⬑] [CONT] , the rest of the suspended command line is executed. Among other uses, this provides an easy way to save a copy of the current stack while you carry out some unrelated calculations. With an empty command line, execute HALT [ENTER] . You can now clear the stack and perform any other operations; afterwards you can restore the original stack by pressing [⬑] [CONT] [⬑] [LAST STACK] .

Keep in mind when you're working with a suspended program that local variables created by the program may be present. For example, if a program halts while a local variable A that it created still exists, then executing the name A from the command line returns the value of that local variable, not the value of a global variable A that might also exist. (Pressing the [A] key in the VAR menu always executes the global name A regardless of any local variables that might exist.)

### 12.2.1 ATTN, DOERR and KILL

The [ATTN] (*Attention!*) key is intended to let you get the "attention" of the calculator. Pressing it tells the calculator to stop what it is doing: stop all operations, procedures, etc., clear any special displays, reactivate the normal keyboard, and show the standard stack display. You also use the key to turn the calculator on, although that's almost a secondary role compared to the key's ATTN role (labeling the key face with ON rather than ATTN is primarily for the sake of people using the calculator for the first few times).

[ATTN] is a "gentle" interruption--global variables are unaffected, the stack is preserved, and the recovery stack, arguments and command lines are left intact. However, you can't resume execution of a program stopped by ATTN because all of the subroutine returns associated with that program are cleared. This does *not* apply to suspended programs, which can be resumed by CONT after any number of ATTN's or other errors.

As discussed in section 9.6.1, although there is no associated beep or message display, ATTN is treated as an error when it is pressed during command execution. The error number is zero, and the error message (as returned by ERRM) is the empty string "". Accordingly, 0 DOERR is the programmable form of ATTN. Executing 0 DOERR in a program (or in the command line) acts as though the ATTN key were pressed at the point in the program where the DOERR appears. The program stops, and all pending returns to procedures that called that program are cleared. Like ATTN, DOERR works in the current suspended program environment--if there are any suspended programs, they are unaffected. You can use 0 DOERR in a program to terminate program execution early, when some situation is encountered that makes further execution pointless. Usually this is done with an IF structure, such as

IF *situation-is-hopeless* THEN 0 DOERR END.

Note that 0 DOERR, like ATTN, clears special displays. If you want to abort a program and return an explanatory message, you can use DOERR with a string argument (section 12.2.1).

The only command that does affect suspended programs is KILL. KILL not only terminates the current program like 0 DOERR does, but also cancels *all* suspended programs and turns off the suspended program annunciator. All of the local memories associated with the suspended programs are removed. You can use KILL in a program, but that is a rather drastic thing to do, since in general a program doesn't "know" what programs are suspended when it is executed. It is better to use 0 DOERR in a program, then execute KILL manually if needed. Your most frequent use of KILL is likely to be to abort some half-finished program that you have been single-stepping, after you have found the problem you have been seeking.

### 12.2.2 Single-Stepping

The SST (single-step) operation is a combination of CONT and HALT that lets you execute a program one object at a time. Single-stepping is an important debugging tool, as it lets you follow the execution of a program step-by-step and discover where its calculations go awry.

To understand the mechanics of single-stepping, picture it as the equivalent of pressing ⬅ CONT when a HALT is temporarily inserted immediately after the next object in the program. From this model it follows that a program must be suspended before you can single-step it. The easiest way to do this is to enter the program, or its name if it is stored in a global or local variable, into level 1 and then press ≡DBUG≡ (*DeBUG*), found in the program control menu ( PRG ≡CTRL≡ ). This suspends the program before executing its first object. If instead you want to start single-stepping farther along in a program, you must include a HALT or a PROMPT at the point where you want to start stepping. Then when you execute the program, it will suspend execution after the HALT and you can proceed with single steps.

At each ≡SST≡ press, the HP 48 executes the next object in the suspended program, then halts and suspends the program again. To help you keep track of where you are in the program, each object is displayed in display line 1 after it is executed. If you single-step the » that ends the suspended program, the program completes execution and the suspended program environment is cleared. You can also execute CONT, which resumes and completes normal program execution.

A consequence of the behavior of SST as a one-step CONT is that each SST clears the current suspended program environment, then creates a new one after the step. This means that you can't cancel any stack effects of the object that was single-stepped by pressing ⬅ LAST STACK --the recovery stack present before the SST is deleted by the SST.

Some additional notes about SST:

- An IFERR structure is treated as a single object by SST. That is, when you press ≡SST≡ at an IFERR, the entire IFERR...THEN...ELSE...END structure is executed. If an error occurs between IFERR and THEN, the *then-sequence* between THEN and ELSE is executed; otherwise the *else-sequence* (if it is present) between ELSE and END is executed. The next ≡SST≡ will single-step whatever object follows the END. If you want to step through individual parts of the IFERR structure, you must insert HALT(s) within the structure.
- If a single-stepped object causes an error, the error is reported normally, but the single-step execution does not advance. If you press ≡SST≡ again, the HP 48 will attempt to execute the same object again. This gives you a chance to fix whatever it is that causes the error, such as a missing stack argument, then proceed with single-stepping.

## 12.3 Debugging

*Debugging* is the art of finding and removing programming errors--"bugs." The process ranges from simple visual inspection of a program to look for obvious errors, through careful single-stepping of parts of a program to watch for incorrect results at each stage.

Programming errors usually manifest themselves in two ways when you execute a program: either the program halts due to an error, or the program completes execution but returns incorrect results (which may be due to an incorrect algorithm, rather than a program defect). In either case, you know something is amiss--the trick is to find out where things go wrong in the program.

A good debugging technique for any programming language is to write the program correctly in the first place. This sounds facetious, but chances are, if you take extra time in designing a program before entering it into the calculator, you will save time in the long run by reducing the amount of debugging time. For HP 48 programs, a good approach is to write out a program of any complexity on paper, or better yet on a personal computer using a text editor, with the program formatting conventions discussed in section 1.4. Most importantly, as you add steps to a program, include comments or simple stack contents listings at least every few steps. This will help you get the program right in the first place; failing that, the comments stack listings will be your most valuable tool for debugging.

When a program fails, the first step in finding errors is to verify that you have entered the program correctly. If you know the correct checksum for the program, you can use BYTES (section 12.5.1) to check that the actual program's checksum matches the correct value. If you don't know the checksum, or if there is a discrepancy, then you should view the program using ⬅ EDIT or ➡ VISIT to see if it matches your program listing (this should happen automatically if you download the program from a computer file). If you have a printer, you can use PRVAR to print out a complete listing of the program. If these tests indicate that the program has been entered correctly, there must be a logical error in the program design.

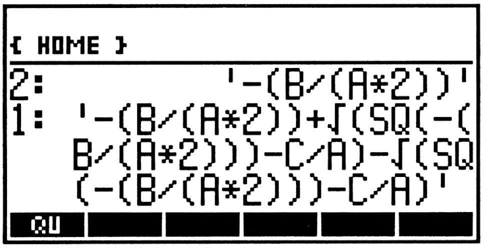
Before resorting to single-stepping, you may be able to apply the HP 48's symbolic capabilities to find an error. That is, even when a program is designed for purely numerical calculation, you can execute the program with symbolic arguments, then compare the symbolic results with the intended program algorithms (this is a good thing to do to verify any numerical program, not just when you're explicitly looking for an error).

For example, in section 12.4 we develop a program that finds the two roots of a quadratic equation $ax^2 + bx + c = 0$, where the three coefficients $a$, $b$ and $c$ are specified. The final version of the program is:

| QU | | | *Quadratic Root Finder* | | | 18E8 |
|---|---|---|---|---|---|---|
| | *level 3* | *level 2* | *level 1* | \| | *level 2* | *level 1* |
| | $a$ | $b$ | $c$ | ☞ | $x_1$ | $x_2$ |

| | |
|---|---|
| `<<` | $\| a \; b \; c \|$ |
| `3 PICK /` | $\| a \; b \; c/a \|$ |
| `SWAP ROT 2 * / NEG` | $\| c/a \; -b/2a \|$ |
| `DUP SQ` | $\| c/a \; -b/2a \; b^2/4a^2 \|$ |
| `ROT - √` | $\| -b/2a \; \sqrt{[(b/2a)^2-c/a]} \|$ |
| `DUP2 +` | $\| -b/2a \; \sqrt{[(b/2a)^2-c/a]} \; x_1 \|$ |
| `3 ROLLD -` | $\| x_1 \; x_2 \|$ |
| `>>` | |

Because this program involves a lot of stack manipulations, it's easy to lose track of the program flow as you develop it. Suppose that when writing the program, you miscounted the number of stack objects, and entered SWAP in place of the 3 ROLLD at the end. If you execute the program with numerical values for the coefficients, you will obtain incorrect results--but no indication that they are wrong. To guard against this, you can verify the program by executing it with symbolic arguments 'A', 'B', and 'C' (purging those variables first, if necessary, to ensure symbolic calculations). With these arguments, the bad version of the program returns



```
{ HOME }
2:          '-(B/(A*2))'
1: '-(B/(A*2))+√(SQ(-(
   B/(A*2)))-C/A)-√(SQ
   (-(B/(A*2)))-C/A)'
QU
```

By inspecting the level 1 result, you can see that the program correctly added the radical '√(SQ(-(B/(A*2)))-C/A)' to '-B/(A*2)', but then subtracted the same radical from the sum in level 1 rather than from the other '-B/(A*2)' in level 2. This suggests that the error is a stack error near the end of the program, and it is then a simple matter to figure out that the SWAP should have been 3 ROLLD.

The final resort in debugging is to single-step the program, from the beginning if necessary, until you discover an incorrect step. As described in section 12.2.2, in order to use SST, you must either use DBUG, to start single-stepping at the start of the program, or you must include a HALT (or a PROMPT) in the program at the point where you want to start single-stepping. If you do the latter, remember to remove the HALT after you have found the program error. If you are sure you have the solution, remember to remove the HALT as you edit the program. Otherwise, you can leave it in until after you verify the new version. When the program halts, press ⬏ CONT to resume.

In addition to DBUG and SST, the program control menu contains two other operations associated with single-stepping:

- SST↓ is a variation of SST that you may use when you want to step through a named program that is being used as a subroutine. That is, when the next object in a suspended program is the (global) name of a program, pressing SST↓ is equivalent to executing DBUG on that program, so that you can then single-step through that program. While single-stepping the subprogram, ⬏ CONT at any time, or SST on the final ≫, completes its execution so that subsequent single-stepping resumes in the original program.

  SST↓ applied to any other types of objects, or to names of variables that don't contain programs, has the same effect as SST. [Through versions A-E of the HP 48, there is an exception: applying SST↓ to the name of a *directory* just returns the directory object to the stack, whereas SST applied to the same name switches to that directory.]

- NEXT previews the next single-step by displaying the next object in a suspended program in the top display line (remember that the object displayed by SST or SST↓ is the object that *was* executed last). Due to the intricacies of HP 48 program execution, usually *two* objects are displayed if there is room on one line, but in some cases you will see only one object. (If the second object is a quoted name, you will see only the leading quote. The quote is actually a separate object from the name, but the two are generally treated as a single object.)

▪ *Example.* Find the error in the following program MINL. The program is designed to return the minimum value from a list of numbers. Starting with an initial value of MAXR, the program successively replaces the current value with the minimum (MIN) of the current value and the next number from the list. If you execute this program with a list of numbers, the program aborts with the Too Few Arguments error, identifying ROLLD as the culprit. To see what the correction should be, single-step through the program.

| MINL | *Minimum in a List (Bad version)* | | E7EB |
|---|---|---|---|
| | *level 1* | | *level 1* |
| | *{ numbers }* | ☞ | *minimum* |

```
≪ MAXR →NUM SWAP DUP SIZE
  1
  DUP ROT
  START
    GETI
    4 ROLL MIN 4 ROLLD
  NEXT
  DROP2
≫
```

| **Keystrokes:** | | **Results:** | |
|---|---|---|---|
| { 1 2 3 } VAR ′ ≡MINL≡ PRG ≡CTRL≡ ≡DBUG≡ | 1: | { 1 2 3 } | The argument list. |
| ≡SST≡ | 2: | { 1 2 3 } | |
| | 1: | 9.99999999999E499 | Initial "minimum" value. |
| ≡SST≡ (SWAP) ≡SST≡ (DUP) ≡SST≡ (SIZE) | 3: | 9.99999999999E499 | |
| | 2: | { 1 2 3 } | |
| | 1: | 3 | Number of elements in the list. |
| ≡SST≡ (1) ≡SST≡ (DUP) ≡SST≡ (ROT) | 5: | 9.99999999999E499 | |
| | 4: | { 1 2 3 } | |
| | 3: | 1 | Start value for GETI index. |
| | 2: | 1 | Start value for START. |
| | 1: | 3 | End value for START. |
| ≡SST≡ (START) | 3: | 9.99999999999E499 | Current minimum. |
| | 2: | { 1 2 3 } | |
| | 1: | 1 | Current GETI index. |

| | | | |
|---|---|---|---|
| SST (GETI) | 4: | 9.99999999999E499 | Current minimum. |
| | 3: | { 1 2 3 } | |
| | 2: | 2 | New GETI index. |
| | 1: | 1 | First list element. |
| SST (4) SST (ROLL) | 4: | { 1 2 3 } | |
| | 3: | 2 | GETI index. |
| | 2: | 1 | List element. |
| | 1: | 9.99999999999E499 | Current minimum. |
| SST (MIN) | 3: | { 1 2 3 } | |
| | 2: | 2 | GETI index. |
| | 1: | 1 | New minimum. |
| SST (4) SST (ROLLD) | | Too Few Arguments. | |

Here you can see exactly what is wrong. The program tries to execute 4 ROLLD with only three objects on the stack (attempting to put the objects back in the correct positions for the next iteration of the loop). The solution is to change the 4 ROLLD to 3 ROLLD. Here's the correct program listing:

| MINL | *Minimum of a List (Good Version)* | | | 5BF7 |
|---|---|---|---|---|
| | *level 1* | \| | *level 1* | |
| | { *numbers* } | ☞ | $x_{\min}$ | |

| | |
|---|---|
| « MAXR →NUM SWAP DUP SIZE | \| maxr { $x_i$ } n \| |
| 1 | Initialize $m$ (list index). |
| DUP ROT | Loop from 1 to n. |
| START | \| $x_{\min}$ { $x_i$ } $m$ \| |
| GETI | $x_m$ |
| 4 ROLL MIN 3 ROLLD | \| $x_{\min}$ { $x_i$ } $m$ \| |
| NEXT | |
| DROP2 | |
| » | |

You can verify that this version works correctly by using a symbolic input. For example,

{ A B C } MINL ☞ 'MIN(C,MIN(B,MIN(A,9.99999999999E499)))'.

## 12.4 Program Optimization

The fastest, most compact, and most memory efficient HP48 programs are usually those that carry out all of their calculations on the stack, using no local or global variables, and only fine-tuned RPN sequences for mathematics. These programs are also the hardest to write, since you have to keep track of the stack positions of everything, and spend time thinking about efficient ways to write the programs.

In this section, we will illustrate the process of *program optimization*, the process of revising working programs so that they execute faster or more efficiently. In general, program optimization involves

a. writing a first version of the program;

b. replacing parts of the program with more efficient sequences;

c. knowing when to stop optimizing and use the current version.

There is no fixed prescription for HP48 program optimization. There are two general purpose approaches that apply in most situations:

- Reduce the use of variables by keeping more objects on the stack.
- Replace long algebraic objects with RPN sequences that allow you to reuse intermediate results.

We will illustrate the application and effect of these two ideas in an extended program development example. Other methods are apparent in the program examples in this chapter and elsewhere in the book.

▪ *Example.* Develop and optimize a program QU that computes both roots $x$ of the quadratic equation $ax^2 + bx + c = 0$, where the (numerical) coefficients $a$, $b$, and $c$ are supplied as stack arguments. The mathematical algorithm is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Using local variables and algebraic objects, it is easy to translate the algorithm into a first version of the program. This version uses 151 bytes and takes .25 seconds to execute with arguments 8, −3, and 2:

*Version 1*:

```
<<                                                  | a b c |
  → a b c                                           Name the arguments.
  << '(-b+√(b^2-4*a*c))/(2*a)' EVAL                 x1
    '(-b-√(b^2-4*a*c))/(2*a)' EVAL                  x2
  >>
>>
```

To optimize this program, the first thing you might notice is that the solution algorithm can be written more compactly as

$$x = -b' \pm \sqrt{b'^2 - c'},$$

where $b' = b/2a$ and $c' = c/a$. You can incorporate this revised form into a new version of the program:

*Version 2*:

```
<<                                                  | a b c |
  → a b c
  << 'b/(2*a)' EVAL 'c/a' EVAL → b c                Store c' and b'.
    << '-b+√(b^2-c)' EVAL                           x1
      '-b-√(b^2-c)' EVAL                            x2
    >>
  >>
>>
```

Version 2 takes .23 seconds to execute, so compacting the algorithm has yielded a modest speed improvement. However, version 2 is 162.5 bytes, 11.5 bytes larger than version 1--the extra local variable structure has cost more in program size than the algorithm compaction saved. As the next step in optimization, you can eliminate that extra structure by computing $b'$ and $c'$ directly from the original stack arguments:

*Version 3*:

```
<<                                                  | a b c |
  3 PICK /                                          | a b c/a |
  SWAP ROT 2 * /                                    | c/a b/2a |
  → c b                                             Store c' and b'.
  << '-b+√(b^2-c)' EVAL                             x1
      '-b-√(b^2-c)' EVAL                            x2
  >>
>>
```

Version 3 occupies 118.5 bytes of RAM, which is 32.5 bytes smaller than version 1. It is also slightly faster (.21 seconds) than version 2. To improve on this version, you can observe that the two algebraic objects in the program are very similar, which means that the program performs some arithmetic twice. You should therefore be able to improve matters by breaking up the algebraic objects into smaller parts that are common to both expressions.

*Version 4*:

| | |
|---|---|
| `«` | $\vert a \;\; b \;\; c \vert$ |
| `3 PICK /` | $\vert a \;\; b \;\; c/a \vert$ |
| `SWAP ROT 2 * /` | $\vert c/a \;\; b/2a \vert$ |
| `→ c b` | Store $c'$ and $b'$. |
| `« '-b' '√(b^2-c)' DUP2` | Make 2 copies of the partial results. |
| `+ EVAL` | $\vert -b \;\; \sqrt{(b^2-c)} \;\; x_1 \;\; \vert$ |
| `3 ROLLD - EVAL` | $x_2$ |
| `»` | |
| `»` | |

Version 4 has shrunk the program size to 100 bytes, but execution has slowed to .28 seconds. The slowdown has resulted from a subtle cause: the final + and − that combine the partial results are acting on symbolic arguments, returning symbolic results (which are then evaluated into the final numeric results using EVAL). Symbolic addition and subtraction are intrinsically slower than numeric arithmetic. You can fix this problem with a simple rearrangement so that the partial results '-b' and '√(b^2-c)' are evaluated *before* they are added or subtracted:

*Version 5*:

| | |
|---|---|
| `«` | $\vert a \;\; b \;\; c \vert$ |
| `3 PICK /` | $\vert a \;\; b \;\; c/a \vert$ |
| `SWAP ROT 2 * /` | $\vert c/a \;\; b/2a \vert$ |
| `→ c b` | Store $c'$ and $b'$. |
| `« '-b' EVAL '√(b^2-c)'` | |
| `EVAL DUP2` | Make 2 copies of the partial results. |
| `+` | $\vert -b \;\; \sqrt{(b^2-c)} \;\; x_1 \vert$ |
| `3 ROLLD -` | $x_2$ |
| `»` | |
| `»` | |

Version 5 is the same size as version 4, but it executes in .14 seconds, which is the fastest time yet.

The progress made so far in optimizing this program suggests completing the process of converting the algebraic expressions into pure stack arithmetic, eliminating the use of variables.

*Version 6 (final version):*

| QU | | | *Quadratic Root Finder* | | | 18E8 |
|---|---|---|---|---|---|---|
| | *level 3* | *level 2* | *level 1* | \| | *level 2* | *level 1* |
| | $a$ | $b$ | $c$ | ☞ | $z_1$ | $z_2$ |

| | |
|---|---|
| `«` | $\| a \; b \; c \|$ |
| `3 PICK /` | $\| a \; b \; c/a \|$ |
| `SWAP ROT 2 * / NEG` | $\| c/a \; -b/2a \|$ |
| `DUP SQ` | $\| c/a \; -b/2a \; b^2/4a^2 \|$ |
| `ROT - √` | $\| -b/2a \; \sqrt{}[(b/2a)^2-c/a] \|$ |
| `DUP2 +` | $\| -b/2a \; \sqrt{}[(b/2a)^2-c/a] \; x_1 \|$ |
| `3 ROLLD -` | $\| x_1 \; x_2 \|$ |
| `»` | |

Version 6 requires only 57.5 bytes, and executes in .10 seconds. This represents a 62% reduction in program size, and a 2.5× speed improvement over version 1.

The lesson here is not that algebraic objects evaluate numerically more slowly than their RPN sequence equivalents. The execution time difference between, for example, `'1+2+3+4+5+6+7' EVAL` and `1 2 + 3 + 4 + 5 + 6 + 7 +`, is only a few milliseconds--the time required to put the algebraic object on the stack. Instead, the point is that RPN lets you avoid repeating mathematical operations by breaking calculations into unique elements, and then duplicating and reusing the results. Furthermore, it is always faster for a program to leave results on the stack rather than storing them in variables, and similarly faster to retrieve arguments from the stack than to recall them from variables.

## 12.5 Memory Use

To help you in optimizing programs for minimum memory size, Tables 12.1 and 12.2 list the memory size of various objects and structures included in a program. Table 12.1 shows the memory occupied by program structures, not counting the objects that are entered between the structure words. Table 12.2 (next page) lists the memory size of individual objects.

There are a few exceptions to the sizes listed in Table 12.2, since the HP 48 has built in certain commonly used objects, to save memory. For example, the real number 1 uses only 2.5 bytes, instead of the 10.5 bytes normally used by a real number. Similarly, each of the following built-in objects uses 2.5 bytes:

- Real integers from −9 through +9.
- The real constants 3.14159265359 (π), 2.71828182846 (*e*), 1E−499 (MINR), and 9.99999999999E499 (MAXR).
- The complex constant (0,1) (*i*).
- The null string "".

**Table 12.1. Program Structure Sizes**

| *Structure* | *Size (bytes)* |
|---|---|
| IF ... THEN ...* END | 12.5† |
| IF ... THEN ...* ELSE ...* END | 20† |
| IFERR ...* THEN ...* END | 17.5† |
| IF/IFERR ...* THEN ...* ELSE ...* END | 25† |
| CASE ... THEN ...* END ... END | 20† |
| *(additional)* THEN ...* END | 10† |
| DO ... UNTIL ... END | 7.5† |
| WHILE ... REPEAT ...* END | 12.5† |
| START/FOR ... NEXT/STEP | 5 |
| → ... « ... » | 7.5 |
| → ... ' ... ' | 7.5 |

†A program savings of 5 bytes in each instance is obtained whenever any of the structure sequences marked with an asterisk ( ...* ) consists of one object.

### 12.5.1 Using BYTES

The easiest way to determine the memory size of an object is to execute BYTES with the object as its argument. BYTES returns (level 1) the actual memory size occupied by the object, plus a *checksum* (level 2). The checksum, a four-digit binary integer, is computed essentially by adding up the object's memory bit pattern to produce a 16-bit number. The chance of two different objects having the same checksum is only 1 in 65535, so the checksum provides an excellent test of the identity of two objects. This is most useful to verify that you have entered a program correctly according to a listing; it is easy to make an error that does not affect a program's size, but any error is very likely to affect the checksum.

BYTES treats global names slightly differently than other object types. Instead of

**Table 12.2. Object Sizes**

| Object Type | Size (bytes) |
|---|---|
| Real number | 10.5 |
| Complex number | 18.5 |
| String | 5 + *number of characters* |
| Vector | 12.5 + 8 × *number of elements* |
| Complex vector | 12.5 + 16 × *number of elements* |
| Matrix | 15 + 8 × *number of elements* |
| Complex matrix | 15 + 16 × *number of elements* |
| List | 5 + *included objects* |
| Unquoted global or local name | 3.5 + *number of characters* |
| Quoted global or local name | 8.5 + *number of characters* |
| Program | 12.5 + *included objects* |
| Algebraic | 5 + *included objects* |
| Binary Integer | 13 |
| Graphics Object | 10 + *rows* × CEIL(*columns*/8) |
| Tagged Object | 3.5 + *number of tag characters* + *untagged object* |
| Unit Object | 7.5 +: |
| real magnitude | 2.5 or 10.5 |
| each prefix | 6 |
| each unit name | 5 + *number of characters* |
| each *, ^, or / | 2.5 |
| each exponent | 2.5 or 10.5 |
| XLIB name | 5.5 |
| Directory | 6.5 + *included variables* |
| Backup Object | 12 + *number of name characters* + *included object* |
| Command | 2.5 |

returning the size and checksum of the name, BYTES computes those parameters for the object stored in the named global variable. Furthermore, the memory size returned is the total size of the variable, which includes the memory for the object itself, plus an additional amount for the variable structure. Specifically, the variable "overhead" is 4.5 bytes plus one byte for each character in the variable name. The memory used by a stored object is the same as the amount listed in Table 12.1, with one exception. Programs require 10 bytes (plus the included objects) rather than the 12.5 bytes listed in the table for programs within programs.

## 12.6 Obtaining Input.

In programs and in manual operations, the stack is the basic input/output mechanism. You can enter all the data a program needs as stack objects, execute the program, then read its results from the stack. This works fine under two conditions: first, you know in advance what objects to enter at the start, and second, there are not so many inputs or outputs that you lose track of which is which among the stack objects. In the following sections we will consider several methods for improving on this bare-bones approach.

### 12.6.1 Halting for Input

The most flexible method for obtaining input after a program has begun execution is to include a HALT or PROMPT in the program to suspend its execution (section 12.2). While the program is suspended, you have complete access to the calculator's resources, including the stack and variables. You can use those resources to calculate or otherwise produce the input. For example, if you want to enter √3/2, you can compute it by any means you want, such as '√3/2' [EVAL], rather than having to type in the digits of the number. You can store values in variables, set flags, or even run other programs to produce results that then become inputs for the suspended program. When you have entered those inputs by whatever means, you then press [⬑] [CONT] to resume the suspended program.

PROMPT suspends a program and displays a one- or two-line message in the status area (in the medium font). It is equivalent to 1 DISP 1 FREEZE HALT, with the important addition that the displayed message remains visible during command line entry instead of disappearing at the next keystroke as a FREEZE display does. PROMPT's display persists until the next ENTER (and all of the execution caused by the ENTER is completed), or until some other display operation replaces it. In particular, the prompt is visible during command line entry, which is convenient when you are typing in the input indicated by the prompt.

▪ *Example.* The following sequence prompts successively for length, width, and height, as might be needed by a program that computes the volume of a box:

"Enter length:" PROMPT "Enter width:" PROMPT "Enter height:" PROMPT.

Upon execution, the sequence halts and displays "Enter length". At this point, you enter a value for the length, and press ⬑ CONT . Then the display shows "Enter Width", and so on. Since PROMPT allows a two line message, the above sequence could be more specific by including "and press CONT" in the prompts. This suggests creating a general purpose input utility to save repeated entry of the same text:

| PROMPTCONT | *Prompt with CONT Display* | | E4BD |
|---|---|---|---|
| | *level 1* | *level 1* | |
| | *"text"* | ☞ | |

| | |
|---|---|
| « "Enter " SWAP + | Prepend "Enter " to the text. |
| 10 CHR + | Add a newline. |
| "and press CONT" + | Append the second line. |
| PROMPT | Stop for input. |
| » | |

(You could embed a newline directly in on or the other of the two strings in the program, but using 10 CHR + instead makes program editing easier because you don't have to worry about invisible space characters at the end of a line).

Using PROMPTCONT, the sequence to prompt for volume parameters becomes:

"length" PROMPTCONT "width" PROMPTCONT "height" PROMPTCONT.

PROMPTCONT fits the definition of a *subroutine*, which is a program that performs a task common to many programs but which doesn't have much value for manual execution. There are many ways to extend this subroutine to do even more standard input tasks. For example, a good program, after obtaining manual input, checks that input to verify that it is valid for the remainder of the program, and warns and reprompts you if it is not. The next program, CHKINPUT, demonstrates this process.

| CHKINPUT | *Prompt and Check Input* | | | 2C9F |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | *"text"* | *≪ test ≫* | ☞ | *object* |

```
≪ → prompt test                      Save the test program.
  ≪ WHILE prompt PROMPTCONT           Get the input.
      test EVAL NOT                   Exit if the input is valid.
    REPEAT
      "Invalid Input" 10 CHR +
      1 DISP                          Display error message.
      200 .3 BEEP .7 WAIT             Beep and wait .7 seconds, then repeat.
    END
  ≫
≫
```

CHKINPUT requires two arguments: the first (level 2) is the prompt string as used by PROMPTCONT, and the second is a program to test the input. CHKINPUT does not finish until it can return a valid object as determined by the test program, which should take one object from the stack and return the object and *true* (1) if it is valid, and *false* (0) otherwise. For example, when prompting for box dimensions, you might want to accept only real numbers with values between 1 and 10. The test program then would look like this:

```
≪ IF DEPTH                                 If the stack is not empty...
  THEN → object                            Save the object.
    ≪ IF object TYPE NOT                   If the object is a real number,
       THEN
         IF 'object≥1 AND object≤10'       and it is in the valid range,
         THEN object 1                     Then return the object and true.
         ELSE 0                            Return false (out of range).
         END
       ELSE 0                              Return false (not a real number).
       END
    ≫
  ELSE 0                                   Return false (empty stack).
  END
≫
```

CHKINPUT is an example of the use of a program as an argument, which is discussed in more detail in section 12.8.

### 12.6.1.1 Verbose Prompts

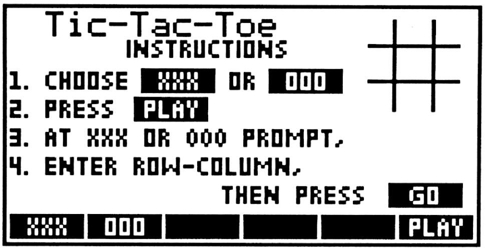
By definition, PROMPT is limited to two lines of prompt text, so that text can fit within the status area of the display. You can also use the stack area of the display for additional prompt text by preceding the execution of PROMPT with the use of DISP to display text in lines 3 - 7. In that case the status area text will remain until ENTER, but the stack area prompts will disappear at the next keystroke.

For even more flexible prompt displays, you can use HALT instead of PROMPT, preceding the HALT with any of the display commands described in Chapter 10, including FREEZE to preserve the special display when execution halts. The entire prompt display is replaced by the standard display at the next keystroke.

The follow program is an example of an elaborate prompt intended to begin a tic-tac-toe game program. The prompt mixes text, graphics, and a menu:

```
<< ERASE                                                               Clear the graph screen.
                                                                       Display text:
  PICT { #10d  #0 } "Tic-Tac-Toe"  2  →GROB  REPL
  PICT { #21  #8 }  "Instructions"  1  →GROB  REPL
  PICT { #0  #17d } "1. Choose^^XXX^^or^^OOO"  1  →GROB  REPL          Carets "^" here indicate
                                                                       space characters.
  PICT { #0  #25d } "2. Press^^PLAY"  1  →GROB  REPL
  PICT { #0  #33d } "3. At XXX or 000 prompt,"  1  →GROB  REPL
  PICT { #0  #41d } "4. enter row-column,"  1  →GROB  REPL
  PICT { #60d  #49d } "then press^^^GO"  1  →GROB  REPL
                                                                       Invert key labels:
  PICT { #73d  #16d } DUP2 { #93d  #22d } SUB NEG REPL
  PICT { #37d  #16d } DUP2 { #57d  #22d } SUB NEG REPL
  PICT { #35d  #24d } DUP2 { #55d  #30d } SUB NEG REPL
  PICT { #107d  #48d } DUP2 { #127d  #54d } SUB NEG REPL
                                                                       Draw grid:
  { #108d  #27d } { #108d  #2d } LINE
  { #119d  #27d } { #119d  #2d } LINE
  { #126d  #9d } { #101d  #9d } LINE
  { #101d  #19d } { #126d  #19d } LINE
  { XXX  OOO  ""  ""  ""  PLAY } TMENU                                 Make temporary menu.
  PICT  RCL  ERASE  →LCD  3  FREEZE                                    Display the prompt.
  HALT
>>
```

Executing the program produces this display:



```
Tic-Tac-Toe
      INSTRUCTIONS
1. CHOOSE XXX OR OOO
2. PRESS PLAY
3. AT XXX OR OOO PROMPT,
4. ENTER ROW-COLUMN,
            THEN PRESS GO
```

### 12.6.1.2 Prompting with Menus

The tic-tac-toe example above includes a temporary menu as as part of its prompt. Using a menu is a important enhancement to ordinary display prompting, since the menu labels themselves can act as instructions, and they remain visible indefinitely. Furthermore, a menu key can include a CONT as part of its definition, so that pressing a menu key not only indicates a choice, but also resumes execution of a suspended program, all in one operation.

While you can use any built-in menu or the VAR or CST menus for prompting, a temporary menu activated by TMENU is particularly useful for this purpose. A temporary menu has all of the flexibility of a custom menu, but does not replace the normal custom menu defined by the variable CST. The construction of menus by TMENU and MENU is described in section 7.3; here we will focus on the use of CONT directly or indirectly in a custom menu.

In the prompting examples so far, resuming a program suspended for input has required an explicit press of [⬅] [CONT] to resume execution after the input objects are entered. However, because CONT is a programmable command, you can include it as part of a menu key definition and eliminate the need to press an additional key. Incorporating the continue operation into a menu is also a good practice because it allows you to focus entirely on the menu for instructions without having to think about how to resume a program.

The tic-tac-toe prompt sequence displays a temporary menu defined by { XXX OOO }. Presumably XXX and OOO are the (global) names of subroutines that store the choice of whether you want to play X's or O's. Any easy way to record such information is with a flag; for example, XXX might name the program « 1 SF » and OOO is « 1 CF ». But in this case there is no reason not to continue the main program as soon as XXX or OOO is executed, so XXX can be « 1 SF CONT » and OOO can be « 1 CF CONT ». CONT should always be the last object in such programs, since any objects following CONT will never be executed. [*Last object* also means last in the sense that there are no pending returns to any other programs. When CONT is executed, *all* currently executing programs are terminated, and the most recently *suspended* program is resumed.]

Actually, you don't need global variables at all for the menu key subroutines, since the custom menu system (section 7.3.3) allows you to associate unnamed programs with menu keys. That is, the temporary menu list in the example might be:

```
{ { "XXX"  « 1  SF  CONT » }  { "OOO"  « 1  CF  CONT » } }
```

In many programs, you may wish to enter several quantities during the same program halt. In such cases, you might use separate menu keys for each item, then have a single menu key to resume the program. To return to the box dimensions example, the input sequence could look like this:

| | |
|---|---|
| "Enter length, width"  10  CHR  + | |
| "and height, press GO"  + | Two-line prompt string. |
| { { "LENG"  « 'L'  STO » } | LENG key. |
|  { "WIDTH"  « 'W'  STO » } | WIDTH key. |
|  { "HT"  « 'H'  STO  » } | HT key. |
|  "" | Blank key. |
|  { "GO"  CONT } | GO key. |
| } | End of temporary menu list. |
| TMENU  PROMPT | Activate the menu and halt. |

This method has the advantage that you can enter the input values in any order, and can re-enter a value if you change your mind. Only when you press GO are your current entries locked in. However, you may not wish to use global variables to hold the box dimensions; the following modification uses local variables during entry, then returns the three values to the stack before exiting:

| | |
|---|---|
| 0  0  0  →  l  w  h | *Initialize* l, w, and *h*. |
| « | |
|   "Enter length, width"  10  CHR  + | |
|   "and height, press GO"  + | Two-line prompt string. |
|   { { "LENG"  « 'l'  STO » } | LENG key. |
|    { "WIDTH"  « 'w'  STO » } | WIDTH key. |
|    { "HT"  «  'h'  STO  » } | HT key. |
|    "" | Blank key. |
|    { "GO"  CONT } | GO key. |
|   } | End of temporary menu list. |
|   TMENU  PROMPT | Activate the menu and halt. |
|   l  w  h | Return the parameters to the stack. |
| » | |

### 12.6.2 Protected Entry.

An important advantage of suspending a program for input is that you can perform arbitrary operations while the program is suspended. However, in many situations this capability can actually be a disadvantage. Since you have access to the stack and memory, you can accidentally or deliberately alter or remove objects used by the program. There is nowhere a program can save information that is completely "safe" while the program is suspended. The best recourse is to save objects in local variables with offbeat names that are unlikely to be used inadvertently. For example, if all of a program's current parameters are on the stack, the following sequence protects them while the program is suspended:

DEPTH →LIST → στασθ << *procedure* >>

*Procedure* must contain both the prompt/input sequence and the stack retrieval sequence (e.g. στασθ LIST→ DROP).

There are two alternative means of obtaining input, in which the stack and other calculator resources are not accessible during entry:

- Use INPUT to restrict entry to the command line.
- Use KEY to restrict entry to single keystrokes.

We will examine these methods in the next two sections.

### 12.6.3 Using INPUT.

INPUT is a special data entry command that activates the command line for entry. Further program execution is postponed, although the program is not suspended in the sense of HALT or PROMPT (in particular, pressing $\boxed{\text{ATTN}}$ twice terminates the program). INPUT finishes, and automatically resumes program execution, when you press $\boxed{\text{ENTER}}$ ; since program entry mode (PRG) is turned on, $\boxed{\text{ENTER}}$ is the only option. The command line is not executed; instead its text content is returned as a string object for use by the remainder of the program.

INPUT also provides the following features:

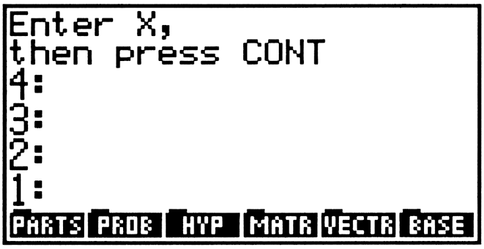
- Optional multi-line text prompts.
- The ability to "pre-load" the command line with objects to assist with entry.
- Control over command line cursor type and position, and entry mode.
- The choice of whether or not to use normal command line interpretation.

You can select one or more of these options by means of the two arguments for INPUT, which may be either two strings, or a string (level 2) and a list. The string in level two specifies a prompt that appears in the medium font in the stack display area (starting in display line 3); this prompt persists during keystroke entry, until [ENTER] terminates the INPUT operation. You can create a prompt of up to three 22-character lines, by including one or two newline characters in the level 2 string.

The level 1 argument can also be a string, which is used as the initial contents of the command line. For example, the following sequence prompts for a new value for a variable X:

```
"Enter X:"  X  STD  →STR  INPUT  OBJ→  'X'  STO
```

Here we have used the current value of X as the initial contents of the command line. When the sequence is executed with 100 stored in X, the following display appears:



```
Enter X,
then press CONT
4:
3:
2:
1:
```

At this point, you can edit the current value, or press [ATTN] to clear the command line and type a new value for X. (If you press [ATTN] again, or any time the command line is empty, the program is aborted.) Pressing [ENTER] returns the contents of the command line to level 1 as a string object, and the program resumes execution with OBJ→.

In the example, the command line initially contains the level 1 string argument, with the insert cursor ◊ at the end of the string; upon [ENTER], the command line string is pushed as is onto the stack. For additional control over the INPUT command line, you can use a list as the level 1 argument. The list can contain one or more elements (the order does not matter):

- To specify the initial command line text, include a string object (if this is the only element, then you can use the string object by itself as in the preceding example). The string may contain newlines, to produce a multi-line entry. If no string is specified, the command line will initially be empty.

- To place the cursor at a particular position in the command line, include a real integer to specify the character position, counting from the start of the command line (and including newlines in the count). Character number 0 specifies that the cursor is to be placed at the end of the command line (to the right of the last character). Alternatively, you can use a list { *row column* } that specifies the row (counting from the top down) and column (counting from the left) position for the cursor. Column number 0 indicates that the cursor is to be placed at the end of the specified row; row 0 specifies the last row of the command line. If no cursor position is specified, the cursor will be placed at the end of the command line.

  You can also use the cursor position object to select replace entry mode, in which typed characters overwrite the characters at the cursor. This is done by entering a *negative* character or row number. Positive numbers specify the default insert mode.

- To activate the command line in algebraic-program entry mode (ALG PRG), include the name ALG.

- To activate alpha-lock, include the name $\alpha$.

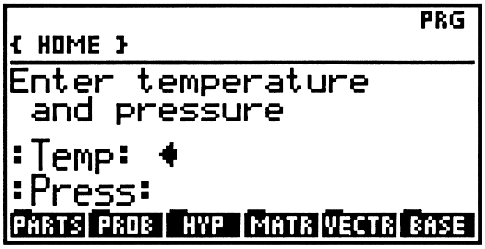
- Since the command line contents are returned to the stack as a string, INPUT normally does no syntax checking on the string following ENTER . However, if you include the name V (for verify), the string is checked for valid object syntax. If there is a syntax error, the HP48 beeps and reactivates the command line with the highlighted error position, just as with ordinary command line entry. This is useful when you are using INPUT to enter objects in their standard form, i.e. you follow INPUT with OBJ→ to convert the result string to objects. If you don't use the V option and an entry has invalid object syntax, OBJ→ will error and abort the program. The V option allows the HP48 to catch such errors before the program resumes.

Note that the symbols $\alpha$, ALG, and V are entered into the INPUT strings as name objects--without any delimiters. However, these names are not executed, so it doesn't matter if you have variables with those names.

▪ *Example.* (In the following sequence, spaces within strings are marked by "^" characters for clarity.)

| | |
|---|---|
| "Enter temperature<br>ˆand pressure" | Two-line prompt string. |
| { ":Temp:ˆ<br>:Press:ˆ" | Initial command line text. |
| { 1 0 } | Cursor at end of first line. |
| V<br>} | |
| INPUT | Stop for input. |
| OBJ→ | Convert entered text into objects. |

Executing this sequence yields the following display:



```
                                   PRG
{ HOME }
Enter temperature
 and pressure
:Temp: ◆
:Press:
PARTS PROB HYP MATR VECTR BASE
```

The cursor is at the end of the first row, following the tag :Temp: that indicates that a temperature should be entered. After entering the temperature, pressing [▽] moves the cursor to the second row, following the :Press: tag. For example, these keystrokes

300_ [◁] [UNITS] [NXT] ≡TEMP≡ ≡ K ≡ [▽]
100000_ [◁] [UNITS] [NXT] ≡PRESS≡ ≡ PA ≡
[ENTER]

return the tagged object :Temp:300_K to level 2, and :Press:100000_Pa to level 1. Here the primary purpose of the tags is to indicate the command line order of the entries; the fact that the resulting stack objects are tagged will not interfere with any subsequent program calculations.

Unless you select the verify (V) option, INPUT does not require any structure or syntax for the text returned from the command line. This means, for example, that you can use INPUT to enter strings or names without quotes, binary integers without #'s etc. (see also section 7.4.1). The ≡NEW≡ keys in the PLOT, SOLVE, and STAT menus actually use INPUT to prompt for and enter names without requiring quotes.

### 12.6.4 Keystroke Input

All of the input methods outlined so far are designed for object entry, and permit multiple-keystroke entry while waiting for a particular key press (e.g. ENTER or ◁ CONT ) to resume program execution. The HP48 also provides two commands for the entry of individual keystrokes--either where a single key or key combination automatically resumes program execution, or without stopping program execution at all.

#### 12.6.4.1 KEY

When you press an HP48 key, a code representing that key is entered into a special memory location called the *key buffer*. Each time the HP48 completes any operations in process, it checks the key buffer to see if any key codes were recorded while it was busy. If so, it removes the codes one at a time (in the same order in which they were pressed), then performs whatever operations are associated with the keys. This two-stage key processing is responsible for the HP48's "type-ahead" capability, whereby up to 15 keystrokes can be stored in the buffer while the busy annunciator is on.

Programs can check and act on the contents of the key buffer by executing KEY. KEY attempts to remove the oldest key code from the key buffer. If there are codes in the buffer, KEY returns a two-digit real number key code *rc* to level 2 and a *true* flag (1) to level 1. The first digit *r* of the key code is the keyboard row of the key; *c* is the column. If there are no codes available in the key buffer, KEY returns only a false flag (0) to level 1, and no key code. Note that the key code does not include a key plane (shift) digit like that used by ASN (section 7.2.1) and WAIT (section 12.6.4.2); the shift keys act like any other keys in this case and return a two-digit code.

By using KEY, programs can accept keyboard input, on a key-by-key basis, without actually halting execution. If a program is to pause indefinitely to wait for a keystroke, then 0 WAIT is a better choice than KEY, since during the execution of 0 WAIT the HP48 is in a low power consumption state (and can even turn off after 10 minutes of inactivity). KEY is better suited for requirements such as these:

- To provide for interrupting a long-running program in a manner that will let the program save enough information to restart at a later time.
- To have a program wait for a key only for a fixed time, then continue whether or not a key is pressed.

The first of these cases is illustrated by the program KEYHALT. If you interrupt a program with ATTN, the program stops immediately, with no chance to exit gracefully. You could put embed the entire program in an IFERR structure that traps ATTN, but that still does not provide any information about the state of the program when it is interrupted. Instead, you might include (the name) KEYHALT inside any time-consuming iterative loops in the program. Then, if you press any key other than [ATTN] while the program is running, KEYHALT saves the current stack in a local variable and halts. To resume the program, you need only press [◁] [CONT] .

| KEYHALT | *Halt if a Key is Pressed* | 5055 |
|---|---|---|

| | |
|---|---|
| `« IF KEY` | |
| `  THEN DEPTH →LIST → στασθ` | Save the stack. |
| `  « "Program interrupted."` | First line of prompt. |
| `    10 CHR +` | Add a newline. |
| `    "Press CONT to resume." +` | Second line of prompt. |
| `    PROMPT` | Suspend the program. |
| `    στασθ OBJ→ DROP` | Restore the stack |
| `  »` | |
| `  END` | |
| `»` | |

The next program example, KEYTIME, waits a specified amount of time (specified in HH.MMSSSS format) for a keystroke. If one is detected, then the program returns the keycode and *true*. If no key is pressed in the indicated time interval, KEYTIME returns false.

| KEYTIME | *Wait a Specific Time for A Key* | | | 62BE |
|---|---|---|---|---|
| | *level 1* | \| | *level 2* | *level 1* |
| | *hh.mmssss* | ☞ | *rc* | 1 |
| | *hh.mmssss* | ☞ | | 0 |

| | |
|---|---|
| `« TIME → δt t` | Save the time interval, start time. |
| `  « WHILE TIME t HMS- δt <` | *True* if elapsed time < δt. |
| `      IF KEY` | If a key was pressed, |
| `      THEN SWAP DROP 1 0` | then replace time flag with *false,* |
| | return *true* key flag. |
| `      ELSE 0 SWAP` | Else, return *false* key flag. |
| `      END` | |
| `    REPEAT DROP` | Drop the key flag and try again. |
| `    END` | |
| `  »` | |
| `»` | |

#### 12.6.4.2 WAIT

The WAIT command nominally is designed to produce a simple pause in program execution. *x* WAIT produces a pause of *x* seconds, during which program execution does not proceed, but the display is not changed and no key entry is processed (the key buffer will still accumulate key codes). A common application of WAIT is to display messages or other pictures while a program is running. If your program shows a series of messages, you can put a WAIT after one or more of the display commands to ensure that the message remains visible long enough to be read conveniently.

It is also possible to make WAIT pause program execution indefinitely, by using 0 or −1 as its argument. For 0, the current display is not affected by WAIT; for −1, the menu labels are updated to reflect the current menu. In either case, execution resumes only when a key is pressed, when WAIT returns the corresponding key code to level 1. The key code returned by WAIT is a three-digit code *rc.p* like that used by ASN (section 7.2.1), where *r* is the key row, *c* the column, and *p* the key plane. Note that 0 or −1 WAIT only terminates when a "complete" key is entered, either a non-shift key by itself or such a key preceded by one or more shift keys.

#### 12.6.4.3 ATTN

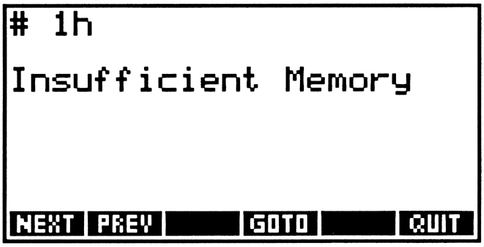
For the sake of KEY and WAIT (0 or −1 arguments), ATTN is not an ordinary key that returns a key code. Pressing ATTN always interrupts program execution, even if you have redefined this key and activated user mode. The only way for a program to treat ATTN as an ordinary key to use an error trap that checks for error 0, and returns the key code 91 when that error occurs. An example of such processing is given in the program ASN41, in section 7.2.1.1.

#### 12.6.4.4 An Input Programming Example

The program MSGSHOW listed below allows you to display all of the HP 48's built-in messages, both error messages and prompting text. The program itself is of limited practical value, but it does illustrate a number of programming techniques:

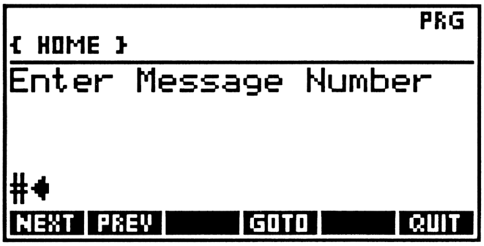
- The use of WAIT to obtain single-keystroke input.
- The use of INPUT to enter a hexadecimal number using a command line preloaded with the # and h delimiters.
- An error trap to handle ATTN.
- A CASE structure.
- A temporary menu (section 7.3).
- Extensive use of local variables.

MSGSHOW starts with the following display:


```
# 1h
Insufficient Memory
```

This shows the message number and text of the first HP 48 message, with a menu of choices:

- ≡NEXT≡ displays the next message. The program contains a list containing sublists defining the message number ranges for which there are valid messages; if ≡NEXT≡ advances past the end of one of the ranges, it skips to the start of the next range. At the last message (#D04h), ≡NEXT≡ skips back to message 001.
- ≡PREV≡ moves backwards through the messages, in the same manner as ≡NEXT≡ .
- ≡GOTO≡ allows you to skip directly to any message. It produces the following display:


```
                    PRG
{ HOME }
Enter Message Number


#
```

  Here you may enter any message number, followed by ENTER . If the number is in an allowed range, the corresponding message is displayed; otherwise No Such Message is displayed briefly, and you are prompted for a new number. You can cancel the change by pressing ATTN to clear the command line, then ENTER .

- ≡QUIT≡ exits from the program, and restores the original menu.

MSGSHOW *Show Messages* FECE

```
«  RCLMENU  HEX  CLLCD                              Get current menu.
  { "NEXT"  "PREV"  ""  "GOTO"  ""  "QUIT" }  TMENU  Set temporary menu.
  {{ #1h  #10h }  { #101h  #122h }  { #124h  #13Dh }  List of valid message numbers ranges.
    { #201h  #208h }  { #301h  #305h }
    { #501h  #506h }  { #601h  #62Eh }
    { #A01h  #A06h }  { #B01h  #B02h }
    { #C01h  #C17h }  { #D01h  #D04h }
  }  DUP  SIZE  1  0  →  L  nmax  N  exit           Save list, exit flag.
  « L  1  GET  OBJ→  DROP  OVER  →  imin  imax  I   Initialize message number, limits.
    « DO  I                                         Start indefinite loop.
        IFERR  DOERR                                Do the Ith error.
        THEN ERRN 1 DISP ERRM 10 CHR + 3 DISP       Show the message.
        END
        -1  WAIT  →  keycode                        Get a key.
        « CASE                                      Actions for various keys:
            'keycode==12.1'                         PREV key.
            THEN  'I'  1  STO-                      Decrement I.
              IF  'I<imin'                          Out of range?
              THEN                                  Then go to the next range.
                IF  'N==1'                          First range?
                THEN  nmax  'N'  STO                Then go to the last.
                ELSE  'N'  1  STO-                  Otherwise decrement N.
                END  L  N  GET  OBJ→
                DROP  DUP  'I'  STO  'imax'  STO
                'imin'  STO                         Reset the limits.
              END
            END
            'keycode==14.1'                         GOTO key.
            THEN
              WHILE  "Enter Message Number"         Entry loop.
                1  FREEZE  { "#h"  2  V  α }  INPUT  Get a message number.
                IF  ""  OVER  SAME                  If the command line is null,
                THEN  DROP  I  N  0                 go back to the main loop.
                ELSE  OBJ→  →  m                    Otherwise, see if it's a valid number:
                  « 1  1  L  SIZE
                    FOR  j  L  j  GET  OBJ→  DROP
                      IF  m  ≥  SWAP  m  ≤  AND
                      THEN  DROP  m  j  0  99
                        'j'  STO
                      END
                    NEXT
                  »
                END
```

*(continued on next page)*

MSGSHOW *continued from previous page:*

```
          REPEAT "No Such Message"              Invalid message; try again.
            10 CHR + 1 DISP 300 .3 BEEP
          END 'N' STO 'I' STO L N GET
            OBJ→ DROP                           Update counters and ranges.
          'imax' STO 'imin' STO CLLCD
        END
        'keycode==16.1' THEN 1 'exit' STO END   EXIT key.
        'keycode≠11.1' THEN 300 .2 BEEP END     Beep unless NEXT key.
        'I' 1 STO+                              Increment I.
        IF 'I>imax'                             Out of range?
        THEN                                    then goto the next range.
          IF 'N==nmax'                          Last range?
          THEN 1 'N' STO                        Goto the first range.
          ELSE 'N' 1 STO+
          END L N GET OBJ→ DROP
          OVER 'I' STO 'imax' STO
          'imin' STO                            Update counters.
        END
      END
    >>
    UNTIL exit                                  Quit if exit is true.
    END
  >> MENU                                       Restore original menu.
 >>
>>
```

## 12.7 Displaying Output

A nice intelligible display of a program's results is desirable for the same reasons that motivate input prompting. Furthermore, the methods of producing the text and graphics that show a result are essentially the same as those for producing input displays. The program OLABEL (section 10.2) is a good general purpose utility for output labeling, but you can easily create more elaborate displays using the methods presented in Chapter 10 and the preceding sections of this chapter.

There are a few differences between input and output display methods that are worth noting:

- Output display usually does not require program suspension, so PROMPT is not a good way to display a result. Use DISP and FREEZE to display text that will remain in view after a program finishes.
- You don't need FREEZE to show results while a program is executing. However, you should ensure that any display created while a program is running will persist long enough to be read. Use WAIT in cases where a display might be replaced too

quickly.

- When you want a program-ending display to be available after the next keystroke as provided by FREEZE, create the display on the graph screen instead of the text screen. You can still show the display at the end of a program using PVIEW and FREEZE, but after the picture disappears, you can view it again by pressing [◁] . Using the graph screen also lets you use all of the display (or more, if you create a large graph screen), whereas the menu area of the text screen is not available.

### 12.7.1 Tagged Objects

The *tagged* object type (section 3.4.8) provides a very useful method of output labeling that is especially useful for programs that are intended both as stand-alone programs and as subroutines. For the latter purpose, programs should return their results to the stack where they may be used for subsequent calculations. However, the bare presentation of objects on the stack is not a very helpful style for programs used manually, especially when a program returns two or more objects of the same type. One solution is to tag the output objects: the tags label the objects for visual identification, but do not interfere with the objects' use for further operations.

The command LR is good illustration of using tagged objects for output. Both of LR's results are real numbers; unless you use LR frequently you will be hard pressed to remember which result is which without reference to a manual. Fortunately, you don't have to: the results are returned with the tags Intercept and Slope, clearly distinguishing the two.

The program LCM&GCD listed below demonstrates the creation and use of tagged objects for output. The *least-common-multiple* (LCM) of two numbers is equal to their product divided by their greatest-common-divisor (GCD). LCM&GCD calls the program GCD (section 9.5.2.2), then uses the result to compute the LCM, returning it and the GCD as tagged objects.

| LCM&GCD | *LCM and GCD* | | | | BD00 |
|---|---|---|---|---|---|
| | *level 2* | *level 1* | | *level 2* | *level 1* |
| | *x* | *y* | ☞ | GCD(*x*,*y*) | LCM(*x*,*y*) |

```
« DUP2 * → p                Save the product as p.
  « GCD                     Compute the GCD (section 9.5.2.2).
    "GCD" →TAG              Tag the result.
    p OVER /                Compute the LCM.
    "LCM" →TAG              Tag the result.
    »
»
```

## 12.8 Programs as Arguments

An unusual and powerful feature of the HP48 is its ability to use *procedures* as arguments for commands and other procedures. This capability is clearly illustrated in HP48 symbolic algebra, where algebraic objects can be the arguments for functions. In this section, we will demonstrate the use of *programs* as arguments. The fact that HP48 programs are objects, and that therefore you can put an *unexecuted* program on the stack, means that one program can transfer procedural information to another program as easily as it can transfer data.

The program CHKINPUT presented in section 12.6.1 is a simple example of the use of a program as an argument. Any program that is used to specify a test for CHKINPUT could be included directly in the definition of CHKINPUT, but then the latter program would only be usable for the specific case determined by the test program. By leaving the test as an argument, CHKINPUT can be used as a general utility.

As a more ambitious illustration of the use of programs as arguments, we will develop a program INFSUM to compute the sum

$$\sum_{n=n_0}^{\infty} f(n),$$

where $f(n)$ is an argument for INFSUM, not part of the program. That is, to use INFSUM, you enter $n_0$ and a program representing $f(n)$, as stack arguments.

The following is an example of program development, where you start with a single-purpose program, and expand it in stages to a more general case. The program SUM4 shown below serves as an example of a single-purpose program. It computes the specific sum

$$\sum_{n=1}^{\infty} \frac{1}{n^4}$$

SUM4 accumulates terms until successive sums are equal, i.e. additional terms are less than $10^{-12}$ of the current total. It returns the result 1.08232323295.

| SUM4 | *Sum 1/$n^4$* | CE09 |
|---|---|---|
| | \| *level 1* | |
| | ☞ *sum* | |

| | |
|---|---|
| `<< 0` | Initialize sum. |
| `1` | Starting value of $n$. |
| `DO` | \| *sum(n)* $n$ \| |
| `DUP -4 ^` | \| *sum*$(n)$ $n$ $n^{-4}$ \| |
| `SWAP 1 +` | Increment n. |
| `ROT ROT OVER +` | \| $n+1$ *sum*$(n)$ *sum*$(n+1)$ \| |
| `DUP 4 ROLLD` | \| *sum*$(n+1)$ $n$ *sum*$(n)$ *sum*$(n+1)$ \| |
| `UNTIL ==` | Keep going until *sum*$(n+1)$ = *sum*$(n)$. |
| `END DROP` | Drop $n$. |
| `>>` | |

In reviewing SUM4, you can observe that the sequence $-4$ ^ is the only part of SUM4 that is specific to the particular sum $\Sigma n^{-4}$. The rest of the program just handles the mechanics of adding successive terms and deciding when to stop. You can make the program work for any sum $\Sigma f(n)$ by replacing $-4$ ^ in the fourth line of the program with the name TERM. The variable TERM should contain a program that computes $f(n)$, where $n$ is provided in level 1. The summation program becomes:

| SUMTERM | *Compute an Infinite Sum from TERM* | E3BC |
|---|---|---|
| | \| *level 1* | |
| | ☞ *sum* | |

| | |
|---|---|
| `<< 0` | Initialize sum. |
| `1` | Starting value of $n$. |
| `DO` | \| *sum(n)* $n$ |
| `DUP TERM` | \| *sum*$(n)$ $n$ $f(n)$ \| |
| `SWAP 1 +` | Increment n. |
| `ROT ROT OVER +` | \| $n+1$ *sum*$(n)$ *sum*$(n+1)$ \| |
| `DUP 4 ROLLD` | \| *sum*$(n+1)$ $n$ *sum*$(n)$ *sum*$(n+1)$ \| |
| `UNTIL ==` | Keep going until *sum*$(n+1)$= =*sum*$(n)$. |
| `END DROP` | Drop $n$. |
| `>>` | |

To compute $\Sigma n^{-4}$ with SUMTERM:

```
<< -4 ^ >>  'TERM'  STO  SUMTERM  ☞  1.08232323295.
```

Actually, the use of the variable TERM is an unnecessary contrivance. The need is to supply SUMTERM with the information of how to compute *f(n)*--but that information, which is represented by the program « −4 ^ », can just as well be supplied as a stack argument. To see how, omit the 'TERM' STO from the preceding sequence. Then, at the point where TERM is about to be executed in SUMTERM, the stack looks like this:

```
4:          « -4 ^ »
3:            sum(n)
2:                 n
1:                 n
```

Thus, the effect of executing TERM (evaluating $f(n)$) can be achieved by the sequence 4 PICK EVAL. The program INFSUM (listed on the next page) makes that replacement, and to generalize further, makes the initial index $n_0$ an input argument as well.

■ *Example.* Use INFSUM to compute the sum $\sum_{n=1}^{\infty} \frac{n^2}{2^n}$.

In this case, the program argument is « → n 'n^2/(2^n)' », and $n_0 = 1$. So the sum can be obtained with

```
« → n 'n^2/2^n' » 1 INFSUM ☞ 5.99999999999
```

or

```
« DUP SQ 2 ROT ^ / » 1 INFSUM ☞ 5.99999999999
```

(The second version is faster.) INFSUM may run for a considerable amount of time if the sum converges slowly. For $f(n) = n^{-4}$, it takes 670 terms to compute the result 1.08232323295, which is accurate to the tenth decimal place (the correct value is 1.08232323371). The program will take correspondingly longer for sums that converge more slowly than this. We therefore list a second version, MINFSUM, that you can use instead of INFSUM when you want to monitor the sum as it accumulates.

Additional variations of INFSUM are discussed in section 12.11.4.

| INFSUM | *Compute an Infinite Sum* | | | 6840 |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | « *term* » | $n_0$ | ☞ | *sum* |

| | |
|---|---|
| « 0 | Initialize sum. |
| SWAP | |
| DO | \| *proc.* *sum(n)* *n* \| |
| DUP 4 PICK EVAL | \| *proc.* *sum*(*n*) *n* *f*(*n*) \| |
| SWAP 1 + | Increment n. |
| ROT ROT OVER + | \| *proc.* *n*+1 *sum*(*n*) *sum*(*n*+1) \| |
| DUP 4 ROLLD | \| *proc.* *sum*(*n*+1) *n* *sum*(*n*) *sum*(*n*+1) \| |
| UNTIL == | Keep going until *sum*(*n*+1)==*sum*(*n*). |
| END ROT DROP2 | Discard *n* and *procedure*. |
| » | |

The argument « *term* » must have the logical form « → *n* '*term*(*n*)' ».

| MINFSUM | *Compute an Infinite Sum (Monitor)* | | | BD3B |
|---|---|---|---|---|
| | *level 2* | *level 1* | \| | *level 1* |
| | « *term* » | $n_0$ | ☞ | *sum* |

| | |
|---|---|
| « 0 | Initialize sum. |
| SWAP | |
| DO | \| *proc.* *sum(n)* *n* |
| DUP 4 PICK EVAL | \| *proc.* *sum*(*n*) *n* *f*(*n*) \| |
| SWAP 1 + | Increment n. |
| ROT ROT OVER + | \| *proc.* *n*+1 *sum*(*n*) *sum*(*n*+1) \| |
| DUP 1 DISP | Display the running sum. |
| DUP 4 ROLLD | \| *proc.* *sum*(*n*+1) *n* *sum*(*n*) *sum*(*n*+1) \| |
| UNTIL == | Keep going until *sum*(*n*+1)==*sum*(*n*). |
| END ROT DROP2 | Discard *n* and *procedure*. |
| » | |

The argument « *term* » must have the logical form « → *n* '*term*(*n*)' ».

## 12.9 Timing Execution

Minimizing execution time is an important aspect of program development and optimization. It is straightforward to use the HP 48 system clock to time program execution; the best way is to create a general purpose timer program that takes an object (such as a program) as an argument, executes the object, then returns the execution time. The program TIMED listed below illustrates this method; it returns the execution time of any object, in seconds. The object may either be in level 1 or stored in a variable specified by a name in level 1 (that is, if the level 1 object is a name, it is replaced by the contents of the corresponding variable). TIMED was used to determine the various execution times listed in this book.

| TIMED | *Timed Execution* | | 48A9 |
|---|---|---|---|
| *level 1* | | *level 1* | |
| *object* | ☞ | *time* | |
| *name* | ☞ | *time* | |

| | |
|---|---|
| « IF DUP TYPE 6 == | If the object is a name, |
| THEN RCL | then replace the name with the stored object. |
| END MEM | Pack memory. |
| RCWS 64 STWS → t w | Set maximum wordsize. |
| « TICKS 't' STO | Save the start time. |
| EVAL | Evaluate the object. |
| TICKS t - | Compute the elapsed time in ticks. |
| B→R 8192 / | Convert to decimal seconds. |
| .0129 - w STWS | Correct for local store, restore wordsize. |
| » | |
| » | |

TIMED uses TICKS, which returns the current system time as a binary integer in HP 48 clock "ticks," which are equivalent to 1/8192 second. The correction factor of .0129 seconds at the end of the program compensates for the time used to store the first time value in the local variable t, between the two executions of TICKS. This number may vary slightly from calculator to calculator; you can adjust the value used in your calculator by timing the execution of the object 1. Within the resolution of the system clock, executing 1 takes essentially zero time, so adjust the correction factor if necessary to make 1 TIMED return 0.000.

▪ *Example*. How long does it take the HP 48 to invert a 7 × 7 identity matrix?

```
7 IDN « INV » TIMED ☞ 2.27.
```

The answer is 2.27 seconds.

TIMED executes MEM not to determine available memory, but to force memory packing (see the next section) so that subsequent packing that might interfere with execution timing is postponed as long as possible. The value returned by MEM is only used as a dummy object for the creation of the local variable t.

### 12.9.1 Erratic Execution

You have probably noticed that HP 48 execution, in everything from keystroke entry to user program execution, does not always proceed smoothly but is frequently interrupted by momentary pauses. This is quite noticeable in plotting, for example, where the orderly plotting of points is broken by periodic pauses as if the calculator were "catching its breath." This erratic execution is normal behavior for the HP 48, and should not concern you except to keep it in mind when you are timing program execution. Two consecutive identical operations may take quite different times to execute.

During the course of operations, the HP 48 creates dozens or even hundreds of "temporary objects." These are the objects that you put on the stack and which remain unnamed (i.e., not stored). Between the times when the stack display is updated, various operations may also create many temporary objects that you never see. When a temporary object is dropped from the stack, either for use as an argument, or when it is stored in a global or a port variable, or just by DROP, the memory used for the temporary object is not recovered right away. Eventually, memory fills up with temporary objects, and the HP 48 must perform some "memory packing" in order to continue. This packing consists of reviewing all of the temporary objects, discarding those that are no longer needed, then packing together the remaining objects into the minimum amount of memory. It is this memory packing that is taking place during the execution pauses that you observe.

Ordinarily, the execution pauses caused by packing are so short that they have little effect on your use of the calculator. However, there are some circumstances in which the packing can be very time consuming, effectively paralyzing the HP 48 for many seconds or even minutes. For example, if you enter 1000 numbers onto the stack, executing MEM takes about 2.5 seconds (MEM always performs a memory pack). The worst situation, which you should be careful to avoid, involves the creation of large temporary lists, and the extraction of the objects within the lists. After this sequence,

```
1  1000  FOR  x  x  NEXT  1000  →LIST  OBJ→
```

MEM takes about 10 minutes to execute, during which the keyboard does not respond (type-ahead still works, however). You can only interrupt the packing with a system halt (section 5.8), which also clears the stack.

If you find it necessary to work with large lists, you can avoid the delays due to memory packing by storing the lists in global variables before you take them apart. A similar warning applies to stack programs that enter a large number of objects onto the stack during their execution.

## 12.10 Recursive Programming

The unlimited depth of the HP 48 subroutine return stack provides that programs can not only call other programs without limit, but they can even call themselves any number of times. This feature permits so-called *recursive programming*, in which a repetitive calculation can be achieved by a compact program that iterates by calling itself.

A classic example of recursion is the calculation of a factorial $n! = n(n-1) \cdots 2{\cdot}1$. This definition can be restated in a recursive form:

$$\text{If } n \leq 1 \text{ then } n! = 1; \text{ otherwise } n! = n(n-1)!.$$

The following user-defined function embodies the recursive definition:

```
'FCT(n)=IFTE(n≤1,1,n*FCT(n-1))'  DEFINE
```

The function is defined in terms of itself, so that the name of the variable in which it is stored must match the name used within the defining procedure.

Recursion is not always the fastest or most memory efficient method of computing a result. For the factorial (ignoring the built-in FACT function), a FOR...STEP loop is better than the recursive version:

```
« 1 SWAP OVER FOR n n * -1 STEP ».
```

The looping done by FOR...STEP is faster than a program calling itself, and the program structure also takes care of incrementing *n*. However, in cases involving nested data structures, recursion may provide the only solutions.

The program MINL (section 12.3) finds the minimum in a list of real numbers. Using recursion, it is a simple matter to extend that program so that any element of the input list can itself be a list containing numbers or additional lists, and so on. Here's the revised version:

| RMINL | *Recursive Minimum of a List* | AF1E |
|---|---|---|
| | *level 1* \| *level 1* | |
| | $\{x_1 \cdots x_n\}$ ☞ $x_{\min}$ | |

```
«  MAXR  →NUM  SWAP  DUP  SIZE          | MAXR {xi} n |
  1                                     Initialize m (list index).
  DUP  ROT                              Loop from 1 to n.
  START                                 | xmin {xi} m |
    GETI                                xm
    DUP  TYPE                           Determine the type of object xm.
    IF  5  ==                           Lists are type 5.
    THEN  RMINL                         If it's a list, find its minimum.
    END
    4  ROLL  MIN  3  ROLLD              | xmin {xi} m |
  NEXT
  DROP2
»
```

This program provides another illustration of the power of the unlimited stack. At the point in the program where RMINL calls itself, there is a list in level 1, which is the required argument. It doesn't matter that previous parts of the program have put other objects on the stack--they will still be in the right place when RMINL returns (to the rest of itself). RMINL returns one number to level 1, which is appropriate for the remainder of the program. The initial list can be a list of lists of lists ..., nested indefinitely. For example:

{ 1 { 2 3 } { 4 { 5 { 6 7 8 } 9 0 } { 11 } } 12 }  RMINL ☞ 0.

A classic example of recursive programming is provided by the program SORT, in section 11.5.3. Lists also figure prominently in the recursive system of programs used for computing the determinants of symbolic matrices, described in section 11.7, and in the HP48S program FIND, listed in section 5.7.3. The latter program features a self-recursive program created within a program and stored in a local variable.

A final note on recursive programs. Remember that if you change the name (variable) of a program that calls itself, you have to edit the program to replace all incidences of the old name with the new.

## 12.11 Additional Program Examples

### 12.11.1 Random Number Generators

The HP48 command RAND generates uniformly distributed pseudo-random numbers $x_i$, where an $x_i$ is equally likely to have any value in the range $0 < x < 1$. Using a uniform distribution generator, it is possible to generate random numbers with various other distributions.

#### 12.11.1.1 Poisson Distribution

Assume $x$ is a random variable with a uniform distribution $0 < x < 1$. If $k$ is the smallest integer for which

$$\prod_{n=1}^{k-1} x_n \leq e^{-N}$$

is satisfied, then $k$ is a random variable from a population conforming to the Poisson distribution with mean $N$. This distribution is defined as

$$P(k) = \frac{N^k}{k!} e^{-N},$$

where $P(k)$ is the probability of obtaining $k$ events in an interval where the mean number of events is $N$.

The program POIS uses this algorithm to return one random value $k$, where the mean $N$ is entered as a stack argument.

▪ *Example.* Generate 500 random numbers from a Poisson distribution with mean 10, and compute the mean and standard deviation of the 500 numbers.

▪ *Solution.* Use Σ+ to accumulate the random numbers into ΣDAT, then use MEAN and SDEV.

```
.54321 RDZ CLΣ 1 500 START 10 POIS NEXT { 500 1 } →ARRY Σ+
```

generates the numbers (include the sequence .54321 RDZ if you want to check your results against those shown below). After executing the sequence (which takes several minutes), you can compute the sample statistics:

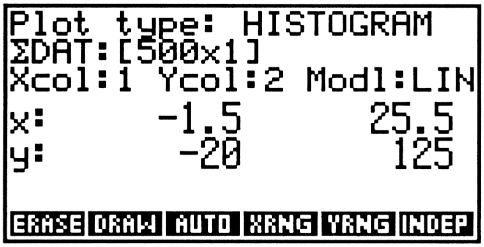
MEAN ☞ 9.994

SDEV ☞ 3.354

The nominal standard deviation of a Poisson distribution is $\sqrt{N}$, which is $\sqrt{10} \simeq$ 3.1623 for $N = 10$.

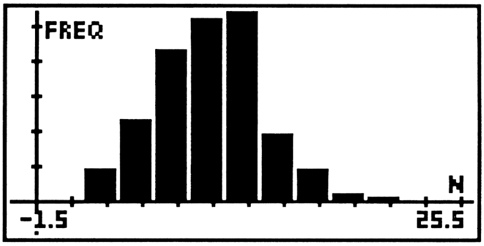
We can use the automatic histogram plotter for a visual inspection of the distribution of the data:

```
[◁] [PLOT]   ≡PTYPE≡   [NXT]   ≡HIST≡
[▷] [PLOT]   -1.5  25.5   ≡XRNG≡
-20  125   ≡YRNG≡   ☞
```



```
Plot type: HISTOGRAM
ΣDAT:[500x1]
Xcol:1 Ycol:2 Modl:LIN
x:      -1.5       25.5
y:       -20        125
```

```
'N'   ≡INDEP≡   'Freq'   [NXT]   ≡DEPN≡
 [◁] [PREV]   ≡ERASE≡   ≡DRAW≡   ≡LABEL≡   [-]   ☞
```



Notice the longer tail on the right, which is characteristic of the Poisson distribution.

| POIS | *Poisson Generator* | 70B5 |
|---|---|---|
| | *level 1* \| *level 1* | |
| | *N* ☞ *k* | |

| | |
|---|---|
| `<< NEG EXP` | exp(−*N*) |
| `-1 1` | Start *k* at −1; the product at 1. |
| `DO SWAP 1 +` | Increment *k*. |
| `SWAP RAND *` | Multiply by the next *x*. |
| `UNTIL DUP 4 PICK ≤` | Keep going until the product is ≤exp(−*N*). |
| `END DROP SWAP DROP` | Return *k*. |
| `>>` | |

### 12.11.1.2 Normal Distribution

Assume $x$ is a random variable with a uniform distribution $0 < x < 1$. With a definition of $y$ as

$$y = \sqrt{-2\ln x_i}\,\cos(2\pi x_j),$$

where $x_i$ and $x_j$ are randomly drawn from the population of $x$, $y$ is a random variable from a population conforming to the normal (Gaussian) distribution with mean 0 and standard deviation 1. The normal distribution for a variable with mean $\bar{y}$ and standard deviation $\sigma$ is

$$P(y) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y-\bar{y})^2}{2\sigma^2}\right),$$

where $P(y)\,dy$ is the probability of obtaining a value in the range between $y$ and $y+dy$. The program NORM computes normally distributed random numbers with zero mean and standard deviation 1.

You can obtain random numbers $y_i'$ from a normal distribution with mean $\bar{y}$ and standard deviation $\sigma$ by multiplying the values $y_i$ obtained with NORM by $\sigma$ and adding $\bar{y}$. The program MNORM returns such random numbers $y_i'$, where the mean and standard deviation are specified on the stack.

| NORM | *Normal Distribution Generator* | 88E7 |
|---|---|---|
| | \| *level 1* | |
| | ☞ $y_i$ | |

| | |
|---|---|
| « RAND | $x_i$ |
| LN -2 * √ | $\sqrt{-2\ln x_i}$ |
| RAND | $x_j$ |
| 2 * π →NUM * RAD COS | $\cos(2\pi x_j)$ |
| * | $y$ |
| » | |

NORM leaves radians mode active.

| MNORM | *Modified Normal Distribution Generator* | 7038 |
|---|---|---|
| | *level 2* *level 1* \| *level 1* | |
| | $\bar{y}$ $\sigma$ ☞ $y'_i$ | |

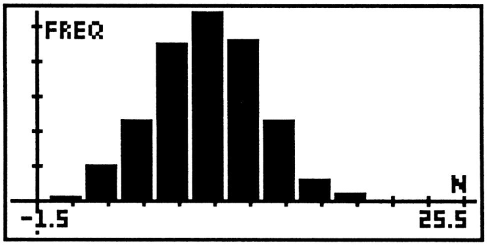
| | |
|---|---|
| « NORM * + | $y'$ |
| » | |

MNORM leaves radians mode active.

▪ *Example*. Generate 500 data points from a normal distribution with mean 10 and standard deviation of 3.16, for comparison with the Poisson data in the previous example.

```
.54321 RDZ 1 500 START 10 3.16 MNORM NEXT
{ 500 1 } →ARRY STOΣ
```

A histogram of this data, using the same plot parameters as in the previous example, looks like this:



Notice that this distribution is more symmetric than the Poisson data.

■ *Example*. Create a ΣDAT matrix that contains points $[x_i, y_i]$ representing a "noisy" straight line:

$$y_i = 0.5x_i + b_i,$$

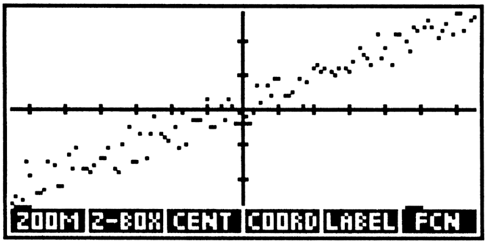
where $b_i$ is a normally distributed random variable with mean 1 and standard deviation 3, and the $x_i$ are the integers $-50$ through $+50$.

■ *Solution*:

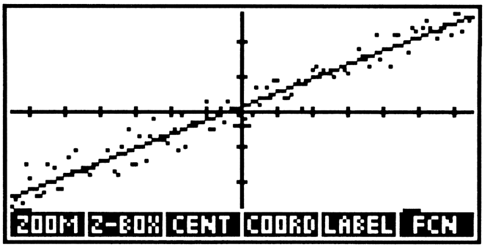
| | |
|---|---|
| .54321 RDZ | Random number seed. |
| CLΣ | Initialize ΣDAT. |
| -50 50 | $x$ from $-50$ to $+50$. |
| FOR x x | $x_i$ |
| 1 3 MNORM x 2 / + | $y_i$. |
| NEXT | |
| { 101 2 } →ARRY STOΣ | Store the data. |

You can create a scatter plot of this data by executing

⬅ STAT NXT NXT SCATR .



Then FCN draws the best-fit straight line:



### 12.11.2 Prime Numbers

The program PRIMES returns a list of the first $n$ prime numbers (not counting 1), where $n$ is specified on the stack. The program demonstrates the use of stack flags (section 9.3) to "remember" the results of tests, so that those results can be used for later decisions.

This program starts with a list of three prime numbers 2, 3, and 5, then successively tests integers $m$ greater than these to see if they are prime by dividing each by all prime numbers $n_i$ for which $n_i \leq \sqrt{m}$. If any quotient is an integer, $m$ is not prime, and is discarded. If $m$ is prime, it is appended to the current list of primes. The process continues until the list grows to the specified size.

You can obtain a significant economy in the execution of this process by observing that you don't need to test every integer explicitly, but only those in the series 7, 11, 13, 17, 19, ..., obtained by alternately adding 2 and 4. All integers not in this series are divisible by 2 or 3, and so are not prime.

The basic structure of PRIMES is as follows:

```
DO
  DO Divide a number by the next prime from the list.
  UNTIL (1) either a quotient is a non-integer.
      or
      (2) the prime is bigger than the number's square root.
  END
UNTIL enough primes are found.
END
```

The combination of tests (1) and (2) is complicated by the fact that there is no point in making test (2) if test (1) is true. In PRIMES, therefore, the (flag) result of test (1) is used twice, once by an IF structure than contains test (2), and again to determine whether to continue through the list of prime number divisors.

| PRIMES | *Find Prime Numbers* | 9F2D |
|---|---|---|
| | *level 1* \| *level 1* | |
| | *n* ☞ { *primes* } | |

```
<< 7 → x                              x is the next candidate number; start with 7.
  << { 2 3 5 }                        First three primes.
    1 SF                              Flag 1 determines increment size.
    DO                                Main loop to test x.
      2 SF                            Flag 2 set means x may be prime.
      3                               Start with n=3 (3rd number in the list).
      DO                              Inner loop--divide x by primes ≤ √x
      GETI x OVER /                   | {primes} n  p_n  x/p_n |
      UNTIL                           Keep going until FP(x/p_n) = 0 or p_n>x/p_n
        IF SWAP OVER >                | {primes} n  x/p_n  flag |
          DUP                         If the test is true...
        THEN SWAP DROP                ...then return true to stop the loop.
        ELSE                          Otherwise, check if evenly divisible.
          SWAP
          IF FP NOT                   If the fractional part is zero...
          THEN 2 CF NOT
          END
        END
      END DROP
      IF 2 FS?                        If x is prime...
      THEN x 1 →LIST +                ... add it to the list.
      END
      IF 1 FS?C                       If flag 1 is set...
      THEN 4                          ...then add 4;
      ELSE 2 1 SF                     ...else add 2 and set the flag.
      END
      x + 'x' STO                     Increment x.
    UNTIL                             Repeat until list is the desired length.
      DUP2 SIZE ≤
    END SWAP DROP                     Leave the list on the stack.
  >>
>>
```

PRIMES uses flags 1 and 2.

## 12.11.3 Simultaneous Equations

Consider the set of simultaneous linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= c_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= c_n, \end{aligned}$$

where there are $n$ equations in $n$ unknowns $x_1 \cdots x_n$. The $a_{ij}$ are the coefficients of the unknowns, and the $c_i$ are the constant terms.

These equations are straightforward to solve on the HP 48. Defining the coefficient matrix

$$\mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix},$$

and the unknown and constant vectors

$$\vec{x} = \begin{vmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{vmatrix} \qquad \vec{c} = \begin{vmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{vmatrix},$$

then the set of simultaneous equations can be represented as the matrix equation

$$\mathbf{A}\vec{x} = \vec{c}.$$

The solution can be found by premultiplying both sides of the equation by the inverse of **A**:

$$\vec{x} = \mathbf{A}^{-1}\vec{c}.$$

On the HP 48, you can obtain this solution by entering the constant vector $\vec{c}$ into level 2 and the coefficient matrix **A** into level 1, then executing / (divide). This returns the unknown vector $\vec{x}$ to level 1.

This method is very simple, but has the drawback that it requires *you* to determine the coefficients and constants from the equations, and enter them in a very specific order, which is contrary to the spirit of the HP48. A better approach is demonstrated by the program SIMEQ below, which does all of this work for you. SIMEQ expects to find a list of names in level 1, preceded in higher levels by as many equations as there are names in the list. The specified names indicate which of the variable names in the equations are the unknown variables--all other variables that appear in the equations must have numerical values (via →NUM). The equations may appear in any order, and there are no restrictions on the form of the equations, except that they must be linear in the unknown variables.

SIMEQ determines the constant terms in the equations by setting all of the unknowns to zero, then evaluating the equations. It next subtracts the constants from the equations, and determines the coefficients by assigning the value 1 to one unknown variable at a time, and evaluating the equations. The coefficients are combined into a matrix, and the constants into a vector so that the vector of unknowns can be obtained by dividing. Finally, the values of the unknowns are stored in the corresponding variables.

▪ *Example.* Five packages are weighed in pairs, yielding the weights 90, 110, 120, 140, 120, 130, 150, 150, 170, and 180 pounds. What are the weights of the individual packages?

▪ *Solution.* Call the unknown weights $A$, $B$, $C$, $D$, and $E$, where $A$ is the lightest weight package and $E$ is the heaviest. Then the lightest combination is $A$ and $B$, so

$$A + B = 90 \text{ lbs.}$$

The next lightest combination must be $A$ and $C$:

$$A + C = 110 \text{ lbs.}$$

Similarly, the heaviest two combinations are

$$D + E = 180 \text{ lbs,}$$

and

$$C + E = 170 \text{ lbs.}$$

Finally, you can observe that the total weight of all the combinations must be four times the total weight of the packages:

$$4(A + B + C + D + E) = 1360 \text{ lbs.}$$

These are the five equations you need to solve the problem:

```
'A+B=90' [ENTER]

'A+C=110' [ENTER]

'D+E=180' [ENTER]

'C+E=170' [ENTER]

'4*(A+B+C+D+E)=1360' [ENTER]
```

puts the equations on the stack; then

```
{A  B  C  D  E} SIMEQ
```

solves the equations.

```
VAR  A     ☞   40
B          ☞   50
C          ☞   70
D          ☞   80
E          ☞  100
```

| SIMEQ | *Simultaneous Equations* | 4AD3 |
|---|---|---|
| *level n ... level 2* | *level 1* | |
| *'equation$_1$' ... 'equation$_n$'* | *{ name$_1$ ... name$_n$ }* | ☞ |

```
«  DUP  SIZE  →  v  n                 Save the list of names in v, and the number
                                      of names in n.
  «  n  →LIST  →  e                   Combine the equations into a list, and save
                                      in e.
    «  1  n
      FOR  x  0  v  x  GET  STO       Store zero in each unknown variable.
      NEXT
      e  OBJ→                         Put the equations on the stack.
      1  SWAP
      START  n  ROLL  →NUM  NEG       Compute each constant term.
      NEXT
      n  →LIST  →  c                  Combine the constants into a list, and save
                                      in c.
      «  1  n                         For each variable...
        FOR  x  1  v  x  GET  STO       Assign the value 1 to the variable.
          e  OBJ→                       Put the equations on the stack.
          1  SWAP                       For each equation...
          FOR  i  n  ROLL  →NUM           Evaluate the equation, and
            c  i  GET  +                  subtract the constant term,
                                          leaving the coefficient.
          NEXT
          0  v  x  GET  STO           Reset the variable to 0.
        NEXT
        n  DUP  2  →LIST
        →ARRY  TRN                    Combine all the coefficients into a square
                                      matrix.
        c  OBJ→  1  →LIST  →ARRY      Convert the constant list in a vector.
        SWAP  /                       Compute the unknown vector.
        OBJ→  DROP                    Put the values on the stack.
        n  1
        FOR  m  v  m  GET  STO        Store each value in its variable.
        -1 STEP
      »
    »
  »
»
```

### 12.11.4 Infinite Sums

In section 12.8 we presented a program INFSUM that computes an infinite sum of terms defined by a separate program. For some sums, it is more accurate to compute each term $T_n$ from the previous one $T_{n-1}$, rather than computing each term independently. The programs PTINFSUM and XPTINFSUM (listed in section 12.11.4.3) use this approach. The first program PTINFSUM is a variation of INFSUM, for which you supply a stack program that computes $T_n$ as a function of $n$ and $T_{n-1}$. PTINFSUM also requires you to specify the initial value $n_0$ of the index, and the value of the first term $T_{n_0}$.

■ *Example*. Compute $\sum_{n=1}^{\infty} \frac{n^3}{2^n}$.

■ *Solution*: In this case, $T_n = \frac{1}{2}\left(\frac{n}{(n-1)}\right)^3$, $n_0 = 1$, and $T_1 = 0.5$. Thus,

```
<< DUP  1  -  /  3  ^  2  /  * >>  .5  1  PTINFSUM ☞ 25.9999999997.
```

Many mathematical functions can be computed from an infinite sum for which the terms are functions of a variable as well as of the summation index. The program XPTINFSUM is a further variation of PTINFSUM, in which the value of a variable is also an input argument, in addition to the arguments required by PTINFSUM. The program that computes $T_n$ from $T_{n-1}$ and $n$ can also be a function of the variable.

The programs SI and CI in the next sections illustrate the use of XPTINFSUM to compute sine and cosine integrals, respectively. The series expansions for these integrals are taken from M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions* (National Bureau of Standards, 1964).

### 12.11.4.1 Sine Integral

The *sine integral* $Si(x)$ is defined as follows:

$$Si(x) = \int_0^x \frac{\sin t}{t} dt$$

The integral can be computed from the infinite series:

$$Si(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)(2n+1)!}$$

for $x > 0$, and $Si(x) = -Si(-x)$ for $x < 0$.

The program SI uses XPTINFSUM to compute this sum, with the assignments $n_0 = 0$, $T_0 = x$, and

$$T_n = T_{n-1}\left(-\frac{(n-½)x^2}{4n(n+.5)^2}\right)$$

Since $T_n$ is a function of $x^2$, SI saves repeated computation of the square of $x$ by using $x^2$ rather than $x$ as the variable argument for XPTINFSUM.

Examples:

```
.5  SI  ☞  .493107418043

3  SI  ☞  1.848652528
```

You could obtain these same results using the HP 48's numerical integration capability, such as with the following alternate form of SI:

```
« → x '∫(0,x,SIN(t)/t,t)' →NUM ».
```

This program is obviously easier to write than the previous version. However, the program using the infinite sum is considerably faster than that using ∫.

### 12.11.4.2 Cosine Integral

The *cosine integral* $Ci(x)$ is defined by

$$Ci(x) = \gamma + \ln x + \int_0^x \frac{\cos t - 1}{t}dt,$$

where $\gamma = .5772156449$ (Euler's constant). $Ci(x)$ can be calculated from the infinite series

$$Ci(x) = \gamma + \ln x + \sum_{n=1}^{\infty} \frac{-1^n x^{2n}}{2n(2n)!}$$

for $x > 0$, and

$$Ci(x) = Ci(-x) - i\pi \quad \text{for } x < 0.$$

The parameters for XPTINFSUM are $n_0 = 1$, $T_1 = -x^2/4$, and

$$T_n = T_{n-1}\left(-\frac{x^2(n-1)}{2n^2(2n-1)}\right).$$

$T_n$ is a function of $-x^2$, so CI uses $-x^2$ rather than $x$ as the variable argument for XPTINFSUM.

Examples:

```
0.5  CI  ☞  -.177784078808

  3  CI  ☞  .11962978602
```

### 12.11.4.3 Sum Programs

| PTINFSUM | *Infinite Sum from Previous Term* | | | | 2DFB |
|---|---|---|---|---|---|
| *level 3* | *level 2* | *level 1* | \| | *level 1* | |
| ≪ *term* ≫ | $T_{n_0}$ | $n_0$ | ☞ | *sum* | |

| | |
|---|---|
| ≪ ROT → term | Save ≪ *term* ≫. |
|   ≪ OVER SWAP | \| $T_{n_0}$ $T_{n_0}$ $n$ \| |
|     DO | \| $sum(n)$ $T_n$ $n$ \| |
|       1 + | Increment $n$. |
|       SWAP OVER term →NUM | \| $sum(n-1)$ $n$ $T_n$ \| |
|       SWAP ROT 3 PICK OVER + | \| $n$ $T_n$ $sum(n-1)$ $sum(n)$ \| |
|       DUP 5 ROLLD | \| $sum(n)$ $T_n$ $n$ $sum(n-1)$ $sum(n)$ \| |
|     UNTIL == | Repeat until the sum is unchanged. |
|     END DROP2 | |
|   ≫ | |
| ≫ | |

The argument ≪ *term* ≫ must have the logical form ≪ → $t$ $n$ '$term(t,n)$' ≫.

| XPTINFSUM | *Infinite Sum in x from Previous Term* | | | | | 11CD |
|---|---|---|---|---|---|---|
| *level 4* | *level 3* | *level 2* | *level 1* | \| | *level 1* | |
| ≪ *term* ≫ | $T_{n_0}$ | $n_0$ | $x$ | ☞ | *sum* | |

| | |
|---|---|
| ≪ 4 ROLL → x term | Save ≪ *term* ≫ and $x$. |
|   ≪ OVER SWAP | \| $T_{n_0}$ $T_{n_0}$ $n_0$ \| |
|     DO | \| $sum(n)$ $T$ $n$ \| |
|     1 + SWAP | Increment n. |
|       OVER x | \| $sum(n-1)$ $n$ $T_{n-1}$ $n$ $x$ \| |
|       term →NUM | \| $sum(n-1)$ $n$ $T_n$ \| |
|       SWAP ROT 3 PICK OVER + | \| $T_n$ $n$ $sum(n-1)$ $sum(n)$ \| |
|       DUP 5 ROLLD | \| $sum(n)$ $T_n$ $n$ $sum(n-1)$ $sum(n)$ \| |
|     UNTIL == | Repeat until the sum is unchanged. |
|     END DROP2 | |
|   ≫ | |
| ≫ | |

The argument ≪ *term* ≫ must have the logical form ≪ → $t$ $n$ $x$ '$term(t,n,x)$' ≫.

| SI | *Sine Integral* | A102 |
|---|---|---|
| | *level 1* \| *level 1* | |
| | $x$ ☞ $Si(x)$ | |

```
<<
  IF  DUP                                   If x=0, just return 0.
  THEN  DUP  ABS  0  OVER  SQ               | x  T_0  n_0  x^2 |
    <<                                      Start of << term >>.
      SWAP  →  n
      <<  n  .5  -  *  NEG  4  /
        n  .5  +  SQ  n  *  /  *
      >>
    >>                                      End of << term >>.
    4  ROLLD                                | x  << term >>  x  T_0  n_0  x^2 |
    XPTINFSUM                               | x  sum |
    SWAP  SIGN  *                           | Si(x) |
  END
>>
```

| CI | *Cosine Integral* | F17C |
|---|---|---|
| | *level 1* \| *level 1* | |
| | $x$ ☞ $Ci(x)$ | |

```
<<  DUP  ABS  DUP  LN
  SWAP  SQ  NEG                             | x  ln|x|  -x^2 |
  DUP  4  /  SWAP  1  SWAP                  | x  ln|x|  -x^2/4  1  -x^2 |
  <<  SWAP  →  n                            Start of << term >>.
    <<  2  /  n  SQ  /  n  1  -  *
      n  2  *  1  -  /  *
    >>
  >>                                        End of << term >>.
  4  ROLLD                                  | x  ln|x|  << term >>  x  T_0  n_0  x^2 |
  XPTINFSUM                                 | x  ln|x|  Σ |
  +  .5772156649  +
  SWAP
  IF  0  <
  THEN  i  π  *  -                          Subtract i π if x<0.
  END                                       Ci (x).
>>
```

# Program Index

| | | |
|---|---|---|
| ADDV | Concatenate Vectors | 300 |
| AGXOR | Animate with GXOR | 264 |
| APLY1 | Apply Program to 1 Symbolic Array | 304 |
| APLY2 | Apply Program to 2 Symbolic Arrays | 304 |
| APVIEW | Animation with PVIEW | 268 |
| AREPL | Animation with REPL | 268 |
| ASN41 | ASN HP41-style | 182 |
| ASTO | Animation with STO | 268 |
| BINCALC | Binary Integer Calculator | 192 |
| BOUNCE | Bouncing Ball Demo | 269 |
| BS? | Bit Set? | 178 |
| CB | Clear Bit | 178 |
| CEQN | Characteristic Equation | 310 |
| CHARDISP | Display HP48 Characters | 258 |
| CHKINPUT | Prompt and Check Input | 331 |
| CI | Cosine Integral | 369 |
| CINT | Circle in a Triangle | 248 |
| COUNT4 | Count in 4 Ranges | 229 |
| CROSSF | CROSS Function | 289 |
| DATENAME | Create a Name from the Current Date | 107 |
| DELROW | Delete a Matrix Row | 280 |
| DFACT | Double Factorial | 234 |
| DIM | Symbolic Array Dimensions | 302 |
| DOTF | DOT Function | 288 |
| DRAWPIX | DRAW using PIXON | 271 |
| FIB | Fibonacci Series Generator | 296 |
| FIND | Find a Variable | 121 |
| FRACALC | Fraction Calculator | 194 |
| FRAME | Frame the Graph Screen | 273 |
| GCD | Greatest Common Divisor | 238 |
| GSAMP | Graphics Samples | 261 |
| GSORT | General-purpose Sort | 298 |
| INFSUM | Compute an Infinite Sum | 349 |
| KEEP | Keep *n* Objects | 78 |
| KEYHALT | Halt if a Key is Pressed | 340 |
| KEYTIME | Wait a Specific Time for a Key | 340 |
| LCM&GCD | LCM and GCD | 345 |
| MINFSUM | Compute an Infinite Sum (Monitor) | 349 |
| MINISTK | Small-font Stack Display | 266 |
| MINL | Minimum of a List | 322 |
| MINOR | Minor of a Determinant | 280 |
| MNDROP | DROP *m* through *n* | 78 |
| MNORM | Modified Normal Distribution Generator | 357 |
| MOVE | Move a Variable | 122 |
| MSGSHOW | Show Messages | 343 |
| NORM | Normal Distribution Generator | 357 |

| | | |
|---|---|---|
| N→S | Numeric to Symbolic | 303 |
| OLABEL | Object Labeling Utility | 259 |
| POIS | Poisson Generator | 356 |
| PRIMES | Find Prime Numbers | 360 |
| PROMPTCONT | Prompt with CONT Display | 330 |
| PTINFSUM | Infinite Sum from Previous Term | 368 |
| QU | Quadratic Root Finder | 319, 326 |
| RC→R | Real/Complex to Real | 227 |
| RMINL | Recursive Minimum of a List | 353 |
| SADD | Add Symbolic Arrays | 305 |
| SA→ | Symbolic Array to Stack | 302 |
| SB | Set Bit | 177 |
| SCOF | (Unsigned) Symbolic Cofactor | 308 |
| SDET | Symbolic Determinant of a Matrix | 308 |
| SI | Sine Integral | 369 |
| SIMEQ | Simultaneous Equations | 364 |
| SKETCH | Sketch Lines | 273 |
| SMINOR | Minor of a Symbolic Matrix | 309 |
| SMS | Scalar Multiply Symbolic Arrays | 306 |
| SMUL | Multiply Symbolic Arrays | 306 |
| SORT | Sort a List in Increasing Order | 297 |
| SSUB | Subtract Symbolic Arrays | 305 |
| STAR | Draw a Star | 272 |
| STRN | Transpose Symbolic Array | 305 |
| SUBCOL | Subtract Columns | 282 |
| SUM4 | Sum $1/x^4$ | 347 |
| SUMTERM | Compute an Infinite Sum from TERM | 347 |
| S→N | Symbolic to Numeric | 303 |
| TIMED | Timed Execution | 350 |
| TPIX | Toggle a Pixel | 272 |
| VANGLE | Angle Between Two Vectors | 283 |
| VSUM | Sum Vector Elements | 234 |
| XARCHIVE | Extended Archive | 108 |
| XFORM | Coordinate Transformation | 289 |
| XPTINFSUM | Infinite Sum in x from Previous Term | 368 |
| →SA | Stack to Symbolic Array | 302 |

# Subject Index

π 56, 211
→ 206, 210, 246, 290
+ 262
= 224
== 223, 224
< 223
≤ 223
> 223
≥ 223
≠ 223
☞ 9
√ 165
∡ 24, 285
' ' 62
«» 213, 214
{ } 40
| 160, 168
[ ] 40
" " 36, 62
# 41
_ 44
: : 43
@ 36, 146
1USR annunciator 180
24-hour format 179
2D 35, 287
3D 287
aborting programs 316
ABS 283
acknowledged alarms 179
action 30, 32
  user key 179
activation 30
alarm beep 179
  catalog 135
ALG annunciator 141
algebraic 48
  calculator 21
  entry mode 141, 143, 207
  evaluation 49
  object 3, 21, 25, 29, 30, 32, 47, 63, 200
  syntax 16, 48
algebraic/program mode 142
alpha key action 179
analytic function 26
angle mode 175, 178, 286
annunciator, 1USR 180
  ALG 141
  busy 145
  PRG 141
  user 180
ARC 273, 274
ARCHIVE 106
argument 16
  disappearing 79
  recovery 72, 87, 179, 242, 312
  saving 87
array 40, 277, 301
  entry 151
→ARRY 72, 277, 288
ARRY→ 278
ASCII files 215
ASN 180, 182, 184
assignment, key 175, 180
ATTACH 102, 104
attach library 101
ATTN 133, 241, 315, 339
automatic linefeed 179
  mode change 142
  simplification 52, 57
automating calculations 195
backspace 168
backup object 46
Bad Argument Type 49
base 165
BASIC 2, 32, 201, 245
beep, error 179
BIN 178
binary integer 40
binary transfer 179
BLANK 262
BOX 273, 274
branch 225
  unconditional 225
built-in object 59
  program object 31
busy annunciator 145
BYTES 10, 41, 224, 318, 327
CASE structure 228
cell 150
  cursor 150
CENTR 138
CF 55, 176, 222
changing variable contents 117
character code 38, 39
characteristic equation 310

checksum 327
CHR 39
Circular Reference 87
CLEAR 68, 135
clear flag 175
clearing 68
clipping 275
CLLCD 257, 258
closing subexpression 160
CLUSR 96
CLVAR 96, 142
code object 30
cofactor 307
column number 116
column vector 282
combining RPN and algebraic 22
command 26, 59, 100
  test 221, 223
command line 63, 139, 140, 144, 146, 202, 335
command stack 145
comment 146, 36, 216
common notation 17
compact format 9
complex array, MatrixWriter 151
complex number 34
complex result 35
composite object 63, 289, 51, 299
CON 117, 120, 279, 294
concatenation 37
conditional 32, 221, 225
configuration program, library 103
constant, symbolic 57, 170, 178, 211
CONT 183, 290, 314, 316, 317, 329, 333
contravariant vector 282
coordinate mode 285
coordinate system 178
coordinates, logical 269
  cylindrical polar 284-286
  polar 34, 284-286
  rectangular 284
  spherical polar 284-286
copying stack objects 70
counted string 37
counter 119
covariant vector 282
cylindrical polar coordinates 284-286
CRDIR 89
CROSS 283, 288
CST 185
current directory 89, 111
current path 89, 101
cursor, cell 150
  graphic 179
  subexpression 170
curve filling 179
custom error 242
custom menu 185, 186, 175
customization 175
cylindrical polar coordinates 284, 286
C→PX 270, 274
C→R 35
data object 32
date format 179
DBUG 317, 319
debugging 317
DEC 178
decimal digits 179
DECR 117, 119
DEFINE 86, 205, 208, 210
defining expression 206
  procedure 11, 246
definite loop 230, 234
definition, object 27
deleting suspended program 316
delimiter 27, 127, 144, 214
←DEL 147
DEL→ 147
DELKEYS 183
denominator 163
DEPTH 293
DETACH 104
determinant 307
digit-group commas 34
directory 45, 60, 88
  current 89, 111
  PURGE 94
Directory Not Allowed 94, 95
Directory Recursion 95
disappearing argument 79
DISP 258, 332, 344
display 255
  freeze 258
  graphics 123, 124, 260
  standard 126, 256
divide bar 163
DO 238
DO loop 236
DOERR 100, 104, 241, 242
DOT 282, 288
double quote 36, 62
double-space mode 179
DRAW 53, 249, 271
DROP 68. 134
DROP2 69

DROPN 69
DUP 70, 82, 134
DUPN 72
ECHO 148
EDIT 76, 147, 154, 311
edit menu 147, 154
edit/view 148
editing program 311
ELSE 225
else-sequence 225
empty 79
END 225
endless execution 87
endless loop 61
ENTER 19, 26, 72, 134, 140
  explicit 140
  implicit 140, 145
αENTER 191
βENTER 191, 265
ENTRY 143
entry mode 141, 142, 187, 189
  algebraic 141, 143, 207
entry, array 151
entry, text 128
environment 12, 126, 132
  plot 12, 126
  standard 12
EQ 118
equality 223
  logical 223
  physical 223
EquationWriter 134, 148, 156, 266
ERR0 241
ERRM 100, 104, 240, 241, 242, 316
ERRN 240, 241, 242
error 239
  beep 179
  trap 239, 242
  custom 242
error-sequence 240
EVAL 31, 47, 51, 55, 60, 64, 97, 114, 290
evaluate 16, 25, 31
  algebraic 49
exception 243
  action flag 243
exchange of arguments 69
executed 31
execution 25, 30, 31, 60, 116
  by address 64
  endless 87
  local name 61
  numerical 52, 55, 56, 58
  postponed 146
  preventing 31
  symbolic 52, 58, 225
exit 234, 225
EXPAN 50
explicit ENTER 140
exponent 33
exponentiated 165
EXPR 170
expression 16, 48
  defining 206
fast catalog 179
FC? 223
FC?C 177, 223
flag 55, 175, 220, 333
  −2 56
  −3 56
  −15 285
  −19 36
  −20-−26 243-244
  −51 34
  −55 73, 243
  −60 129
  −61 129, 180
  −62 180
  clear 175
  exception action 243
  stack 177
  system 175
  user 176, 222
floating-point 33
font 265
FOR 230
FOR...NEXT 246, 230
FOR...STEP 233, 246
formal variable 61, 249
  24-hour 179
  compact 9
  date 179
  decimal number 179
  linear 156
FORTH 5, 67
fraction mark 179
FREE 105
FREEZE 257, 329, 332, 344
freeze display 258
FS 222
FS? 223
FS?C 177, 223
function 16, 25, 26, 48, 205
  menu 132
  analytic 26

user-defined 86, 161, 196, 204, 205, 210, 288
GCD 345
generations, calculator 2
GET 116, 117, 186, 278, 281, 291, 295, 299
GETI 116, 117, 278, 291
global name 30, 59, 60, 61, 83, 90, 245, 327
variable 45, 59, 83, 111, 245, 248, 329
GOR 263, 267, 270
GOTO 217
GO→ 151
GO↓ 151
GRAPH 126, 134, 256, 257
graph screen 126, 255, 260, 266
graphics 256
cursor 179
display 123, 124, 260
display of expression 156
object 42, 260
greatest common divisor 345
GROB 42
→GROB 265
*guillemets* 62
GXOR 263, 267, 270
HALT 109, 290, 313, 315, 316, 317, 320, 329, 332, 335
helvetica 8
HEX 178
hidden operation 133
parentheses 179
HOME 91, 114, 121
home directory 88
HP Solve 53, 138, 196, 201
HP 17B 13
HP 19B 13
HP 27S 13
HP 35 2
HP 41 2-5, 76, 140, 180, 181, 202, 216, 217
HP 65 2
HP 71B 180
i 35
IDN 117, 120, 279
IF 225
IF structure 135, 225
IFERR structure 317, 339
IFT 32, 51, 227
IFTE 32, 51, 227
IM 36
immediate entry mode 141, 143
immediate-execute key 141
implicit ENTER 140, 145
GET 117
parentheses 165
implied multiplication 162
Improper Definition 209
Incomplete Subexpression 161
INCR 117, 119, 237
indefinite loop 230, 234, 235
independent RAM 99
index for GET 116
index wrap 179
infinite result 178
infinite result action flag 243
infix notation 17
operator 162
inner product 283
INPUT 335, 341
input and output 329
input list 294
insert mode 148
interactive stack 74, 134, 148
intermediate result list 296
intermix binary and real 41
real and complex 36
internal accuracy 33
Invalid Array Element 151
Invalid Card Data 105
Invalid Syntax 161
Invalid User Function 210
IOPAR 118
IR port 179
ISOL 207, 249
italics 8
iteration 225, 230
KEEP 76
Kermit message 179
Kermit overwrite 179
KEY 335, 339, 341
key assignment 175
buffer 339
code 341
format 8
menu 8
plane 180
shifted 8
type 141
typing 141, 142, 190
key-per-function 125
keyboard 127
standard 126
keycode 180, 339
KILL 316
label 217
LAST 72
last arguments 72, 87, 179, 242, 312
command 176

error message 123, 124
error number 123, 124
menu 129
stack 72, 144, 176
LASTARG 72, 87, 242
→LCD 257, 260
LCM 345
least-common-multiple 345
LET 86
LEVEL 76
library 30, 96, 99, 100, 123, 124
attach 101, 102, 104
ID 100
title 102
LIBRARY menu 46, 96, 105, 137
LIBS 104
LINE 272, 274
linear format 156
LISP 5
list 40, 277, 289
object 25, 30, 32, 51, 289
input 294
output 296
→LIST 290, 293
LIST→ 291
local memory 109, 110, 123, 124, 245, 248
name 30, 109, 111, 206, 208, 245
name execution 61
name resolution 249
variable 79, 59, 61, 74, 109, 205, 206, 231, 245, 248, 315
variable structure 111, 246
logical coordinates 269
equality 223
operator 221
loop 230
index 231, 246
definite 230, 234
DO 236
endless 61
indefinite 230, 234, 235
WHILE 236-238
loop-sequence 236, 237
LR 345
Łukasiweicz, Jan 17
GXOR 263, 267, 270mantissa 33
manual operation 26
mathematical function 207
matrix 40, 277
MatrixWriter 134, 148, 149, 176
MEM 96, 105, 184
Memory Clear 124, 125
memory reset 123, 124
memory, local 109, 110, 123, 124, 245, 248
user 59, 88
VAR 59
MENU 130, 185, 333
menu, custom 185, 175
exit 132
key label 126
port 99
screen 255
subexpression 170
VAR 83, 112, 137, 185, 245
MERGE 105
message, table 242
message, prompt 179
minor 307
mode 12, 175
coordinate 285
entry 141, 142, 187, 189
insert 148
numeric 225
program entry 76, 143, 214
server 138
user 123, 124, 129, 158, 180
mode-dependent key 141
name, global 30, 59, 60, 61, 83, 90, 245, 327
local 30, 109, 111, 206, 208, 245
object 29, 58, 116, 233
port 97
quoted 62
resolution 90, 110, 249
XLIB 30, 46, 61, 103, 181
NEG 260, 262
negative pixel coordinates 274
newline 151
NEWOB 100, 297, 300
non-analytic function 26
Non-Empty Directory 94, 96
normal-sequence 239
NOT 238
notation 8
common 17
infix 17
Polish 17
prefix 17
Reverse Polish 15
NUM 39
→NUM 32, 55, 56
numbered register 59
numerator 163
numeric mode 225
numeric/symbolic execution 178
numerical execution 52, 55, 56, 58

object 25, 139
  class 32
  composite 63, 289, 51, 299
  data 32
  definition 27
  entry 127
  graphics 42, 260
  name 29, 58, 116, 233
  program 30
  string 36, 144
  symbolic 63
  system 27
  tagged 42, 345
  type 27
  unit 44
  untagged 43
  value 27
Object In Use 99, 104
object-to-grob conversion 265
OBJ→ 35, 38, 43, 44, 69, 112, 191, 278, 288, 291
OCT 178
operation 25, 59
  manual 26
  logical 221
ORDER 85, 112
output, display 344
  list 296
OVER 71
overflow 178, 244
Owner's Manual 5
page 129
parent 90
parsing 144
PATH 89, 113
path name 113
path, current 89, 101
PDIM 267
pencil-and-paper 18
permanent custom menu 185
PGDIR 96, 106
physical equality 223
PICK 70
PICT 267
picture 255
PIX 274
PIX? 272
PIXOFF 271, 274
PIXON 270, 274
plot environment 12, 126
polar coordinates 34, 284
Polish notation 17
port 96, 97, 99, 100, 105
  menu 99
  name 97
  variable 97, 96
POS 39, 293
postponed execution 146
PPAR 118, 138, 269, 270
PR1 138
precedence 17, 179
prefix notation 17
preventing execution 31
PRG annunciator 141
principal value 178
printer port 179
problem solving 195, 198
procedure 47, 200, 201
  defining 11, 246
program 29, 47, 52, 196, 201, 213
  as argument 331
  body 214, 215
  content 47
  definition 47
  editing 311
  entry mode 76, 141, 143, 201, 214
  legibility 205
  object 30
  optimization 323
  quotes 62
  quoted 63
  structure 47, 202, 213, 215, 219, 326
  structure word 32, 142, 213, 219
  suspended 313, 329
  unquoted 63
programming 195, 202
  structured 203, 216, 217, 313
PROMPT 290, 313, 315, 317, 320, 329, 330, 332, 335, 344
prompt message 179
PRTPAR 118
PRVAR 97, 318
PURGE 86, 92, 96, 97, 99, 100, 267
  directory 94
  recovery 87
PUT 117, 119, 120, 186, 278, 281, 291, 295
PUTI 117, 119, 120, 278, 291
PVARS 96
PVIEW 257, 270, 345
PX→ 270, 274
→Q 193
QUAD 249
quotation mark delimiters 62
quote, name 62
  program 62
  single 62

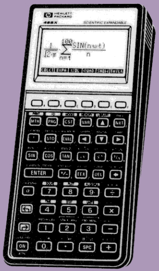
string 62
tagged object 99
RATIO 164
RCEQ 138
RCL 60, 97, 100, 113, 116, 222, 267
RCLALARMS 183
RCLF 177
RCLKEYS 183
RCLMENU 186
RCWS 178
RDM 117, 120, 278
RE 36
real number 33
Recover RAM 124, 125
recovery 314
argument 72, 87, 179, 242, 312
from PURGE or STO 87
stack 72
rectangular coordinates 284
referenced 99
register 115
numbered 59
storage 59
REPEAT 237, 238
REPL 39, 172, 264, 267, 268, 270, 292
Replace RAM, Press ON 105
reschedule 179
resolution, name 110
local name 249
RESTORE 106, 107
result 16
Reverse Polish Notation 15
right hand 9
ROLL 69
ROLLD 69
ROOT 201, 249
ROT 70
row number 116
row order 277
row vector 282
RPL 5, 12, 21, 22, 60, 64, 82, 112
RPN 3, 15
command 26
principle 15, 50
RULES 132, 149, 157, 172
R→C 35
S 183
SAME 44, 223
saving, argument 87
SCONJ 117, 120
screen 126, 255
graph 126, 255, 260, 266
menu 255
text 126, 255, 260
ΣDAT 118
separator 144
sequence 9, 213, 230, 233
server mode 138
set flag 175
SF 55, 176, 222
signal flag 176, 244
simplification, automatic 52, 57
SIN 175
single quote 62
single-step 316, 319
SINV 117, 120
SIZE 38, 262, 267, 278, 291
SKEY 184
←SKIP 147
SKIP→ 147
SNEG 117, 120
solve variables menu 197
ΣPAR 118
space, in EquationWriter 161
spherical polar coordinates 284, 286
square root 165
SST 316, 319
stack 9, 19, 21, 67, 125
diagram 10
flag 177
level 29, 67
recovery 72, 312
roll 69
interactive 74, 134, 148
unlimited 77
display 126, 256
environment 12
keyboard 126
start 230, 231, 233
START...NEXT 234
START...STEP 234
starting and stopping 313
statistics matrix 149
status area 126
step 233
step-wise substitution 60
STEQ 138
STO 44, 83, 87, 92, 97, 100, 267, 268, 281
STO recovery 87
STO– 118
STO+ 118
STO* 118
STO/ 118
STOF 177, 183

STOKEYS 182
stop 230, 231, 233
storage arithmetic 118
storage register 59
→STR 38
string object 36
string, counted 37
string, object 144
stripping tags 44
structure 50
structure word, program 32, 142, 213, 219
structure, local variable 111, 246
structure, program 47, 202, 213, 215, 219, 326
structured programming 203, 216, 217, 313
STR→ 112
STWS 41, 178
SUB 39, 265, 266, 267, 292
subdirectory 90
subexpression 50, 158, 170
subexpression cursor 170
subexpression level 50
subexpression menu 170
subexpression, closing 160
subroutine 217, 330
substitution 64
summation 231
suspended program 313, 329
SWAP 69, 70, 134
symbolic array 301
symbolic calculator 3
symbolic constant 57, 170, 178
symbolic execution 52, 58, 225
symbolic fraction entry 163
symbolic manipulations 196
symbolic math 197
symbolic object 63
syntax 140
syntax, algebraic 16, 48
SYSEVAL 64
system flag 175
system halt 88, 100, 123, 124
system object 27
tag 42
tag, stripping 44
→TAG 43
tagged object 42, 345
tagged object, quoting 99
temporary custom menu 186
test 221
test command 221, 223
test-sequence 225, 236, 237
TEXT 126, 257
text entry 128
text screen 126, 255, 260
THEN 225
then-sequence 225
ticking clock 179
ticks 62
TLINE 272, 274
TMENU 130, 186, 333
Too Few Arguments 210
TRN 117, 120, 279
TVARS 121, 122
TYPE 27, 44, 52, 63
type number 27
type, object 27
type-ahead 339
typing 188
typing key 141, 142, 190
UBASE 224
UFACT 224
unconditional branch 225
Undefined Name 53
underflow 178, 244
unit magnitude 44
unit management 44
unit object 44
unlimited stack 77
unquoted program 63
untagged object 43
UNTIL 236
UPDIR 90
user annunciator 180
user flag 176, 222
user key action 179
user key assignment 180
user memory 59, 88
user mode 123, 124, 129, 158, 180
user-defined function 86, 161, 196, 204, 205, 210, 288
→V2 35, 287, 288
→V3 287, 288
value, object 27
VAR memory 59
VAR menu 83, 112, 137, 185, 245
variable 16
variable, formal 61, 249
variable, global 45, 59, 83, 111, 245, 248, 329
variable, local 79, 59, 61, 74, 109, 205, 206, 231, 245, 248, 3
variable, port 97, 96
VARS 121, 122
VEC 154
vector 40, 277, 282
vector, contravariant 282
vector, covariant 282

vectored ENTER 144, 145, 179, 190
VIEW 76
VISIT 76, 147, 311
VTYPE 27, 52, 122
V→ 35, 287
WAIT 341
where 168
WHILE 238
WHILE loop 236-238
WID→ 150
wordsize 41
XLIB name 30, 46, 61, 103, 181
XROOT 160, 166

# HP 48 Insights
## I. Principles and Programming

The HP 48 is the most powerful calculator ever developed. Along with its extensive symbolic and numeric mathematical functionality including automated graphics, the HP 48 provides exceptional programming and customization facilities that make it applicable to a broad range of practical problems. The sheer extent of the HP 48's capabilities do, however, make the calculator a challenge to learn and master.



*HP 48 Insights Part I* is the first volume of a two-part series by Dr. William Wickes on the operation and application of the HP 48. *Part I* concentrates on the underlying unified principles of HP 48 operation, and the tools and techniques for programming the calculator. Special attention is given to object storage, display management, and customization with key assignments, menus, and modes. All concepts are illustrated with specific examples, including over 100 practical example programs featuring programming techniques such as local variables, program structures, recursion, and the uses of lists and arrays.

This book is derived from *HP 28 Insights*, the book that is recognized as the definitive exposition of HP 28 principles and operation. A companion work, *HP 41/HP 48 Transitions*, is written to assist users of the HP 41 family of calculators in adapting their HP 41 knowledge to the HP 48. *Part II* of *HP 48 Insights* will focus on the integrated systems of the HP 48, such as plotting, symbolic mathematics, and unit management.

**Chapter Headings:**

| | | |
|---|---|---|
| 1. | **Introduction** | 1 |
| 2. | **Understanding RPN** | 15 |
| 3. | **Objects and Execution** | 25 |
| 4. | **The HP 48 Stack** | 67 |
| 5. | **Storing Objects** | 83 |
| 6. | **Methods** | 125 |
| 7. | **Customization** | 175 |
| 8. | **Problem Solving** | 195 |
| 9. | **Programming** | 213 |
| 10. | **Display Operations and Graphics** | 255 |
| 11. | **Arrays and Lists** | 277 |
| 12. | **Program Development** | 311 |

ISBN 0-9625258-3-9