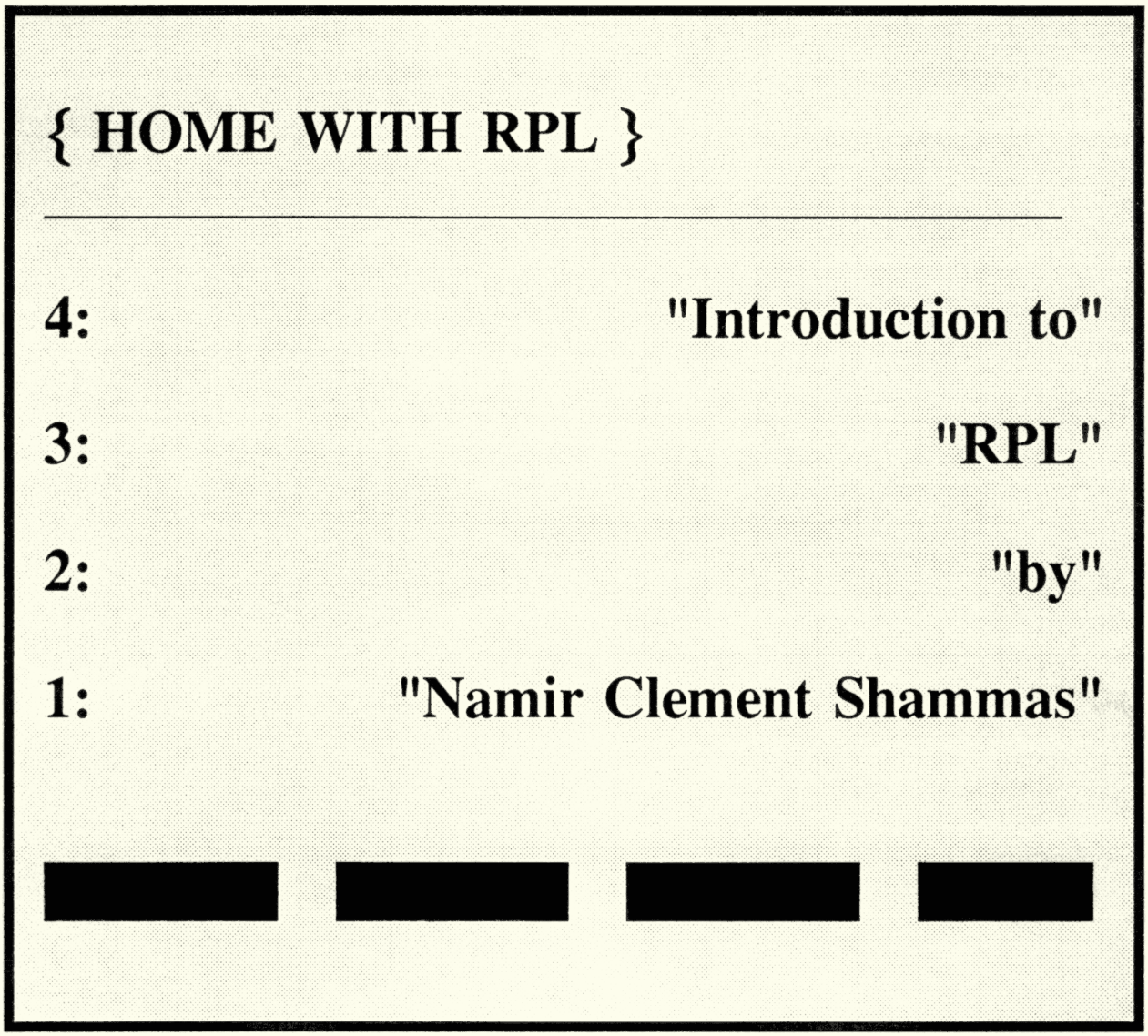
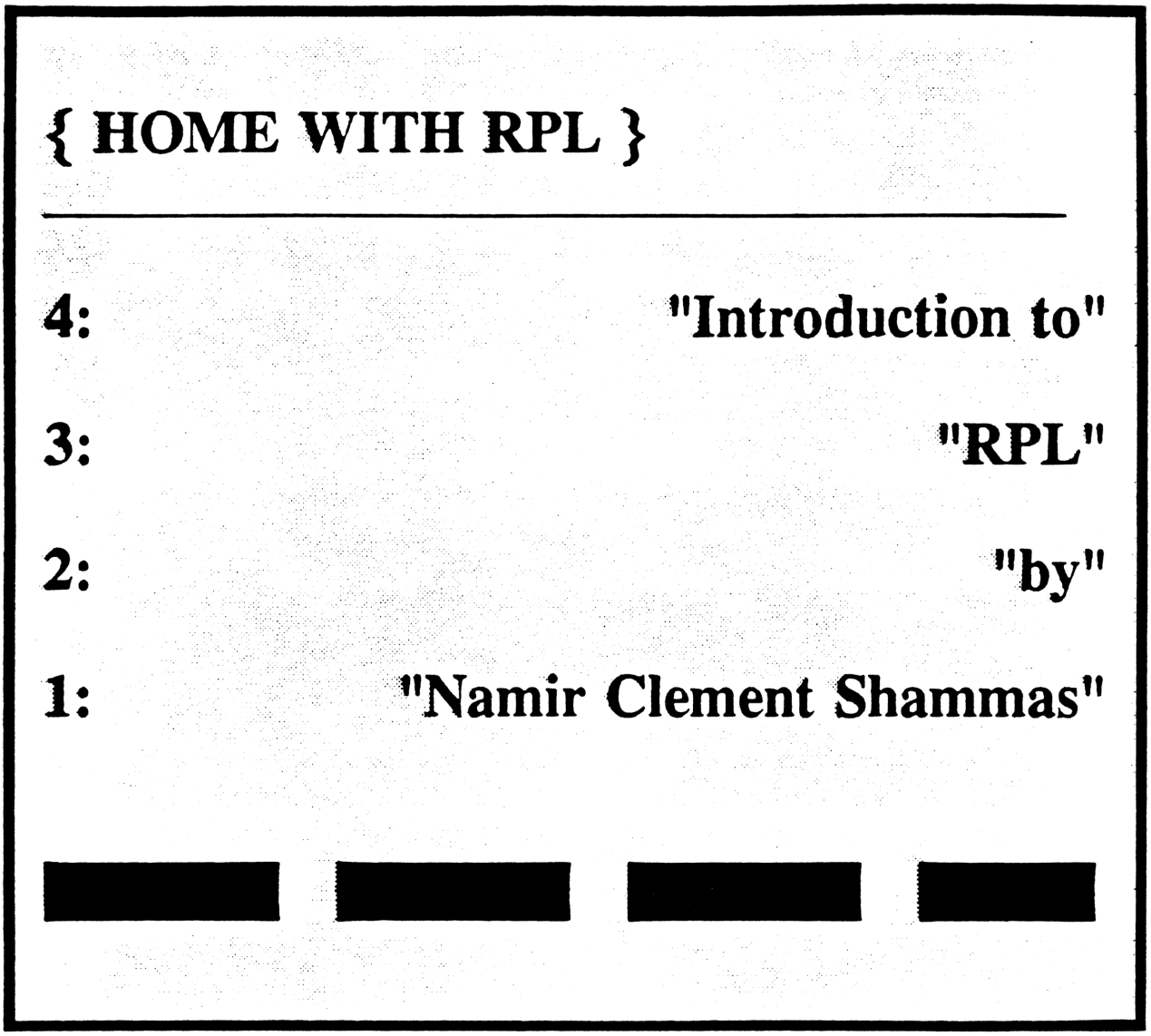

{ HOME WITH RPL }
4: "Introduction to"
3: "RPL"
2: "by"
1: "Namir Clement Shammas"

EduCALC



EduCALC
Laguna Niguel, California

DISCLAIMER

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. IN NO EVENT SHALL EDUCALC, OR THE BOOK AUTHOR (NAMIR CLEMENT SHAMMAS), OR THE DISTRIBUTORS OF EDUCALC BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER OCUENIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THIS EDUCALC PRODUCT, EVEN IF EDUCALC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

HP48SX and HP41C/V are trademarks of the Hewlett-Packard company.
QuickBASIC and GW-BASIC are trademarks of Microsoft Corporation.
BASICA is a tradmark of IBM.

EduCALC
27953 CABOT RD.
LAGUNA NIGUEL, CA 92677

(714) 582-2637

Stock #2454

Dedication

To Richard Nelson

Table of Contents

Chapter 9: Special and Non-Numerical Arrays

Chapter 10: Strings

Table of HP48SX Listings

Introduction

The HP48SX is a fascinating scientific machine. It is a worthy successor to the HP41C. The HP48SX is packed with numerous features that offer a lot of power to the scientists and engineers. This book *introduces* you to the RPL language used by the HP48SX. This book is by no means a rewrite of the HP48SX manuals! Instead, it focuses on the aspects of the RPL language itself. I am assuming that most of the readers are familiar with either the HP41C or BASIC (Microsoft's QuickBASIC, in particular). Consequently, the book contains notes and comments to such readers. My aim is to assist you in becoming more comfortable with programming in RPL.

The first eight chapters present the various aspects of the RPL language. Chapter 9 implements special arrays that can turn your HP48SX into a pocket database that stores your phone numbers and addresses. Chapter 10 presents a string library that most of the readers should find useful.

I would like to thank Richard Nelson for encouraging me to write this book. I would also like to thank the Chicago CHIP chapter for holding the first HP48SX conference in June 1990. They did a superb job in bringing together old PPC and CHHU friends. Finally, I would like to thank Dr. Bill Wickes for leading the team that designed the HP48SX. Keep up the excellent work Bill!

Namir Clement Shammas

January 1991

1

The HP48SX Stack

The HP48SX employs a stack that is a superset of the HP41C and most of the other Hewlett-Packard RPN calculators. This chapter looks at the stack of the HP48SX and compares it with that of the HP41C/CV/CX.

Operational Differences

The differences between the stacks of the HP48SX and those of the HP41C are as follows:

1. The HP41C has a fixed-size stack of four registers (or levels) named X, Y, Z, and T. The HP48SX has a multi-level stack that is only limited by the amount of memory available. You refer to a level by its number. The top of the stack is level 1, the next topmost level is 2, and so on.

2. When the HP41C stack drops due to the execution of a function or math operation, the T register maintains its value while the other registers shift data. No similar mechanism exists in the HP48SX. This is part of the machine's multi-level stack feature. This difference is worth noting if you intend to use some HP41C programming tricks with the new machine.

3. The stack of the HP41C essentially is dedicated to storing floating-point numbers. By contrast, the stack of the HP48SX is able to store a variety of objects (or data types, if you like). The partial list of objects includes floating-point numbers, complex numbers, strings, lists, arrays, matrices, and graphics objects. Thus, the HP48SX is a heterogeneous stack, while that of the HP41C is a homogenous one.

4. The concept of the HP41C *LastX* register has been expanded in the HP48SX to include the last argument, menu, command, and stack contents.

5. The HP48SX has a greater reliance on the stack to supply arguments. For example, to store a value from the stack into a variable, you need to push the name of the variable into the stack before issuing a STO command. This is quite different from issuing a STO 00, for example, on an HP41C to store a number in the register 00. The HP41C works by allowing you to specify some of the arguments as part of the command. Not so with the HP48SX!

Stack Manipulating Commands

The commands that manipulate the HP48SX stack are very similar to those of the stack-oriented FORTH language (RPL's cousin). These commands are:

1. **CLEAR**. This command clears the entire stack of the HP48SX. It is similar to the CLST

command on the HP41C. The difference lies in the fact that in the HP41C the stack registers are filled with zeroes. On the HP48SX all of the objects in the stack are dropped (or popped off, if you like).

2. **DEPTH**. This command returns the number of objects in the stack, regardless of their types. The result is pushed into the stack. This command has no equivalent in the HP41C, since a four-stack register is always at work.

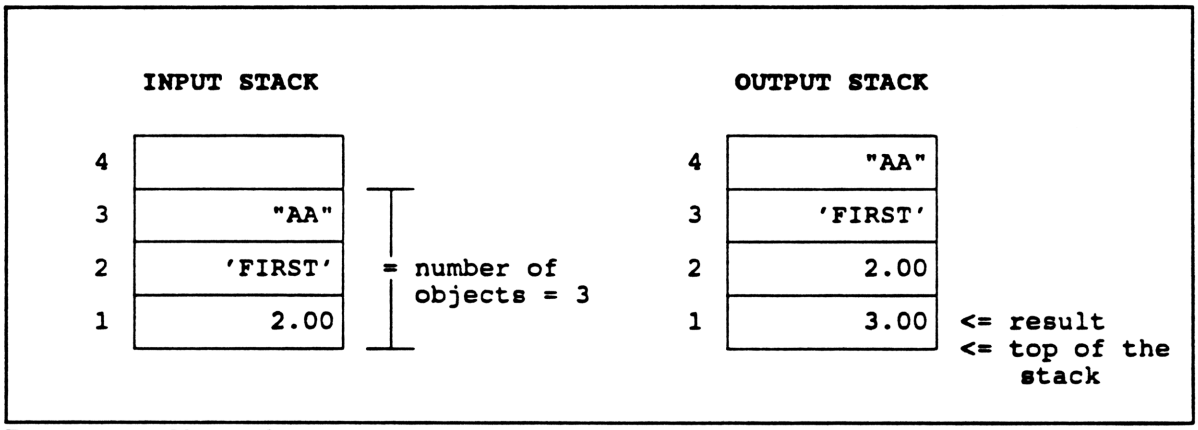


Figure 1.1. Using the DEPTH command.

3. **DROP**. This command drops (or pops off, if you prefer) the object in level 1.

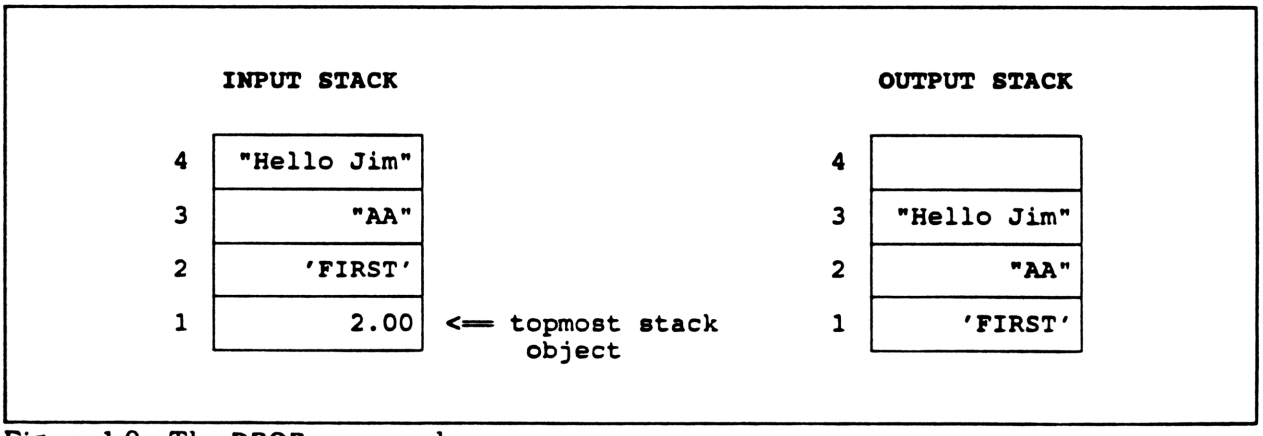


Figure 1.2. The DROP command.

4. **DROP2**. This command causes the objects in levels 1 and 2 (that is, the two topmost objects in the stack) to be popped off the stack. The DROP2 command is a shorthand for issuing two DROP commands.

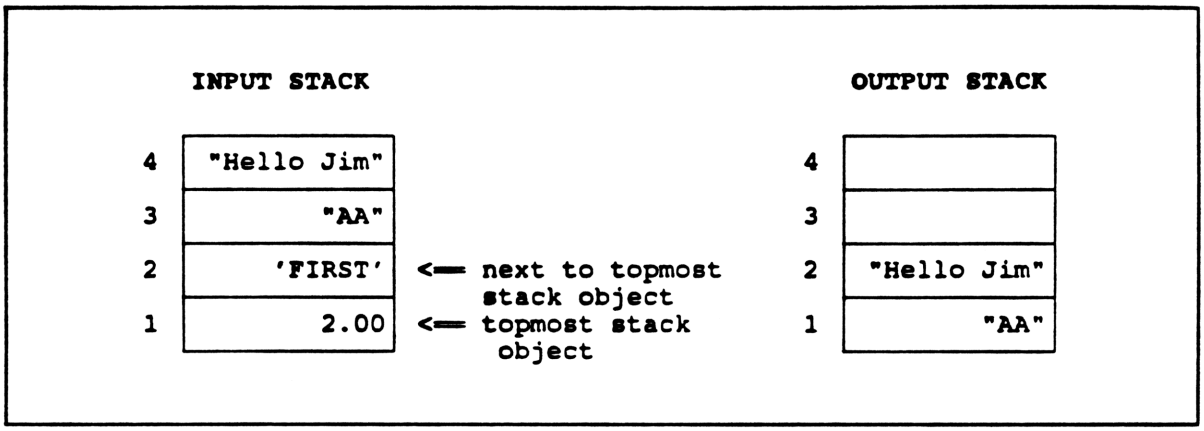


Figure 1.3. Using the DROP2 command.

4. **DROPN**. This command causes N objects in levels 2 through N + 1 to be popped off the stack. The value of N is in level 1 and does not enter in the number of dropped levels since it is always dropped. If N is less than 1 (zero or negative), it is the sole object dropped from the stack.

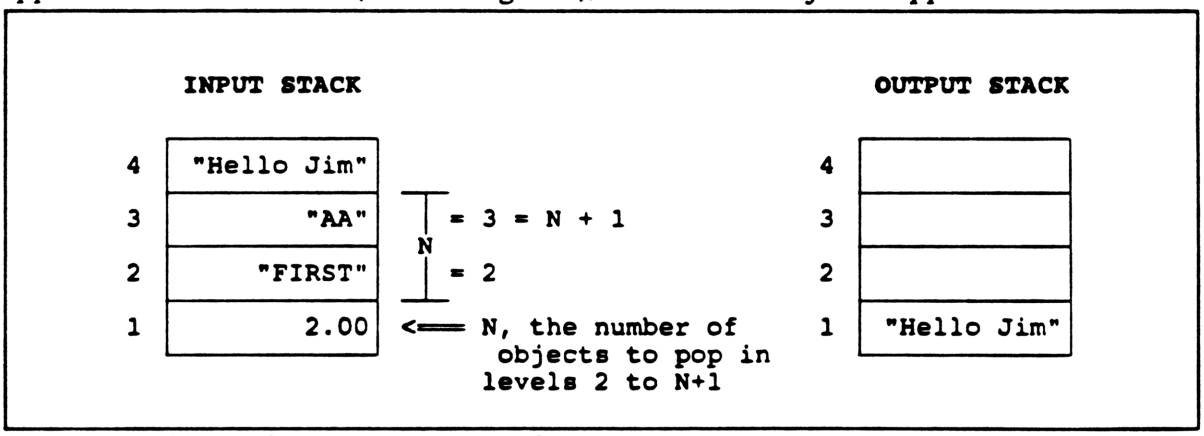


Figure 1.4. Using the DROPN command.

5. **DUP**. This command duplicates the object in level 1. This is similar to a RCL X command on the HP41C. It is a convenient way of duplicating an object before applying various functions to it.

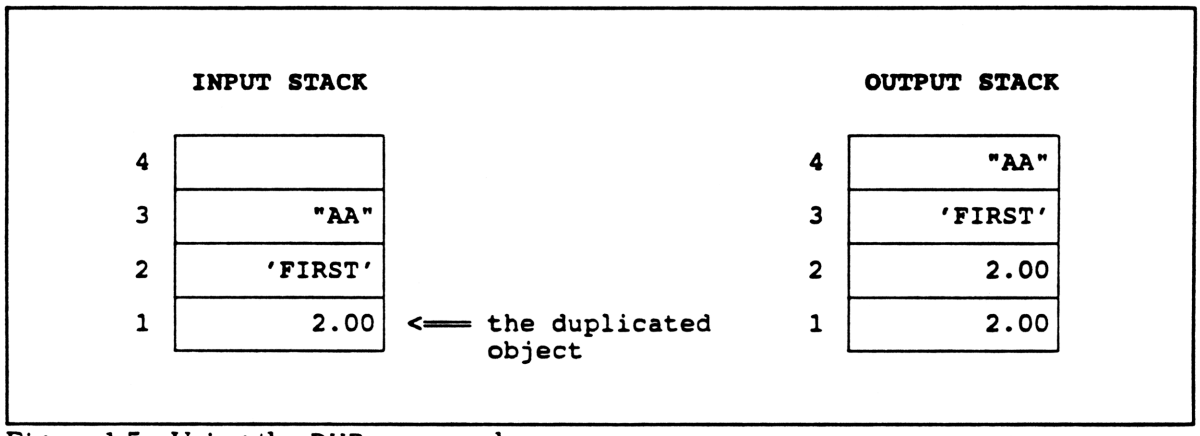


Figure 1.5. Using the DUP command.

6. **DUP2**. This command duplicates the object in levels 1 and 2. The result is that the objects in levels 1 and 3, and levels 2 and 4 are equivalent. This command is equivalent to the following set of HP41C commands:

```
STO Z
X <> Y
STO T
X <> Y
```

This command is a convenient way of duplicating object pairs before applying various functions to them.

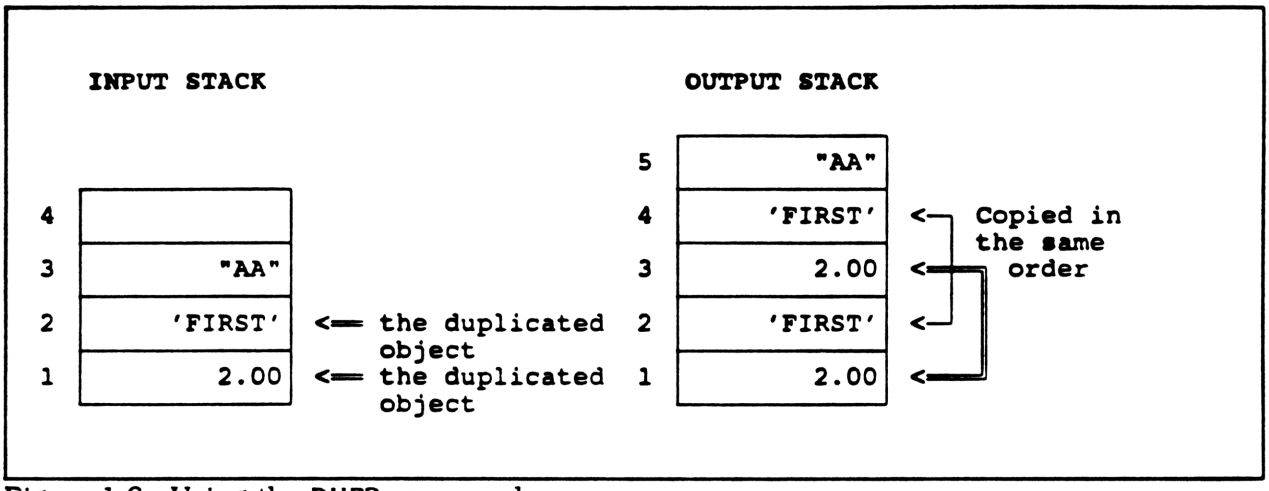


Figure 1.6. Using the DUP2 command.

7. **DUPN**. This command empowers you to duplicate N objects in the stack. The value of N is located in level 1 and the duplicated objects are found in levels 2 through N + 1. The duplication process pops N out of the stack. The order of the duplicated objects is preserved, as in the DUP2 command. This command is an example of HP48SX commands that retrieve its arguments from the stack. The DUP and DUP2 commands are equivalent to DUPN with 1 and 2 placed in level 1, respectively.

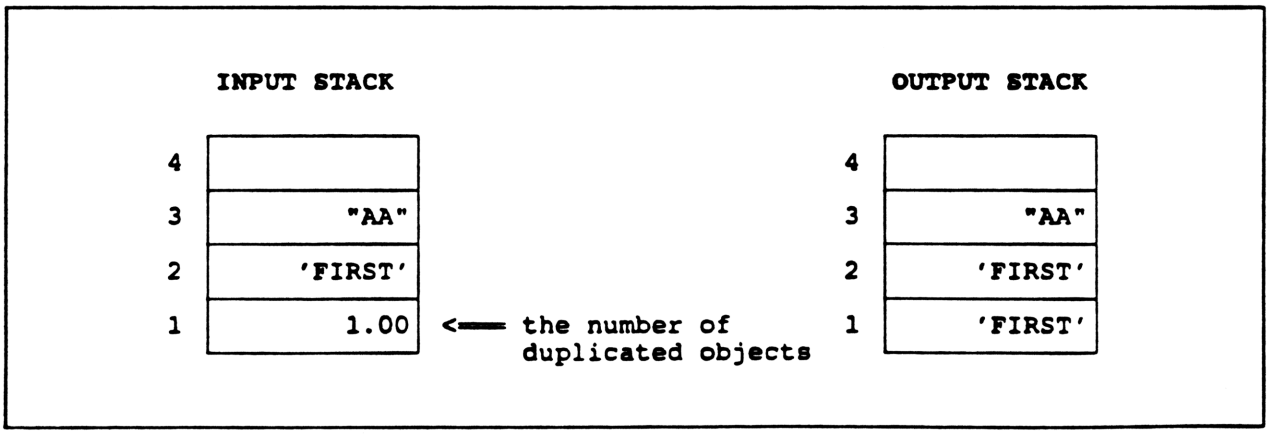


Figure 1.7. Using the DUPN command.

8. **OVER**. This command pushes a copy of the object in level 2 into level 1, and is equivalent to a RCL Y on the HP41C. The OVER command should be used instead of the SWAP command when an additional copy of the object in level 2 is needed in level 1.

9. **PICK**. This command enables you to push a copy of an object at a level greater than 1. The

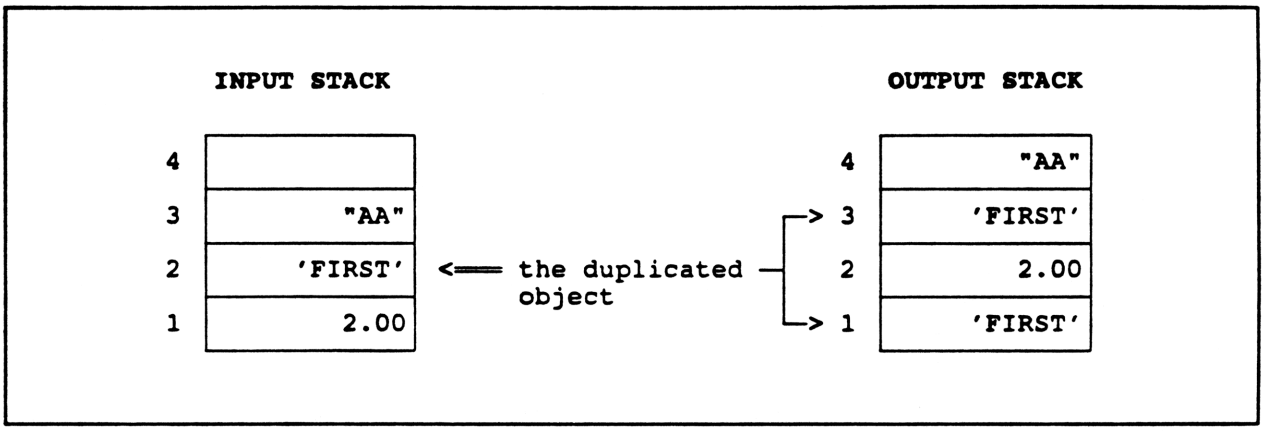


Figure 1.8. Using the OVER command.

index, N, of the accessed level is located in level 1. Therefore, the command PICK copies the object in level N+1 (that is N levels *higher* that level 1)

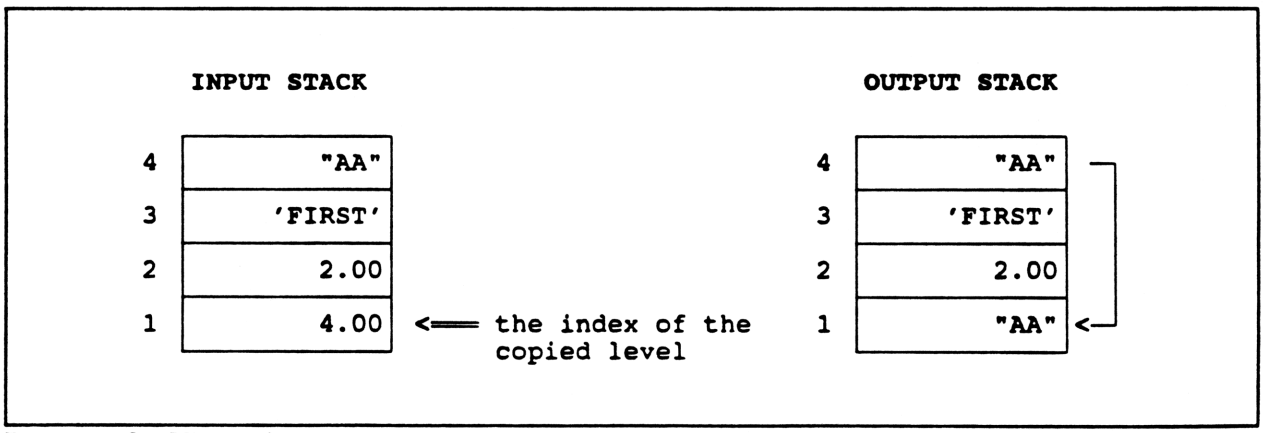


Figure 1.9. Using the PICK command.

10. **ROLL**. This command empowers you to roll up the stack. Unlike the RUP command in the HP41C, ROLL does NOT necessarily roll the entire stack. The power of ROLL comes from the fact that you can specify the levels affected. This information is placed in level 1 before invoking ROLL and represents the number of levels (from 2 and on) that are rolled. The number of rolled levels is popped out of the stack after the stack is partially or fully rolled up. To systematically roll up the entire stack use the following command:

```
DEPTH
ROLL
```

It is worth pointing out that the objects in the levels 2 through N remain in their levels. This

is due to the effect of popping the argument N off the stack and moving the object in level N + 1 to level 1 (see the effect illustrated by Figure 1.9).

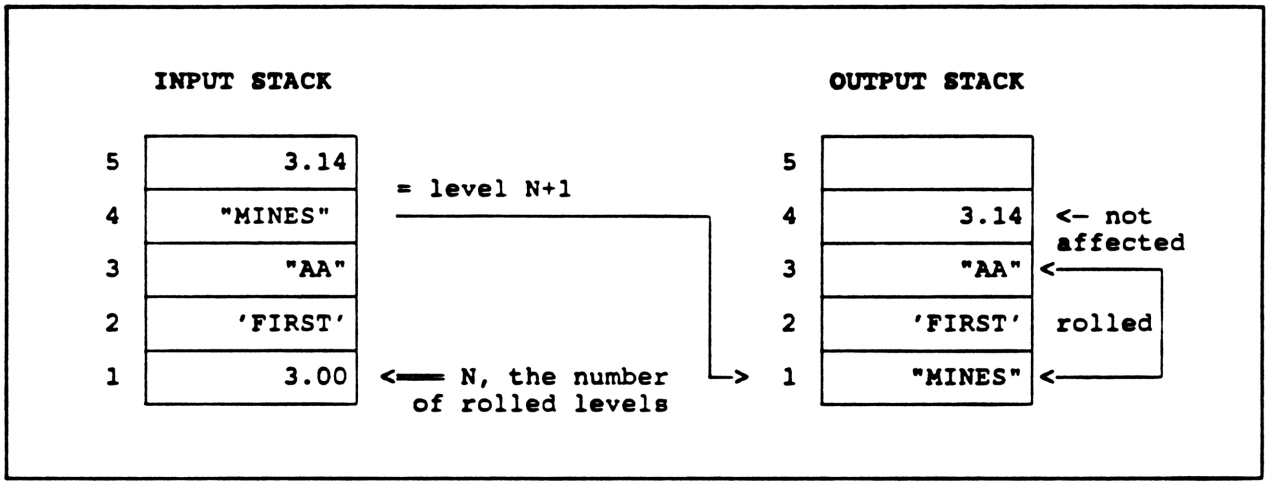


Figure 1.10. Using the ROLL command.

11. **ROLLD**. This command allows you to roll down the stack. Unlike the RDN command in the HP41C, ROLL does NOT necessarily roll the entire stack. The power of ROLLD lies in the fact that you can specify the levels affected. This information is placed in level 1 before invoking ROLLD and represents the number of levels (from 2 and on) that are rolled. The number of rolled levels is popped out of the stack after the stack is partially or fully rolled down. To systematically roll down the entire stack employ the following commands:

```
DEPTH
ROLLD
```

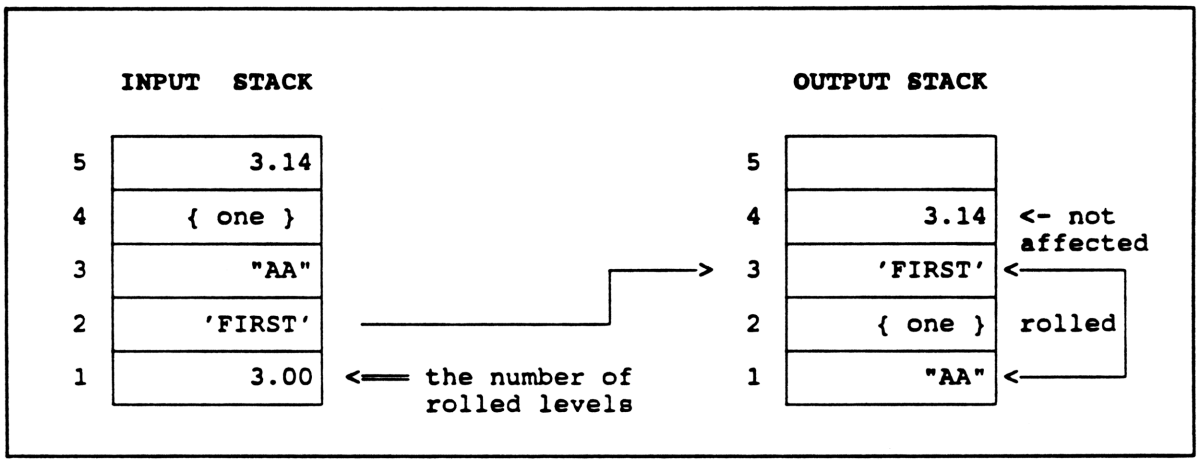


Figure 1.11. Using the ROLLD command.

12. **ROT**. This command rolls up the first three levels. It is a short hand for the 3 ROLL command.

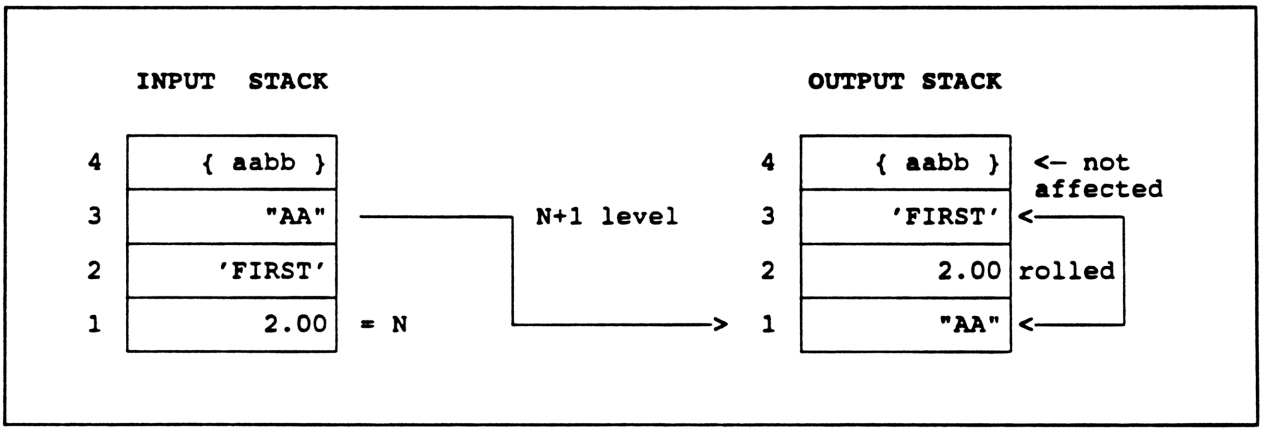


Figure 1.12. Using the ROT command.

13. **SWAP**. This command swaps levels 1 and 2. It is very similar to the X< >Y command in the HP41C.

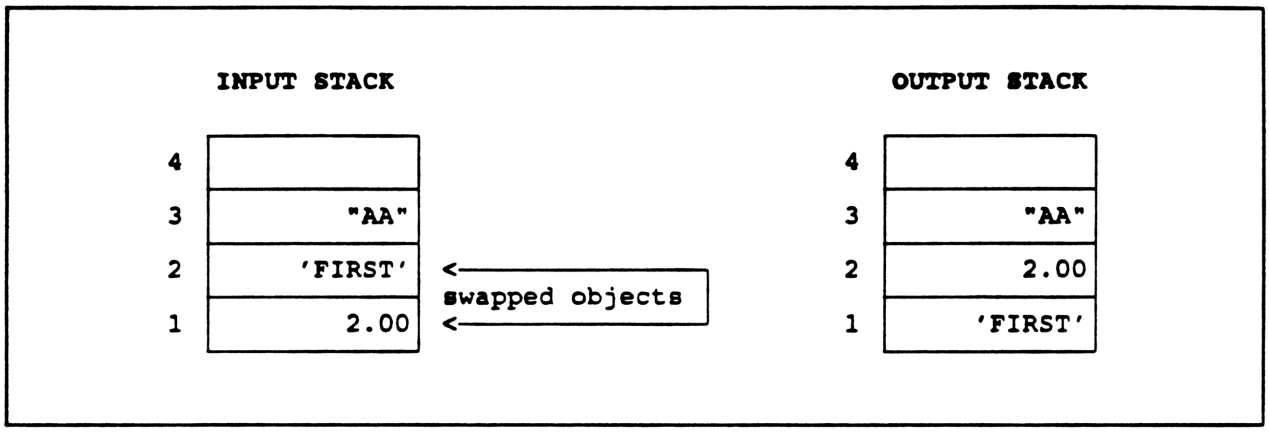


Figure 1.13. Using the SWAP command.

Recovering Arguments

The HP41C LastX command has been expanded into four last commands used to retrieve the last argument, stack, menu, and command line.

1. **LAST ARG**. This command is an expanded version of the HP41C **LastX**. While the HP41C LastX command pushed only the last value of the X register, the LAST ARG recovers all the objects that were involved in the last command issued. Figure 1.14 shows a stack before and after a multiplication is carried out. When the LAST ARG command is executed the operands of the multiplication, namely, the numbers 3 and 4, are pushed back in the stack. Notice that the result of the multiplication, 12, is retained in the stack. Setting flag -55 conserves memory, because the HP48SX does not save the last argument when this flag is set

2. **LAST STACK**. This command implements an interesting variation on LAST ARG. It works by restoring the stack to its contents prior to the execution of the last command. Unlike LAST ARG, LAST STACK eliminates any results obtained by the action of the last command. This command has a *back to the drawing board* effect. Figure 1.15 shows the effect of the LAST STACK command on the results of a multiplication operation. The original operands are restored, while the result of the multiplication is removed.

The LAST STACK command offers you the advantage of quickly recuperating from an erroneous operation. You need not drop the unneeded result and recall the original -- LAST STACK will do it for you!

3. **LAST MENU**. This command displays the last menu enabling you to move to a menu associated with another directory (more about this in a later chapter) and return back.

4. **LAST CMD**. This command enables you to recall and edit the last command you typed. You can either reissue the same command or invoke a modified version of it.

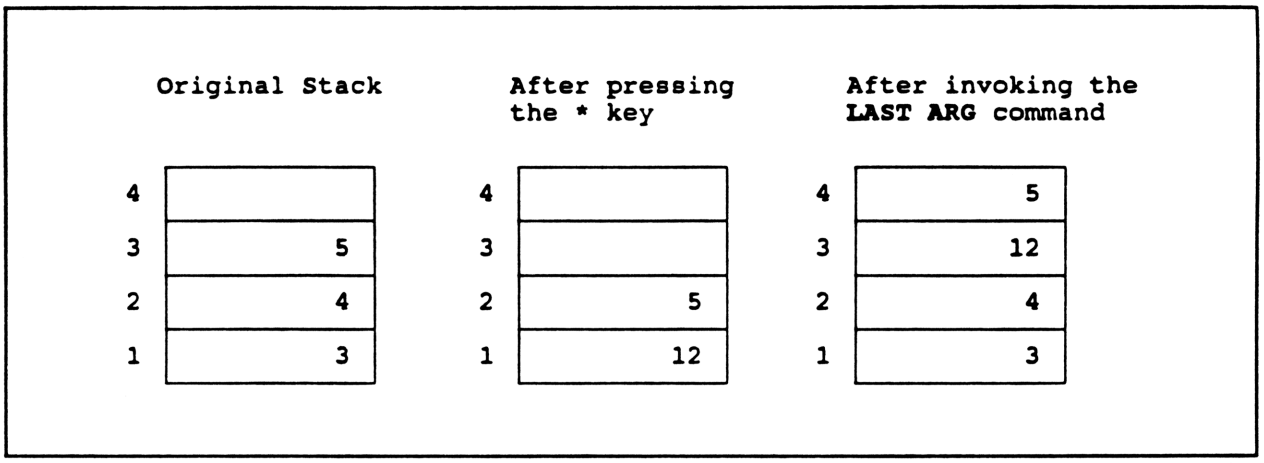


Figure 1.14. Using the LAST ARG command to restore the previous stack and eliminate the effect of the last command.

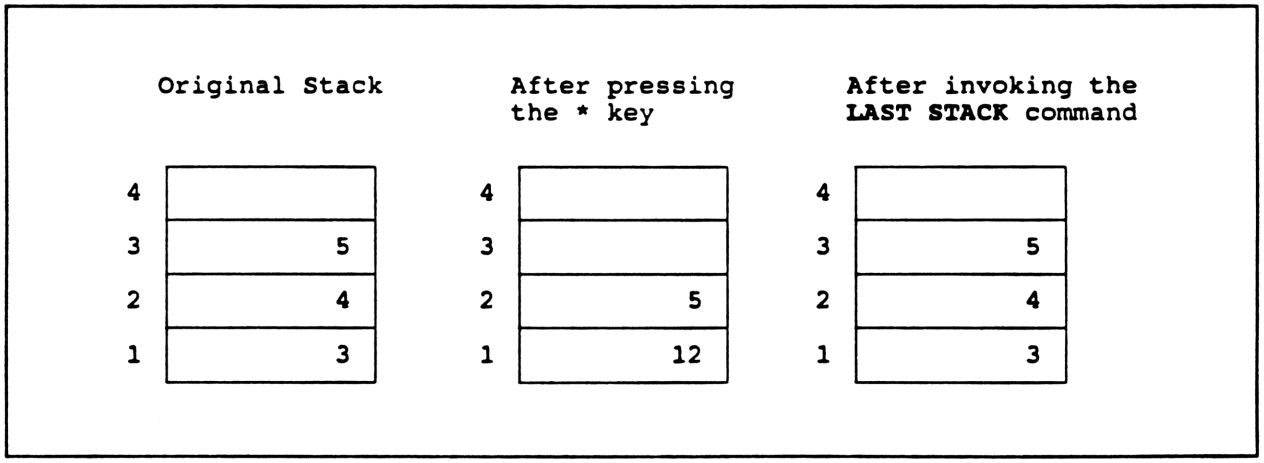


Figure 1.15. Using the LAST STACK command to restore the previous stack and eliminate the effect of the last command.

2

The HP48SX Data Types

Computers, big and small, process a wide variety of data. The versatility of computer languages can perhaps be measured by the number of data types and their complexity. The RPL language supports a good variety of data types (also called object types by the HP48SX manual). This chapter looks at the object types implemented in the HP48SX, compares RPL with other popular microcomputer languages, and offers the basics of manipulating the various object types.

Overview

The designers of the HP48SX chose a rather unique approach in defining and implementing data types for the machine. Whether you have been working exclusively with the HP41C or with popular high-level languages like BASIC and Pascal, you will find that the HP48SX is different. This difference occurs in two ways: the typed data and the variety. Figure 2.1 compares the data types of the HP48SX with those of the HP41C, BASIC, and Pascal. Looking at the table you will notice that the family of data types in the HP48SX includes rather new members, such as algebraic objects, graphic objects, directories, and libraries. While some of these objects (such as directories and libraries) are present in the microcomputer environment, they are not considered as object types by the micro users. Thus, the unique set of types in the HP48SX is custom tailored for the machine itself. This gives the HP48SX powerful capabilities for manipulating data.

Name That Type

The various object types in the HP48SX are associated with type numbers. Figure 2.2 lists the 19 object types by type number and includes examples of the object types. The type numbers are also obtained with the TYPE command.

Real Numbers

Real numbers are familiar to HP calculator users. In the HP48SX positive reals range from 1E-499 to 9.9999999999E+499. The negative reals range from -1E-499 to -9.9999999999E+499. Small numbers in the range of -1E-499 to 1E-499 are rounded to zero.

Real numbers are also used to represent date and time data. The date formats supported are the MM.DDYYYY (where 5.031990 is May 3rd, 1990) and the DD.MMYYYY (where 3.051990 is May 3rd, 1990). The time format used is HH.MMSS, with 11.3000 representing 11:30 a.m., for example.

Data Types	LANGUAGES			
	RPL	HP41C	BASICA	Pascal
Real Numbers	Yes	Yes	Yes	Yes[2]
Complex Numbers	Yes	No	No	Yes[2]
Binary Integers	Yes	No	Yes	Yes
Strings	Yes	Yes	Yes	Yes[2]
Real Arrays	Yes	No	Yes	Yes[2]
Complex Arrays	Yes	No	No	Yes[2]
Names	Yes	No	No	No
Algebraic Objects	Yes	No	No	No
List	Yes	No	No	Yes[2]
Graph Objects	Yes	No	No	No
Tagged Objects	Yes	No	No	No
Unit Objects	Yes	No	No	No
Programs	Yes	No	No	No
Directory Objects	Yes	No	No	No
Backup Objects	Yes	No	No	No
Library Objects	Yes	No	No	No
XLIB Names	Yes	No	No	No
Built-in Functions	Yes	No	No	No
Built-in Commands	Yes	No	No	No

(1) Can be created as a list of matrices or arrays.
(2) As user-defined types in Pascal.

Figure 2.1. Comparing the data types of RPL, HP41C, BASIC, and Pascal.

Real numbers are basic types that enter in the make-up of other object types, such as complex numbers, arrays, and lists. The commands that transform reals into other types are discussed in their respective sections below.

Complex Numbers

Complex numbers are represented by two real numbers and can be represented as either Cartesian or polar coordinates. The HP48SX can display them using either coordinates. Pressing the [right shift][1] keys toggles between Cartesian and polar display mode. You may key in a complex number in either coordinate form -- the HP48SX will convert your input to the current mode, if needed. In Cartesian coordinates a complex number has a real component X and an imaginary component Y, and uses the format (X, Y). Examples of complex numbers are (1,1), (3,4), and (-0.45, 12.54). Using polar coordinates, a complex number is defined by its modulus, M, and angle θ, and uses format (M, $\angle\theta$). Examples of complex numbers in polar form are (1.4142, ∠45), (-.4142, ∠54), and (42.4, ∠127). Converting from one system to another utilizes the following pair of equations:

Object Type	Type Number	Example(s)
Real number	0	1.2, -0.04, 1.2E+200, -3.65E-3
Complex number	1	(1.0345, -4.67), (20 , 45)
String	2	"THE FIRST TIME", "You", "me"
Real array	3	[1.0 3.0 6.0], [[1 2] [3 4]]
Complex array	4	[(1,2) (8,9)]
List	5	{ "Hello" "There" 2 3 (3,4) }
Global name	6	GLBNAME
Local name	7	LCLNAME
Program	8	« DUP 2 * 1 - * 5 + »
Algebraic object	9	'2*(A+1)'
Binary integer	10	#FFFFh, #28o, #234d, #11001b
Graphic object	11	GROB 100x200
Tagged object	12	:IO:'AUG1'
Unit object	13	17_m
XLIB name	14	
Directory	15	{ HOME JAMES }
Library	16	
Backup object	17	
Built-in function	18	
Built-in command	19	

Figure 2.2. HP48SX object types and their numeric code types.

$$M = \sqrt{(X^2 + Y^2)}$$

$$\tan \theta = Y / X$$

Complex numbers can also be assembled from real numbers. The HP48SX places them in the

current coordinate format. The R→C command assembles a complex number from two real numbers (see Figure 2.3). The real component of the complex number is placed in level 2, while the imaginary part is located in level 1. The C→R and the OBJ→ commands disassemble a complex number into its real and imaginary components (see Figure 2.4). The real and imaginary parts are placed in levels 2 and 1, respectively.

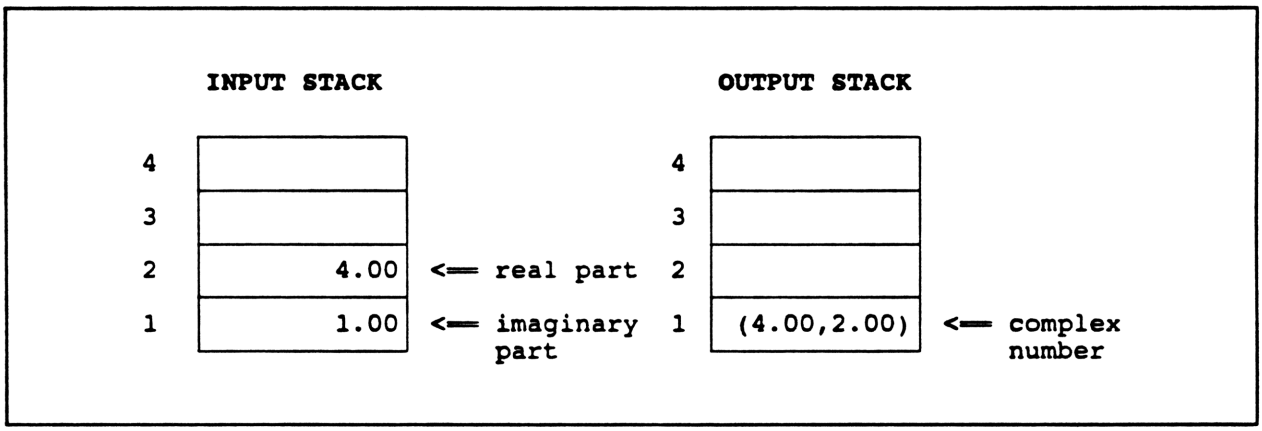


Figure 2.3. Using the R→C command to assemble a complex number from two real numbers.

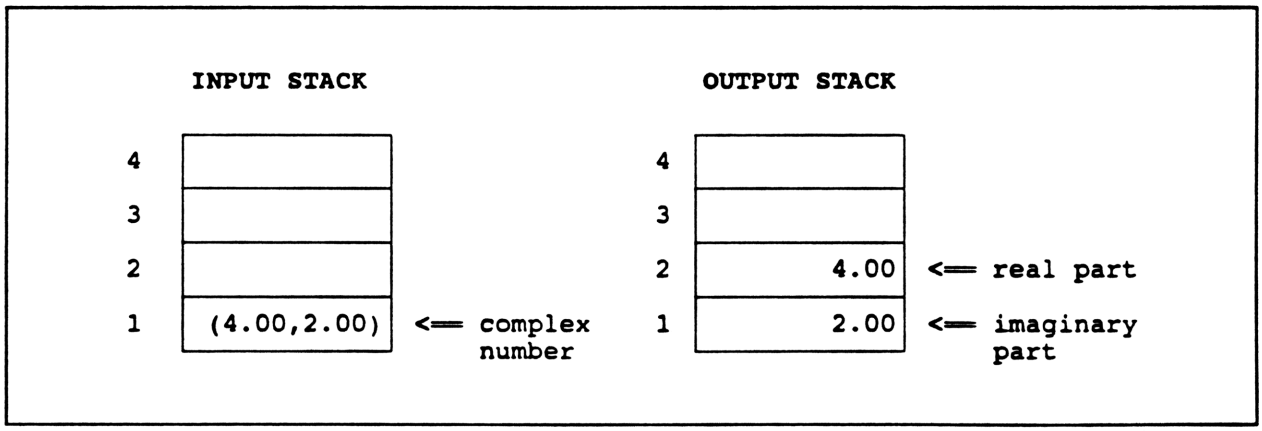


Figure 2.4. Using the C→R command to disassemble a complex number into two real numbers.

Strings

The string type is an important object type. Strings support messages and various types of text. Since the HP48SX is a machine for scientific and engineering applications, its string manipulation features seem of secondary concern by its designers.

Characters and strings can be entered directly from the keyboard. This is done by pressing the [right shift] and [-] keys to enter a pair of double quotes on the command line. To key any letter

press the [α] key. To lock on alphanumeric input press the [α] twice. When you are done, press the [α] another time to exit the alphanumeric input mode. Lowercase characters can be entered by (a) pressing the [α], [left shift], and then the character keys in ordinary input mode, or (b) by pressing the [shift left] and the character keys in alphanumeric input mode. The [left arrow] and [right arrow] keys may be used to edit the string. Pressing the [Enter] key pushes the string into the stack.

You can also obtain a character from a real number by using the CHR command. This command first rounds up the real number to the next whole number and then converts the number into the character whose numeric ASCII is equal to that number. For example, if you type 65 and invoke the CHR command, you obtain "A", the character with an ASCII code of 65. If the input number is 65.4, it is rounded down to 65 and then converted by CHR into "A". By contrast, if the input number is 65.5, it is rounded up to 66 and then converted into "B" (the character with an ASCII code of 66). Figure 2.5 shows the CHR command converting 65.4 into the letter "A".

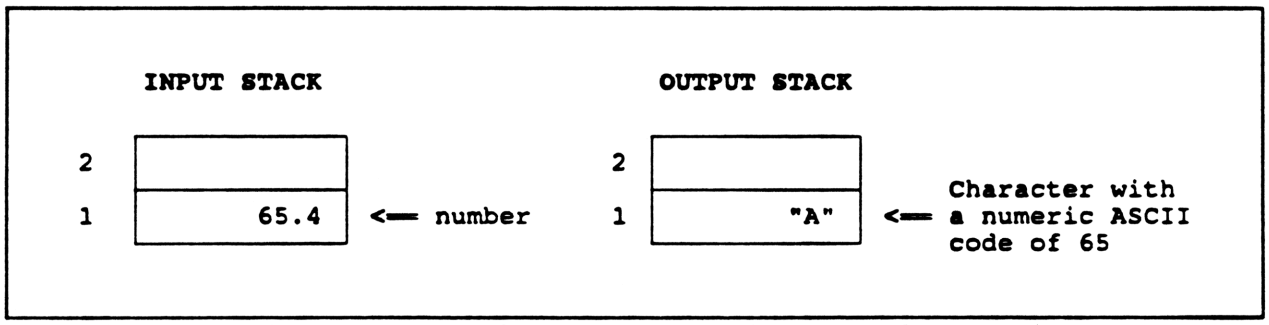


Figure 2.5. Using the CHR command to convert a number into a character.

The CHR command can be used in loops (more about loops in Chapter 7) to create special long test strings.

The reverse of command CHR is NUM. This command takes the first character in a string and returns the ASCII code in level 1. Thus, invoking NUM with the strings "A", "And", or "Add" yields 65 in all of the cases, since the first character in these strings is "A". Figure 2.6 shows the NUM command processing the string "And".

You can also build strings from smaller ones using the + operator. More about this in Chapter 5.

Real Arrays

The HP48SX supports real arrays and matrices. While arrays with more than two dimensions are not explicitly supported, they can still be implemented using lists. Arrays are enclosed by a single set of brackets. An example of an array is [1 2 3 4 5 6]. Matrices use nested sets of brackets -- each row is enclosed in a set of brackets and the entire matrix is enclosed in another set of brackets. Thus, [[11 12] [21 22] [31 32]] is a matrix containing three rows and two

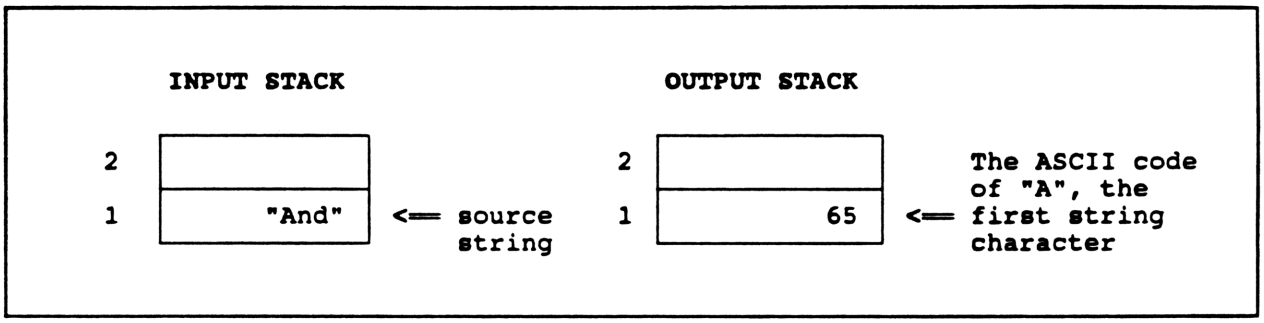


Figure 2.6. Using the NUM command to convert the first character in a string.

columns. The number of rows is equal to the number of nested pairs of open and closed brackets. The number of columns equals the number of elements in each row. A matrix MUST have the same number of elements in each row.

The arrays and matrices can be assembled by either keying in numbers in the command line, pushing numbers in the stack, or using the eloquent MatrixWriter. Keying in numbers in the command line is an easy and straightforward method for entering rather small arrays and matrices. The MatrixWriter converts the display of the 48 into a spreadsheet showing a few rows and columns of the currently edited array or matrix. The MatrixWriter comes with a set of menu options that enhance the input and editing of numbers. For example, you can specify whether you are keying in an array or a matrix. You can also widen or narrow the column width; specify the next-cell cursor movement; insert or delete columns and rows; and interact with the stack. For more details on operating the MatrixWriter consult the HP48SX *Owner's Manual Volume I.*

Concerning the assembly of arrays and matrices from stack elements, RPL applies the following scheme for arrays:

1) The members of the array are pushed into the stack. The first real number pushed in the stack becomes the first array element, and so on.
2) The size of the array is pushed into the stack. The value of the size may be pushed in the stack as a single-member list (e.g. { 3 }).
3) The →ARRY command is invoked to assemble the array.

Figure 2.7 shows how a three-element array, [10 20 30], is assembled from the stack.

Matrices (with R rows and C columns) are assembled from the stack using the following steps:

1) The C elements of the first row are pushed in the stack.
2) Step (1) is repeated for the C elements of the remaining rows (2 though R).
3) A two-element list { R C } is pushed in the stack. This (a) informs the HP48SX that you want to put together a matrix (and not an array), and (b) specifies the number of rows and columns.

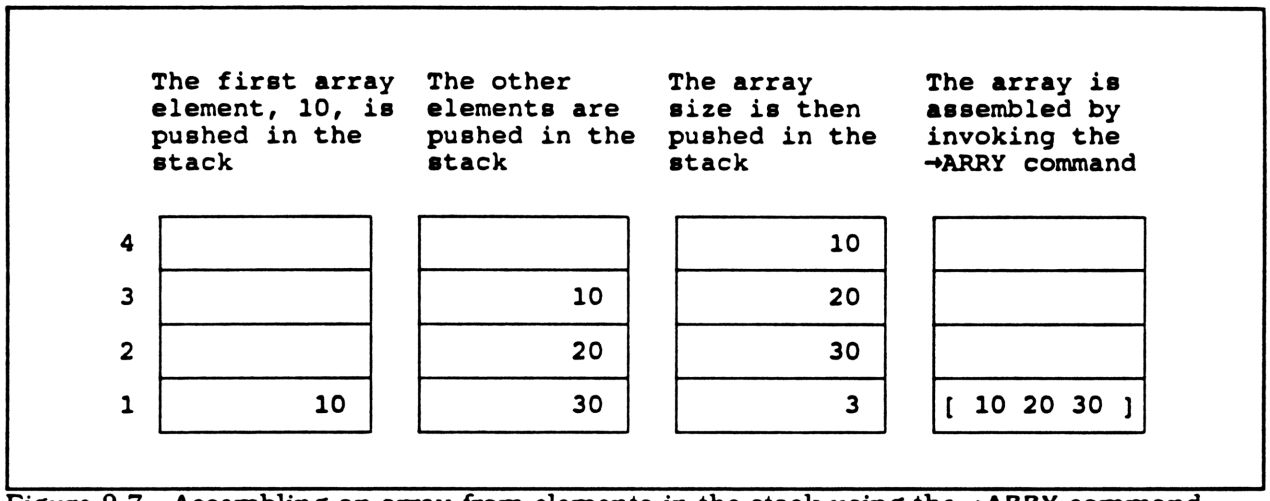


Figure 2.7. Assembling an array from elements in the stack using the →ARRY command.

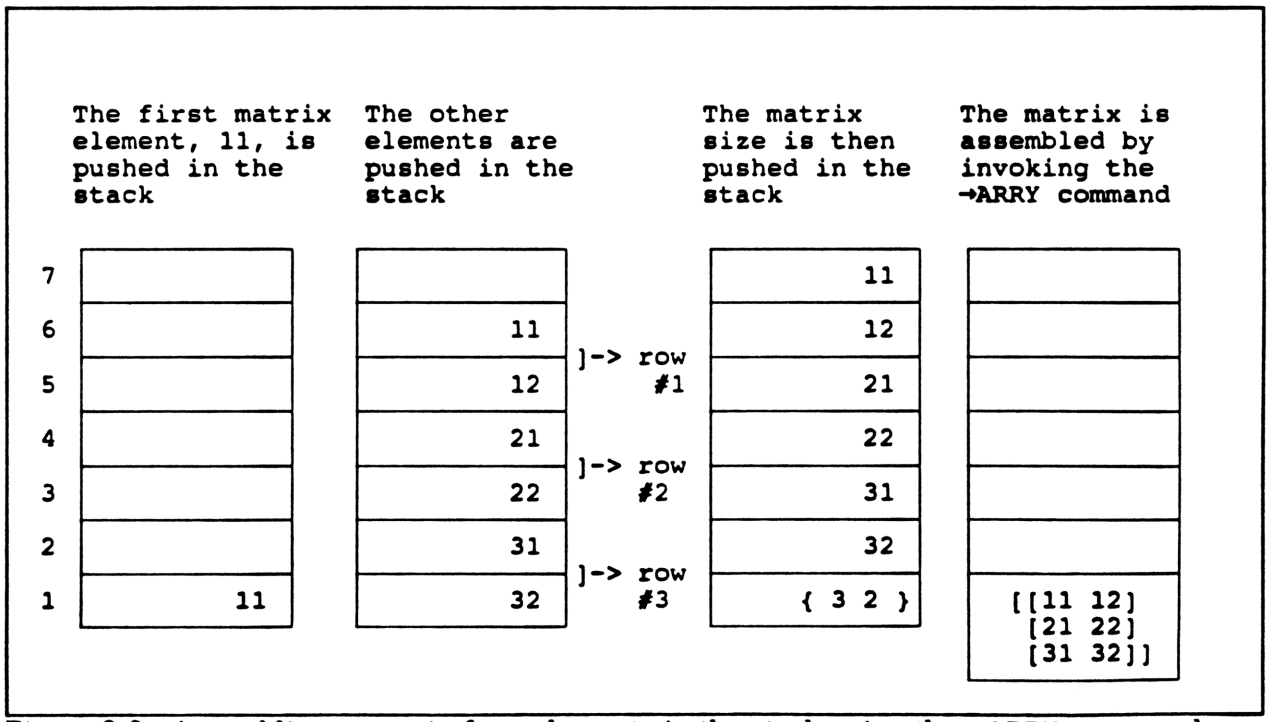


Figure 2.8. Assembling a matrix from elements in the stack using the →ARRY command.

Arrays and matrices can be disassembled, using the OBJ→ command, placing their elements and the object's dimension in the stack. The object's dimensions is placed in level 1. This action reverses the array/matrix assembly.

Programming Tip

Lists can be used as temporary surrogate arrays in the following case:

- The elements of the array are gradually obtained.
- The number of elements is not known ahead of time.
- The number of elements may widely vary.

Once the data collection is complete, the data is converted from the list to an array, via the stack.

Complex Arrays

Complex arrays are extensions of real arrays (or you can say that real arrays are special cases of complex arrays). Complex arrays and matrices are assembled in a manner very similar to real arrays and matrices. This includes the use of the MatrixWriter, command-line input, and assembly using the →ARRY command.

Programming Note

You can mix real and complex numbers when assembling a complex array or matrix using the command-line or the ARRY command methods. The HP48SX scans the input and automatically converts the real numbers into complex numbers with a 0 imaginary part. However, when using the MatrixWriter the first element you enter MUST be a complex number. This tells the MatrixWriter to automatically convert all real numbers into their counterpart complex numbers. If the first number is keyed in as a real number, the MatrixWriter will flag an error when you enter a complex number in subsequent cells!

Complex arrays can be created from two real arrays. The R→C command works with the real arrays in levels 1 and 2. The array in level 2 provides the real components, while the array in level 1 supplies the imaginary components. The C→R command performs the reverse, splitting an complex array into two real arrays.

Lists

Lists, as implemented on the HP48SX, are very powerful data structures. If you program in high-level languages such as Pascal and C, you will most likely find the HP48SX lists extremely flexible. What makes these lists versatile are the following features:

1) They are dynamic. This enables a program or a manual command to readily create them, delete them, insert new data, delete old data, alter existing data, and access list components. These operations are transparent to the user or a running program -- there is no need to explicitly reserve space for the lists.

2) They are heterogeneous. The HP48SX lists may contain members of just about every supported types, including other lists, strings, and arrays! The ability of the HP48SX list to contain members of varying object types opens the door for the creation of records. The lists used to contain records of information can in turn be members of a bigger list, namely, the host data base!

3) They support array-style access. This makes the HP48SX list a hybrid (and a great one at that) between true lists and true arrays. These lists can be the repertoire for arrays of strings, vectors, matrices, sublists, programs, etc (see Chapter 9). The possibilities are simply awesome.

The smallest list is the empty list, { }. List elements are separated by spaces. Examples of lists are shown below:

```
{ 1 2 3 4 }
{ "Namir" "Clement" "Shammas" }
{ 1 (1,2) "Hello" 'X1*Z2+2' { 1 "Me" (2, 3) } [ 1 2 ] [ [ 1 1 ] [ 2 2 ] ] }
```

The first two examples show homogeneous lists containing numbers and strings, respectively. The third example shows how sophisticated lists can get. In the above example the list contains a real number, a complex number, a string, an algebraic expression, a nested list, a real array, and a real matrix. It is worth pointing out that nested lists are considered as single elements in the host lists. Thus the list { { 1, 2 } } contains one element, the nested list { 1, 2 }.

Lists can be created using the →LIST command. This command resembles the →ARRY command used to create arrays. The →LIST command requires that you first push the list members in the stack, and then push the list size. Figure 2.9 shows how this command works in creating a new list from its components. The OBJ→ command performs the reverse action of the LIST→ command. The list located in level 1 is decomposed into its elements and its size. The list size is located in level 1.

You can add new list members by using the + operator. More about this in Chapter 5.

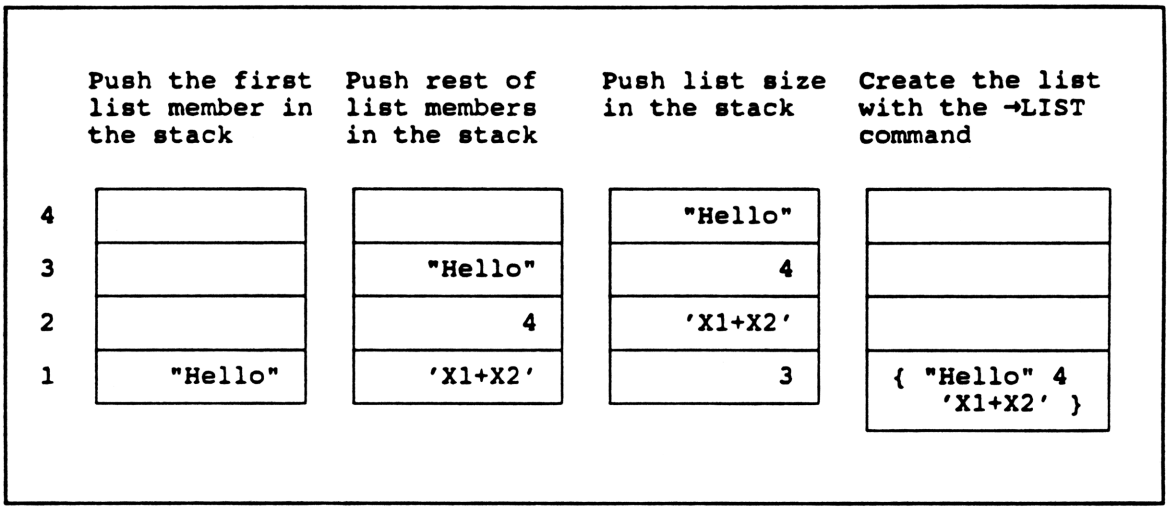


Figure 2.9. Creating a list using the →LIST command.

Global and Local Names

The RPL language on the HP48SX associates names with data and objects. If you are a veteran HP41C programmer you have been used to storing data in numbered registers. This feature of RPL is a welcome one, since it enables you to associate data objects with named containers. The advantage is that the names used to store data objects can be selected to be representative of the type of information stored. If you are familiar with BASIC, names should be familiar to you.

Global names, as the name might suggest, are names of object container or variables. The HP48SX requires that variable names be enclosed in a pair of tick (a.k.a. single quote) characters. This enables the machine and RPL to distinguish between variable names and strings of characters. Examples of names are:

'Velocity'
'Pressure'
'Weight'
'X1'

Unlike strings, names are associated with data objects. These objects may be real numbers, arrays, strings, lists, matrices, graph objects, etc.

The HP48SX makes a distinction between global and local names. A global name makes its contents accessible (or public, if you prefer) to all programs. By contrast, a local name offers limited access to its contents, offering a certain aspect of data hiding. This access depends on where the local name is located.

For the HP41C programmer, local names bring forth a new level of data hiding. The same can

be said for those readers who work with BASIC interpreters where all variables are global. If you are familiar with QuickBASIC, then you already know about global and local variables.

Program Objects

Programs are considered object types by the HP48SX! Yes, this might come across as being very unusual to the majority of us. Even languages like BASIC, C, Pascal, and many others do not regard programs as special data types. Programs are enclosed in a pair of left and right double Guillemet characters, « and ». Pressing the [left shift][-] keys inserts a pair of double Guillemet characters. Examples of programs are shown below:

```
« SQ 1 + »
« DUP DUP * * 1 + »
```

The first program squares the number in level 1 and adds 1 to it. The second takes the cube of the number in level 1 and adds 1 to it.

Programs, especially simple ones, can be pushed into the stack and then run by pressing the [EVAL] key. The program object is removed from the stack. This is not the usual way for executing programs. They should be stored first in a variable so that they can be retained for repeated execution, and edited if need be.

Algebraic Objects

What sets the HP48SX apart from all the previous handheld machines is its ability to perform symbolic manipulation of algebraic equations. For example, if you give the HP48SX an equation like 'A+B=C+D' and ask it to solve for C, you get 'C=A+B-D'. The object type involved here is the algebraic object which consists of an equation or an expression (an expression differs from an equation by the absence of an assignment equal sign). Algebraic objects are enclosed in single quotes. Examples of algebraic objects are shown below:

```
'2*X'
'X=Y/2.5+Z'
'X=SQ(A+5.5)'
```

Basic algebraic objects are simply pushed into the stack. You can build algebraic objects from smaller ones using the four math operators, change of sign operator, and most math functions. In all of these cases, the HP48SX applies the operators and functions symbolically. While algebraic manipulation is not within the scope of this book, I encourage you to experiment with this fascinating feature.

Binary Integers

The HP48SX supports another genre of numbers, namely binary (unsigned) integers. The term unsigned means that the integers have 0 and positive values, and therefore never use a minus sign. This type of integer is of interest to computer scientists and programmers. The general

syntax of binary integers is shown below:

Syntax	Class of Integer	Digits Range
# hexadecimal digits h	hexadecimal	0 to 9, A to F
# decimal digits d	decimal	0 to 9
# octal digits o	octal	0 to 7
# binary digits b	binary	0 to 1

Table 2.10. The general syntax for binary integers.

All binary integers start with the pound character, #, and end with a base designator. This designator also enables the HP48SX to check the validity of your input.

You can specify the current mathematical base to display these binary integers. The HP48SX lets you key in a binary integer in any valid base. Conversion is made by the machine to make your input conform to the current base. Examples of various binary integers and their decimal equivalent are shown below:

Number	Type	Decimal Equivalent
#12h	hexadecimal	18
#FFh	hexadecimal	255
#1Bh	hexadecimal	27
#33o	octal	27
#07o	octal	7
#10d	decimal	10
#99d	decimal	99
#11b	binary	3
#101b	binary	5

Figure 2.11. Examples of binary integers.

The R→C command converts non-negative real numbers into binary integers. The real numbers are first rounded and then converted to the current base. Using negative numbers results in an error. The B→R command performs the reverse, converting binary integers into reals.

Binary integers are available to the HP41C programmer through the PPC and Advantage plug-in ROMs. A number of BASIC interpreters and compilers support binary integer constants in hexadecimal, octal, and binary bases.

Graphic Objects

Graph objects represent a matrix of graphics pixels (short for picture elements). The HP48SX possesses a sophisticated graphics system for plotting functions as well as drawing shapes. The topic of graphics deserves a separate book. These graphs are treated as distinct objects that can be placed in the stack and appear in the form:

GRAPH *n* by *m*

where n and m denote the graph height and width in pixels.

Unit Objects

One of the major drawbacks in computer calculations is the disassociation between units and numbers. This makes engineering calculations vulnerable to erroneous input. For example, a program prompts you to enter the pressure in atmospheres. You mistakenly enter the correct value expressed in psi (pound per square inches). There is no easy way for the computer to discern the intended units. This problem has been tackled by the HP48SX, since it is a handheld machine aimed at engineers and scientists. Units are special objects made up of a real value and an accompanying unit of measurement. Unit objects have the following general syntax:

value_unit

Examples of unit objects are shown below:

```
8_m
12.3_lb/ft^3
0.23_kg/(m*s^2)
```

You can easily key in a unit object from the command line. Simply type in the value associated with the unit, press the [right shift][×] keys (to type the underscore character), and type in the units.

An alternate way of entering the unit objects is the use of the UNITS set of menus. These menus enable you to select the exact units to be associated with the number in level 1.

Invoking the OBJ→ command with unit objects causes level 2 to contain the value associated with the original unit, while level 1 contains a unit value (that is 1_*unit*).

Unit objects can be propagated by the math operations and function. For example, pressing the square root key with 25_ft^2 in level 1 returns 5_ft.

Tagged Objects

The implementation of units in the HP48SX represents an important step in dealing with engineering calculations. Associating units with numbers greatly clarifies data. Another tool that clarifies information is tags. They are labels that are attached to data, separated from each other by a colon. The general format for using a tagged object is:

tag_name : *data*

Examples of tagged objects are shown below:

```
TODAY_TEMP:80_F
SLN_VECTOR:[1.4 5.6 -9.6]
RECORD_STRUCTURE: { "LastName" "FirstName" "Address" "City" "State" "Zip" }
```

Tagged objects can be created from the command line or assembled from data in the stack. To create a tagged object from the command line you employ the :: command (by pressing [right shift][+] keys). This puts the two colons in the command line and places the editing cursor in between these colons. Type in the tag name, and then press the right arrow key to exit the tag. Now you proceed in typing in the accompanying data.

To tag information in the stack the tagged data must be in level 2 and the tag name in level 1. The tag name may either be a name or a string. By invoking the →TAG command, the tag and the data are merged. Figure 2.12 shows how the →TAG command creates the object NAME:"Lia" from the string "Lia" and the name 'NAME'.

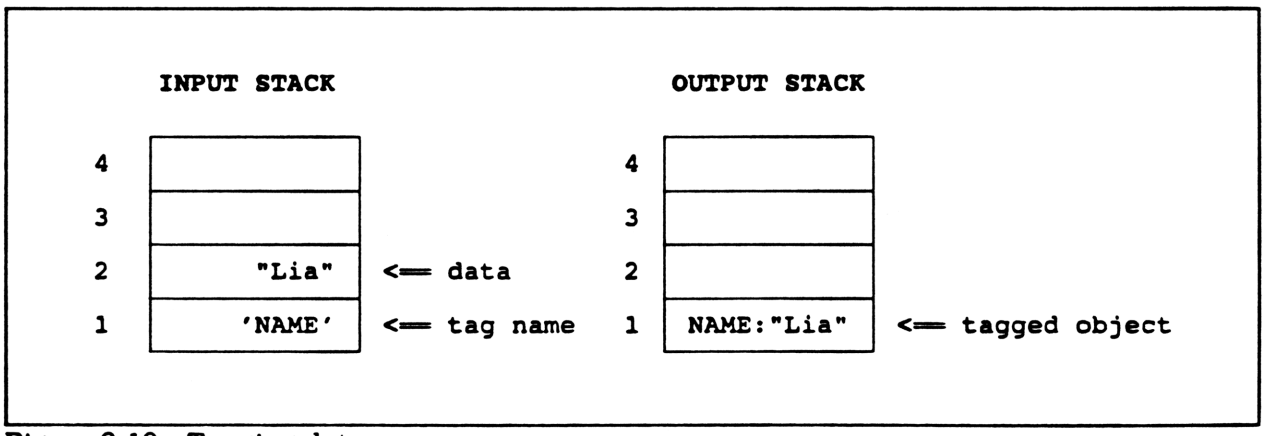


Figure 2.12. Tagging data.

The OBJ→ command reverses the action of the →TAG command. The tagged object is divided into two objects: the tagged data (in level 2) and the tag name (placed as a string in level 1).

Tagged data can be explicitly detagged using the DTAG command. Detagging occurs automatically with most operations.

Directory Objects

This category of objects deals with the hierarchy of directory structures. The HP41C contained all you programs and data registers in a single area. By contrast, the HP48SX offers a more sophisticated level of *work areas*. These work areas are placeholders for variables that store data and programs (I am using these terms in a general sense). This prevents the user from cluttering the same area (especially with the amount of RAM made available to the machine) with numerous objects that belong to different projects. Each work area is called a directory. The HP48SX supports a hierarchy of directories. This means that the directories are connected to each other using parent-child relationships. Directories are objects created using named objects. The same names are used to visit a directory and delete it. Chapter 3 gives you more details about directories.

Other Objects

In addition to the above objects there are a number of special objects. They are:

- Backup Objects. These are objects associated with backups.
- Library Objects. These are objects associated with libraries offered by plug-in cards, or they may reside in RAM.
- XLIB Objects. These are special objects offered by plug-in cards.
- Built-in Functions. These are considered as built-in program objects.
- Built-in Commands. These are also considered as built-in program objects.

Notes

3

Directories, Variables, and Programs

This chapter looks at how the HP48SX uses directories to provide you with different work areas. I will discuss the concept of directories, how they are created, connected, and removed. In addition, I will address how to move between directories. The second topic of this chapter is variables and how their scope is related to the directory hierarchy. The third chapter topic is program objects. You will learn about local program variables, programs using global variables, programs calling each other, and program debugging.

The HP48SX Directories

Imagine for a minute that all the internal walls are removed from your dwelling place! Suddenly you can spot just about every object inside your place. Your eyes would be invaded by a clutter of objects of different shapes and colors. Your mind would also easily tire just looking at things. The bigger your house, the worse it is. Thus, it becomes apparent that internal walls play a valuable role in eloquently structuring your house or apartment into semi-independent rooms. Moreover, these rooms are interconnected in a certain order and sequence.

The above analogy can be applied to the memory of the HP48SX, especially one with additional RAM cards. If the memory offers no partition, then you end up with a single work area that contains ALL your objects! Yes, ALL OF THEM! The HP48SX offers your objects room-like partitions called directories. These directories are interconnected in a special parent-child hierarchial fashion. The HP48SX has a root directory, called HOME. It's always there. The first child directory (or subdirectory) you create is attached to HOME. Additional subdirectories may be attached to HOME or any other existing subdirectory. Figure 3.1 shows a sample directory tree structure. The structures has four levels of subdirectories. The directories STAT, LIST, STRING, and MATH are all attached to the root directory HOME. The directory STAT has one subdirectory attached to it, namely, REGR. The directory STRING has two subdirectories attached to it, namely, WORD and ITEM. The directory MATH has one subdirectory attached to it, namely, OPTM. The latter is also attached to a subdirectory, NLR.

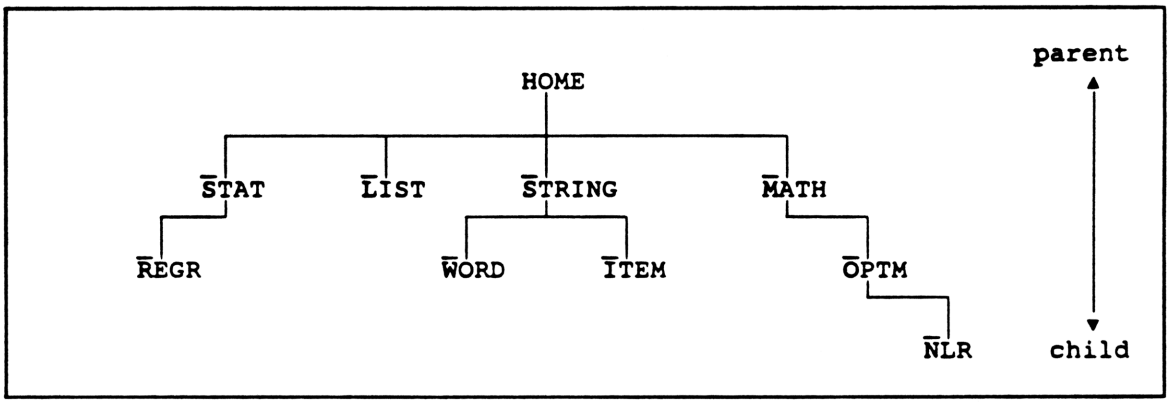


Figure 3.1. A sample directory structure.

Programming Tip

The sooner you employ a directory structure to organize your work, the better.

Creating a New Subdirectory

A new subdirectory is created using the CRDIR command (obtained by pressing the [left shift] [VAR] keys) CRDIR and is attached to the current directory. Once created, the name of the subdirectory appears as a new option in the VAR menu. The name of the new subdirectory must be a name object enclosed in tick characters. For example, to create the STAT subdirectory, enter "STAT" CRDIR.

Removing a Subdirectory

A subdirectory can easily be removed in one of two ways. The first method requires that the subdirectory be empty. The target subdirectory is first chosen by pressing the tick key and entering the subdirectory name or selecting it from the menu. Invoking the PURGE command performs the sought deletion. This command does not work with non-empty directories. The HP48SX displays an error message to that effect. The CLVAR command empowers you to purge all variables from a directory before removing the directory itself using the PURGE command.

The second method performs a quick, decisive deletion of a subdirectory using the PGDIR command. This command wipes clean the target subdirectory (and all of its subdirectories) in one swoop.

Warning!

Use the PGDIR with great caution.

Moving to Another Subdirectory

You can move between two linked subdirectories. Press the VAR key to obtain the menu of objects in the parent directory. The name of the target child directory is selected from that menu. You might need to press the [NXT] key a few times to locate the sought directory. The HP48SX updates the name of the current directory, displayed at the top left corner of the display.

The HP48SX offers two commands, found on the keyboard, to move upward. The HOME command, invoked by pressing the [right shift][']keys, moves you to the root directory. The UP command, invoked by pressing the [left shift][']keys, moves you to the parent directory. The UP command appears as the UPDIR command in a program. The UP (and its alias UPDIR) command have no effect if you are already in the root directory.

The Path to Your Door

It is worth pointing out that the names of subdirectories need not be unique. The HP48SX does not object to using the same subdirectory name at different directory levels. The name HOME is the exception -- you cannot create any subdirectory called HOME.

A directory path is a list of directories that form a branch in the directory tree structure. For example, the path of the NLR subdirectory is the list { HOME MATH OPT NLR }. This list specifies the names of all the connected directories, from HOME to the current ones. The PATH command (found in the first set of MEMORY menu options) returns the current path.

The significance of the path lies in the fact that RPL supports a special inheritance scheme between directories. The following simple rules apply:

1) The variables of ancestor directories are inherited by (or visible to, if you prefer) child directories.
2) The programs stored in variables located in the ancestor directories can be executed in the child directories.
3) When a child directory declares a variable (with data or programs) with a name matching an inherited variable, the latter becomes opaque. From then on, the new version of the same variable is visible to that child directory and all of its subdirectories.

This RPL feature is very powerful. You can place general objects at higher directory levels, while locating more specialized objects in deeper subdirectories.

Variables

This section looks at variables as data containers. One of the first tasks a new HP48SX user learns is to store and recall data in memory. Veteran HP41C programmers find this new way of storing and recalling data a bit strange! The old practice of pressing STO and then two digits is replaced by a new ritual. The name of the variable must be pushed in the stack (or located in the command line) before pressing the [STO] key. Recalling data on the HP48SX is even stranger for the HP41C user. First, there are several ways to do it:

1) Just press the VAR menu option with the name of the sought variable. This works if the variable is stored in the current directory level.
2) Simply enter the name of the variables (with no tick characters). Press the [Enter] key and voila! The contents of the variable are retrieved.
3) Enter the name of the variable enclosed in the tick characters and press the [EVAL] key. The contents of the variables are placed in level 1.
4) Enter the name of the variable enclosed in the tick characters and press the [left shift][STO] keys (this invokes the RCL command). The contents of the variables are placed in level 1.

Variables created on directory levels are basically global variables. They can be accessed by all programs running launched in that directory and all of its subdirectories. As explained in the last section, global variables can have a limited scope of visibility. This occurs when they are overshadowed by other variables (bearing the same name) residing in child subdirectories. Global variables are also overshadowed by local variables, declared inside programs. More about this in the next section.

Programs

RPL programs are special objects that are characterized by the following:

1) Programs bring forth a new level of object visibility.
2) Programs can declare their own local variables and nested program objects.
3) The scope of local variables in restricted to the program level where they are declared, as well as the nested program objects.
4) Programs interact with their environment using the stack and global variables. The stack is a convenient channel for passing data back and forth. Global variables can also provide an adequate source of data input. RPL allows programs to create global variables and store output data in them.
5) Programs that implement functions return one or more results in the stack.
6) Programs that implement procedures usually do not return any data (under normal conditions). Their main role is carrying out a task.

Programs are usually stored in variables. They are executed when their host variables are selected from the menu, or when the names of these are evaluated. To edit a program, enter the name (enclosed in tick characters) of the host variable and execute the VISIT command (press the [right shift][+/-] keys). The HP48SX puts you in edit mode. When you are finished editing, press the [Enter] key. Press the [ON] key to abort program editing and entry.

Programming Tip

Make a temporary duplicate copy of a program by storing it in another variable when:

- Editing an important program.
- Editing a large program.
- Creating an improved version of a program.

Remove the older program versions when the new ones are satisfactorily operating.

Programs vary from very simple to complex. In the rest of this section I will present small programs that give you an idea of the variation in complexity. Consider the following simple program object:

```
« SQ 1 + »
```

This program takes one argument from the stack, squares it, and adds 1 to it. The result is left in level 1 of the stack. This makes the program behave as a function. The next program acts as a procedure:

```
« UPDIR EVAL »
```

The program makes two directory moves. First, it takes you from the current subdirectory and into the parent directory. Second, it takes you to a subdirectory whose name is in level 1 (asuming the program is stored in a variable). The overall effect of this procedure-type program is to move you to a sibling subdirectory.

Using Local Variables

The next program demonstrates the use of local variables to store intermediate data. This process involves moving data from the stack and into the local variables. The right arrow symbol, →, is used to execute such a data transfer. The general syntax is:

→ *local_variable_1 local_variable_2 ... local_variable_n*

Figure 3.2 shows the order of transferring data between the stack and local variables. For the novice RPL programmer, this order is not too obvious.

After you assign data to local variables your program must contain a pair of left and right double Guillemet characters (that is, « and »). This creates a nested program object inside which

Programming Note

When transferring N stack elements to local variables, the element in stack level i is stored in the (N+1-i)'th local variable.

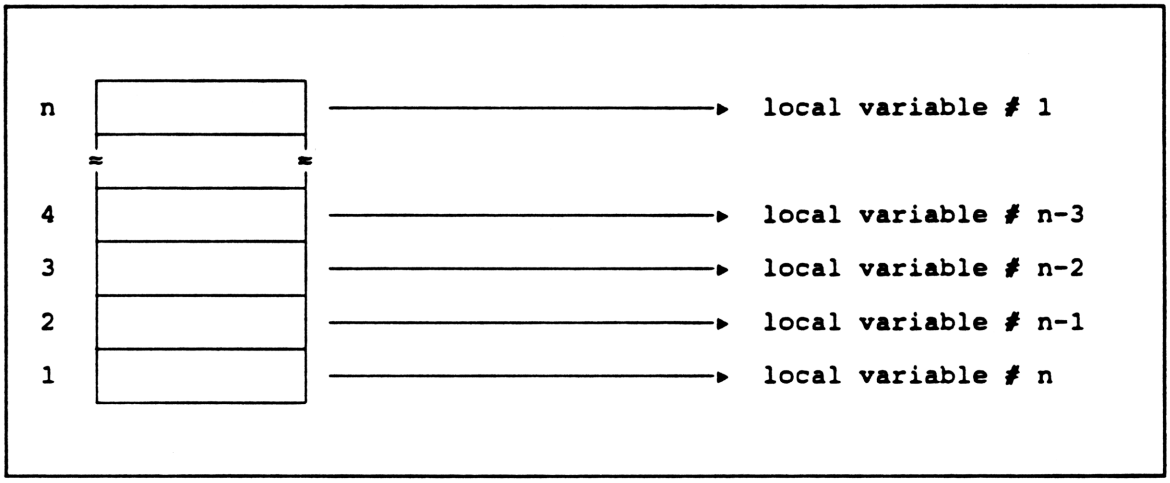


Figure 3.2. The order of moving data between the stack and local variables.

the local variables are accessible. Once the program passes the nested » character, the local variables are removed by RPL.

The next program implements the following two-variable function:

$$f(X, Y) = X / Y + Y / X$$

Rather than performing stack manipulations, the program moves the values of X and Y from the stack into local variables (bearing the same names). This makes the program easier to read than a version that relies more on the stack. The program is shown below and includes comments that appear on the right in different character fonts:

```
; Listing 3.1.  Program to evaluate f(X,Y) = X/Y + Y/X
«
→ Y X                                   ; assign element in level 2 to local variable Y
                                        ; assign element in level 1 to local variable X
      «
      X Y /                             ; calculate X / Y
      Y X /                             ; calculate Y / X
      +                                 ; add the two products
      »
»
```

The assignment to the local variables X and Y is followed by a nested program object. The scope of variables X and Y is confined to the nested object.

Using Algebraic Objects

The second example shows that a local variable is used to store data appearing in an algebraic expression. The program implements a function that evaluates the following function:

$$f(X) = 2X^2 - 5X - 4$$

The program is shown below:

```
; Listing 3.2.  Program to evaluate f(x) = 2X^2 - 5*X - 4
« → X                                   ; store data in local variable X
      «
      '2*SQ(X)-5*X-4'                   ; push algebraic object in stack
      EVAL                              ; evaluate expression
      »
»
```

> *Programming Note*
>
> Storing data in a local variable is recommended if the data appears in an algebraic object. This protects global variables from being accidentally overwritten.

Multi-Level Programs

The next example shows assignment to local variables appearing in two levels. The program solves for the roots (both real and complex) of a quadratic equation:

$$A X^2 + B X + C = 0$$

The quadratic coefficients A, B, and C are stored in local variables bearing the same name. The solution is obtained using the following equation:

$$\text{roots} = (-B \pm \sqrt{D}) / 2A$$

where the D determinant is equal to $(B^2 - 4 A C)$.

The program transfers the coefficients A, B, and C from the stack to the local variables A, B, and C. The square root of the determinant D is calculated and stored in the local variable S. This example shows that local variables are also used to store intermediate results. In addition to variable S, the local variable TWOA stores the value of 2*A. The program is shown below:

```
; Listing 3.3.  Quadratic Solver version 3.1
```

```
«
→ A B C                                  ; store data in local variables A, B, and C
      «
      B SQ                               ; calculate B squared
      A C 4 * *                          ; calculate 4 A C
      -                                  ; obtain the determinant
      √                                  ; get the square root of D
      A 2 *
      → S TWOA                           ; assign the square root of D to local variable S
                                         ; assign 2*A to the local variable TWOA
            «
            B NEG                        ; calculate the first root
            S +
            TWOA /
            B NEG                        ; calculate the second root
            S -
            TWOA /
            »
      »
»
```

Programming Note

The number of nested program objects (which is also the number of program levels) is equal to the number of assignments to sets of local variables (which is also equal to the number of times the → operator is used).

The above program also illustrates that the scope of the local variables (B in particular) extends to the second nested program object.

Programming Notes

RPL applies the inheritance and visibility scheme of directories to local variables and multi-nested program objects.

The next program offers a more algebraic version of the quadratic root solver. Notice that the use of algebraic objects enhances the program's readability:

```
; Listing 3.4. Quadratic Solver version 3.2
«
→ A B C                                  ; store data in local variables A, B, and C
      «
      '√(SQ(B)-4*A*C)' EVAL              ; calculate square root of determinant
```

```
      '2*A' EVAL                          ; calculate 2*A
      → S TWOA                            ; assign the square root of D to local variable S
                                          ; assign 2*A to the local variable TWOA
            «
            '(-B+S)/TWOA' EVAL            ; obtain first root
            '(-B-S)/TWOA' EVAL            ; obtain second root
            »
      »
»
```

Reducing Program Levels

Nested program levels occur when local variables are declared later in the program. It is easier for some programmers to list all of the local variables at the beginning of the program. Local variables that store intermediate results are initilized with dummy data. This method reduces the program levels to two. Applying this technique to the above program perform the following:

1) Push two dummy values (usually zeros) in the stack -- one for S and the other for TWOA.
2) Include the variables S and TWOA in the first list of local variables.
3) Use the 'S' STO and 'TWO' STO to store the meaningful data in the S and TWO variables, respectively.

> *Programming Notes*
>
> The number of program levels can be reduced by declaring all or most of the local variables at the beginning of the program. Local variables storing intermediate results can be assigned dummy values and included in the first list of local variables.

The modified version of the quadratic solver is shown below:

```
; Listing 3.5.  Quadratic Solver version 3.3
«
0 0                                       ; push dummy values in the stack
→ A B C S TWOA                            ; store data in local variables A, B, and
                                          ; C.  Also store dummy zeros in local
                                          ; variables S and TWOA.
      «
      '√(SQ(B)-4*A*C)' EVAL               ; calculate square root of determinant
      'S' STO                             ; assign the square root of D to local variable S
      '2*A' EVAL                          ; calculate 2 A
      'TWOA' STO                          ; assign 2*A to the local variable TWOA
      '(-B+S)/TWOA' EVAL                  ; obtain first root
      '(-B-S)/TWOA' EVAL                  ; obtain second root
      »
»
```

Notice that the names of the variables S and TWO are enclosed in the tick characters. If these

characters are removed, RPL evaluates the variables instead of pushing their names in the stack. This results in a runtime error due to attempting to store one number in another.

Accessing Global Variables

In most of the programs shown so far only local variables were used. This should not give you the impression that programs should never interact with global variables. The intended point stresses that local variables are more recommended than their global counterparts. However, there are applications that must use global variables to store data between program calls. The following simple program implements a random number generator. The seed is stored in the global variable SEED. Whenever the program runs, it recalls the seed value from the global variable SEED, calculates a new random number, and stores the result back in SEED. A copy of the new random number is pushed into the stack for convenience. To run the next program you must first create the global variable SEED and assign it a real number.

```
; Listing 3.6.  Random number generator.
«
  'SEED' EVAL                          ; recall the seed value from the global variable SEED
  π →NUM +                             ; add pi to the old seed value
  3 ^                                  ; cube the result
  FP                                   ; retain the fractional part
  DUP                                  ; duplicate new random number in stack
  'SEED' STO                           ; store a copy back in SEED
»
```

Using the above program you can reseed the random number generator by manually storing a new real number in the global variable SEED. Programs can also use both local and global variables -- local variables are used for short term data storage, while global variables are employed for long term data storage.

Programming Note

Global variables are used to:

- Storing data between program calls.
- Accessing data representing parameters specific to the current state of an HP48SX.

Calling Other Programs

RPL permits programs to call other programs as subroutines. If you have programmed on the HP41C or used BASIC, this is not a new feature. The HP41C XEQ command empowered programs to invoke other programs. Similarly, the BASIC GOSUB and CALL statements enable internal or external subroutine calls. This feature fosters a structured approach for programming on the HP48SX, as well as all other machines. RPL and its cousin language

FORTH enable you to develop a hierarchy of highly independent program objects. This process involves a cycle of program development and testing.

I will illustrate this feature using the following program. Here, a dice is simulated, with the output to the stack ranging from 1 to 6. The dice program calls the random number generator program, presented above. This program assumes that the random number generator is stored in the variable RNG.

```
; Listing 3.7.  Dice simulator.
«
  RNG                                   ; call the random number generator program
  6 *                                   ; multiply by 6.  The result is 0 ≤ result < 6
  IP                                    ; truncate the number in level 1
  1 +                                   ; add one to make the final result in the range of 1 to 6
»
```

While the above example is a simple one it shows the following important point: the RNG program is independent of the dice program. In fact, the RNG program can be used in many other programs that generate a wide variety of random numbers (e.g., numbers that are normally, log-normally, and linearly distributed). Callable programs *should* be independent of their client programs. This may not always be the case. You may find or write callable programs that strictly service multiple client programs. This type of callable program participates in implementing structured programs.

Debugging Programs

Debugging is a form of executing a program in slow motion. This allows the programmer to trace each program step being executed and watch the values of the variables and data. Hewlett-Packard has implemented single-step debugging in all of its programmable pocket calculators. The HP48SX is no exception. RPL allows you to debug a program using the following steps:

- Enter any data the program expects to find in the stack. You might need to enter a particular set of values that are causing the program to malfunction.
- Push the name of the variable storing the program in level 1.
- Press the [PRG] key and select the CTRL menu option.
- Select the DBUG option. This puts the HP48SX in debug mode. The name of the debugged program is popped off the stack.
- Use the SST or SST↓ option to single-step through the program. The SST option causes subprogram calls to be executed at full speed. By contrast, the SST↓ option empowers you to single-step through the execution of a subprogram. Use the SST option when you do not suspect a subprogram to contain the bug.

The HP48SX offers other debugging options. They are:

- The NEXT option permits you to look at the next program step. This is handy if you want to select either the SST or SST↓ option.
- The KILL option takes the HP48SX resets the debugging mode. This option is useful if you

want to stop a debugging session and start a new one.

Next, I present two debugging examples. The first example single-steps through the first version of the quadratic equation solver. The program, assumed to be stored in the variable QDR, is shown again next:

```
; Listing 3.8.  Quadratic Solver version 3.4.
«
→ A B C                                 ; store data in local variables A, B, and C
      «
      B SQ                              ; calculate B squared
      A C 4 * *                         ; calculate 4 A C
      -                                 ; obtain the determinant
      √                                 ; get the square root of D
      A 2 *
      → S TWOA                          ; assign the square root of D to local variable S
                                        ; assign 2*A to the local variable TWOA
            «
            B NEG                       ; calculate the first root
            S +
            TWOA /
            B NEG                       ; calculate the second root
            S -
            TWOA /
            »
      »
»
```

Enter the values 1, -5, 6, and 'QDR' in the stack. Press the [PRG] key and the CTRL and DBUG menu options. The name 'QDR' is popped of the HALT annunciator is displayed at the top of the HP48SX screen. The single-step tracing (using SST) is shown in the table on the next page:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	→ A B C	none	Assignment to local variables
2	B	-5	recall value in B
3	SQ	25	square the number in level 1
4	A	1	recall value in A
5	C	6	recall value in C
6	4	4	push 4 in stack
7	*	24	multiply 4 and C
8	*	24	obtain 4*A*C
9	-	1	obtain determinant
10	√	1	get the square root
11	A	1	recall value in A
12	2	2	push 2 in stack
13	*	2	obtain 2*A
14	→ S TWOA	2	assign results to more local variables
15	B	-5	recall value in B
16	NEG	5	change sign
17	S	1	recall value in S
18	+	6	add -B and S
19	TWOA	2	recall value in TWOA
20	B	-5	recall value in B
21	NEG	5	change sign
22	S	1	recall value in S
23	-	6	subtract -B and S
24	TWOA	2	recall value in TWOA
25	/	3	obtain second root

Since the above program does not call any subprogram you may employ either the SST or SST↓ option.

In the second debugging example we trace the execution of the dice program (saved under the name DICE). Create the global variable SEED, if you already have not. Store 5 in SEED. Enter the name 'DICE' in the stack. Press the {PRG} key and select the CTRL and DBUG options. Trace the program execution using the SST↓ option to single-step through the RNG subprogram. The single-step tracing is shown in the following table:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	'SEED'	'SEED'	the name 'SEED' is pushed in the stack
2	EVAL	5	evaluate 'SEED'
3	π	'π'	push 'π' in the stack
4	→NUM	3.14159265359	evaluate 'π'
5	+	8.14159265359	add SEED and pi
6	3	3	push 3 in the stack
7	^	539.669791716	raise to power 3
8	FP	0.669791716	keep fractional part
9	DUP	0.669791716	duplicate level 1
10	'SEED'	'SEED'	push 'SEED' in stack
11	STO	0.669791716	store level 1 in SEED
12	»	0.669791716	exit from subprogram RNG
13	6	6	push 6 in the stack
14	*	4.018750296	multiply levels 1 and 2
15	IP	4	truncate real number
16	1	1	push 1 in the stack
17	+	5	add 4 and 1
18	»	5	end program DICE

Programming Tip

RPN code is easier to debug than algebraic objects.

The longer the algebraic objects the more difficult they are to debug.

Program Manipulation of Directories

Earlier in this chapter we examined creating, removing, and moving between directories. The instructions discussed were for manual action. What about directory manipulation in programs? The answer revolves around the fact that when you create a directory, RPL creates a variable with the same name. As with all other variables, directory variables are inherited by subdirectories. Thus, if a program pushes the name of a directory variable (i.e., the name of a directory) on the stack and then executes an EVAL command, RPL attempts to evaluate that variable. A successful evaluation leads to a movement to that directory. This results in a very interesting feature, summarized by the following rules:

1) You can move between any two directories by evaluating a sequence of directory variables, starting with HOME. This ensures that you always start with the root directory and proceed to the target directory. The following program, named TDIR (which should be located in the HOME directory) performs this task:

```
; Listing 3.9.  Program TDIR
; Author: Namir Clement Shammas
; Version 1.0, created 8/15/90
; Purpose: perform a directory move
«
DUP                                     ; duplicate list
SIZE                                    ; obtain the size of the list
→ L N                                   ; assign list to local variable L
                                        ; assign list size to local variable N
      «
      1 N                               ; set limits for the FOR-NEXT loop
      FOR I                             ; start FOR loop.  I is the control variable
            L                           ; push the list in the stack
            I                           ; push the loop variable in the stack
            GET                         ; obtain the I'th list member (i.e. directory name)
            EVAL                        ; evaluate the I'th member
      NEXT
      »
»
```

The input to the above program is a list of a complete path. Consider the directory in Figure 3.1 on page 28. If you wanted to move from any directory to the NLR directory, you need to supply the above program with the following argument:

{ HOME MATH OPTM NLR }

The program extracts the members of the supplied path and evaluates each one. Each evaluation moves to that directory. The sequence of evaluation takes you to your target directory.

2) You can move to a child subdirectory by simply evaluating its name. For example, if a program is currently in the HOME directory (see Figure 3.1) and needs to move down to STRING, it pushes the name STRING in the stack and evaluates it. The TDIR program can be used. The input is a single member list, { STRING } in this case.

3) You can migrate upward along the same path by simply evaluating the name of the target ancestor directory. For example, if a program is located in directory NLR and needs to move to MATH, it simply evaluates the name 'MATH'. There is no need to mention OPTM, the parent directory of NLR! This neat feature exists because of the inheritance in variables (including directory variables) along a directory path. In the case of our example, the directory variable MATH is visible in the directory NLR. Evaluating the directory name MATH successfully leads to the MATH directory. The above program can also be used with this type of directory traversal. A single-member list is needed. The input for our example is { MATH }. Notice that the same input is needed to move from OPTM to MATH (although using UPDIR is quicker).

Programs Manipulating Programs

There are two basic types of languages and programs, namely, compilers and interpreters. Compilers transform the readable source text into machine instructions. You need not compile a program again to run it, unless you have changed it. When a compiled program runs, the machine instructions are executed by the machine's Central Processing Unit (CPU). By contrast, interpreters execute a program by interpreting few parts of the program at a time. Once, these parts are executed, the interpreter forgets about them. Consequently, if the interpreter re-encounters program statements, it interprets them again! This is why interpreted programs are slow. There are also language implementations that partially compile programs. The compilers are called p-code (or psuedo-code) compilers. Such languages use a wide variety of approaches. One compiles a program into instructions of an abstract CPU. During program execution, the abstract instructions are translated into the instructions of the actual CPU.

One of the interesting advantages of using interpreters is dynamic program modification. This is based on the fact that interpreted programs are regarded as data. As data, these programs can be modified. One aspect of such modification is for a program to modify itself while it is running. The interpreted BASICA and GW-BASIC possess such a capability. Another aspect of program modification, applicable to RPL, is for a program object to modify another program object before calling it. This approach is slightly different from the first one, since RPL applications often consist of a set of semi-independent and modular program objects.

The basic steps in performing such program modifications are:

1) The edited program object is pushed in the stack and converted into a string.

2) The string image of the edited program is manipulated (this includes adding, removing, and translating text) accordingly.
3) The string is converted back into the program object.
4) The altered program object is then stored in its proper host variable.
5) The modified program can be called, reflecting the runtime changes.

The following figure compares RPL and IBM-PC commands for traversing directory trees. Keep in mind that the root of IBM-PC disks is simply the backslash character, \, while the root of the HP48SX is HOME. The directories of figure 3.1 on page 28 are used.

RPL	BASIC	Comments
HOME	CHDIR("\")	move to the root directory
UPDIR	CHDIR("..")	move to the parent directory
'MATH' EVAL	CHDIR("\MATH")	move from OPTM to MATH
'OPTM' EVAL	CHDIR("\MATH\OPTM")	move from NLR to OPTM
'WORD' EVAL	CHDIR("WORD")	move from STRING to WORD
'STRING' EVAL		
'WORD' EVAL	CHDIR("\STRING\WORD")	move from HOME to WORD

Figure 3.3. Comparing the movement in the directory tree between RPL and IBM-PC BASIC.

Program Guidelines

Good programming practices are built on sound discipline. While there is not one single method that must be followed, a few techniques have evolved. Here are some guidelines for good programming practices:

- Plan your program ahead. The more complex the application, the more planning is needed. Design the features and determine the functionality of the application.
- Determine the primary data objects that are used in your application.
- Develop an outline of the set of tasks required. This process is a cyclic one. Each cycle defines the tasks at a specific level. The first cycle determines the major tasks required. The second cycle defines the tasks required by the first-level tasks. Each successive cycle brings more details on what needs to be done. You will reach a stage where the tasks are so well defined that the next natural step is to write the RPL code. Computer scientists call this method a top-down design.
- Implement the code using a bottom-up strategy. Since you have all the tasks defined, start writing the RPL source code for the lowest level programs. These routines should be coded before moving on the next level of programs, as guided by your outline. This process is repeated until you have coded the complete application.
- Document your programs as if others will read it. You will be surprised how vague undocumented programs will look six months or so after they are written. Give extra

explanation to programming tricks.

- Include in your document the following:
    - Program name.
    - Your name.
    - The version number.
    - The date of creating the program.
    - The date for the last update.
    - Program purpose.
    - Comments.
    - A list of global variables.
    - A list of local variables.
    - A list of called subprograms.

- Comment your documented programs. I use the semicolon to separate trailing comments.
- Adopt a program indentation style. There is no concrete rule for indenting documented RPL listings. I use tabs to indent each logical level. This includes nested program objects and the statements of a loop or decision-making constructs (discussed in later chapters).
- Use meaningful names for local and global variables. Be consistent in using the same names. For example, the variable LEN should consistently mean the length of an array, list, or string. This rule proves to be even more valuable when using single letter names.
- Use bold characters when documenting subroutine calls. Use italic characters when documenting global variables. This makes your listing more readable. Whatever style you select, be consistent.
- Use lowercase variable names for local variables.
- Test your programs with several sets of data. Often you employ a single set of data that works fine with a program. Test boundary and median values. Consider the case of testing a program that inserts a small string in a big string. The median value is the middle of the string. The boundaries are the head and tail of the string. Test the following cases:
    - Inserting the small string before the first character of the big string.
    - Inserting the small string after the last character of the big string.

  Boundary-value cases frequently require additional code to detect them and properly deal with them.

4

Interactive Input and Output

We will explore interactive keyboard input and output in this chapter. The popular term *user-friendly software* is a reminder that (a) the user is supposed to know what to do, and (b) the user never makes mistakes. Most end-users welcome programs that guide them with messages, menus, and other fool-proof measures. This chapter discusses techniques that handle input prompting, menu selection, sound, and labeled output.

Some Prompts Never Die!

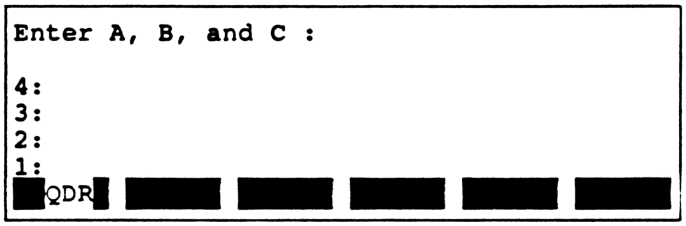
The HP41C PROMPT command is also available in the HP48SX. It offers a simple method (compared to the INPUT command, discussed later) for prompting the user for input. The HP48SX PROMPT command takes a string in level 1 and displays it in the upper left corner of the screen. The program control returns to the keyboard. This means that you are free to key in one or more data objects and even perform intermediate calculations! When you are good and ready for the program to resume, invoke the CONT command (by pressing the [left shift][ON] keys). The following program modifies a version of the quadratic solver from Chapter 3 by adding a PROMPT command. The latter displays the following string at the upper left screen corner:

Enter A, B, and C :

To comply with the request, enter three real numbers and invoke the CONT command to resume program execution.

```
; Listing 4.1.  Quadratic Solver version 4.1
«
"Enter A, B, and C : " PROMPT          ; prompt for input
0 0                                    ; push dummy values in the stack
→ A B C S TWOA                         ; store data in local variables A, B, and
                                       ; C.  Also store dummy zeros in local
                                       ; variables S and TWOA.
      «
      '√(SQ(B)-4*A*C)' EVAL            ; calculate square root of determinant
      'S' STO                          ; assign the square root of D to local variable S
      '2*A' EVAL                       ; calculate 2 A
      'TWOA' STO                       ; assign 2*A to the local variable TWOA
      '(-B+S)/TWOA' EVAL               ; obtain first root
      '(-B-S)/TWOA' EVAL               ; obtain second root
      »
»
```

The display during the program looks like:



Labeling the Output

If you run the above program you might complain of the ambiguity of the output. While the program prompts you for the input, the output consists of two real numbers with no explanation of what they are. The HP48SX offers at least two methods to label the program output. The first uses tags attached to stack objects. For example, the tagged output of the above program might look like this:

```
2: ROOT1:2.5
1: ROOT2:4.5
```

The second technique is to simply build a string that contains the output and its explanation. Thus, the output of the above program might look as follows:

```
2: "ROOT1 = 2.5"
1: "ROOT2 = 3.4"
```

What's the difference between the two methods? Tags are very easy to attach and remove. Consequently, the tagged results can be further used by the program with practically no additional manipulation. Using strings has the advantage of giving you better wording. The down side is that if the results need to be used later, they must be extracted from the string. A program may store the results in variables and concatenate copies to the output strings. This way, the results are easily retrieved from the host variables.

To tag an object in level 1 of the stack, you push the tag name and invoke the →TAG command. The following listing shows the last version of the quadratic solver modified to include tags with the output:

```
; Listing 4.2.  Quadratic Solver version 4.2
«
"Enter A, B, and C" PROMPT              ; prompt for input
0 0                                     ; push dummy values in the stack
→ A B C S TWOA                          ; store data in local variables A, B, and
                                        ; C.  Also store dummy zeros in local
                                        ; variables S and TWOA.
      «
```

```
        '√(SQ(B)-4*A*C)' EVAL          ; calculate square root of determinant
        'S' STO                        ; assign the square root of D to local variable S
        '2*A' EVAL                     ; calculate 2 A
        'TWOA' STO                     ; assign 2*A to the local variable TWOA
        '(-B+S)/TWOA' EVAL             ; obtain first root
        'ROOT1' →TAG                   ; attach a ROOT1 tag to the first root
        '(-B-S)/TWOA' EVAL             ; obtain second root
        'ROOT2' →TAG                   ; attach a ROOT2 tag to the second root
        »
»
```

The next version of the quadratic solver uses strings to display the results. I also wrote the program to store the results in the global variables ROOT1 and ROOT2. The program is shown below:

```
; Listing 4.3.  Quadratic Solver version 4.3
«
"Enter A, B, and C : " PROMPT          ; prompt for input
0 0                                    ; push dummy values in the stack
→ A B C S TWOA                         ; store data in local variables A, B, and
                                       ; C.  Also store dummy zeros in local
                                       ; variables S and TWOA.
        «
        '√(SQ(B)-4*A*C)' EVAL          ; calculate square root of determinant
        'S' STO                        ; assign the square root of D to local variable S
        '2*A' EVAL                     ; calculate 2 A
        'TWOA' STO                     ; assign 2*A to the local variable TWOA
        "ROOT1 = "                     ; push the first string tag in the stack
        '(-B+S)/TWOA' EVAL             ; obtain first root
        DUP                            ; duplicate the above result
        'ROOT1' STO                    ; store in global variable ROOT1
        +                              ; concatenate the string tag and the first root
        "ROOT2 = "                     ; push the second string tag in the stack
        '(-B-S)/TWOA' EVAL             ; obtain second root
        DUP                            ; duplicate the above result
        'ROOT2' STO                    ; store result in global variable ROOT2
        +                              ; concatenate the string tag and the second
                                       ; root
        »
»
```

The INPUT Command

The HP48SX offers a second interactive input command besides PROMPT. The INPUT statement, perhaps intentionally named after BASIC's INPUT statement, offers more power. The INPUT command permits you to display two strings: a prompt string located where level 4 appears, and a string in the command line. The latter may be used to supply default input and more sophisticated prompting. When the INPUT command is invoked it takes the machine into command-line editing mode. You can edit the command line. Pressing the [Enter] key resumes program execution.

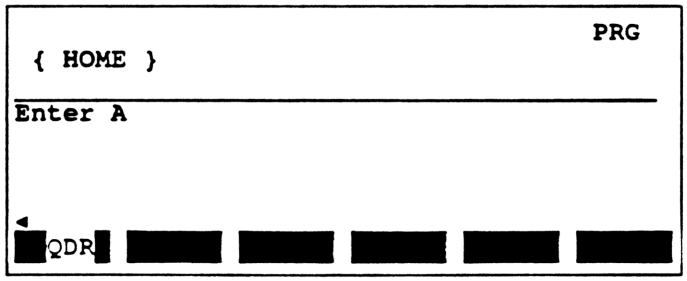
Simple Input

Ley's use a series of short examples to explain the rather un-obvious way of using INPUT. The first example modifies version 4.2 of the quadratic solver by replacing PROMPT with INPUT. Instead of having a single prompt for three data, you get a separate prompt for each input. This

makes the program more friendly to users who are not comfortable with RPN and stacks. The three INPUT statements are similar. Each takes two string arguments from the stack. The first string is the prompt that appears at the upper left screen. The second string is placed in the command line. Since the second string is empty, the command line appears empty when INPUT is invoked. The OBJ→ command transforms the input into the appropriate object type, real numbers in our case.

```
; Listing 4.4.  Quadratic Solver version 4.4
«
"Enter A" "" INPUT OBJ→                 ; prompt for A
"Enter B" "" INPUT OBJ→                 ; prompt for B
"Enter C" "" INPUT OBJ→                 ; prompt for C
0 0                                     ; push dummy values in the stack
→ A B C S TWOA                          ; store data in local variables A, B, and
                                        ; C.  Also store dummy zeros in local
                                        ; variables S and TWOA.
      «
      '√(SQ(B)-4*A*C)' EVAL             ; calculate square root of determinant
      'S' STO                           ; assign the square root of D to local variable S
      '2*A' EVAL                        ; calculate 2 A
      'TWOA' STO                        ; assign 2*A to the local variable TWOA
      '(-B+S)/TWOA' EVAL                ; obtain first root
      'ROOT1' →TAG                      ; attach a ROOT1 tag to the first root
      '(-B-S)/TWOA' EVAL                ; obtain second root
      'ROOT2' →TAG                      ; attach a ROOT2 tag to the second root
      »
»
```

The screen is shown below when prompting for the first coefficient:



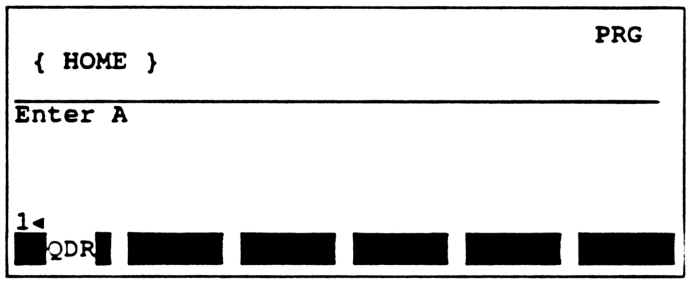
Using a Default Input

The next version of the quadratic solver uses default input data for each coefficient. Thus, the second INPUT string is no longer empty. Instead, it contains a default value. When you run version 4.5 the command line contains a default value. If you press [Enter] the default value is used.

```
; Listing 4.5.  Quadratic Solver version 4.5
«
"Enter A" "1" INPUT OBJ→                ; prompt for A with a default value of 1
"Enter B" "-5" INPUT OBJ→               ; prompt for B with a default value of -5
"Enter C" "6" INPUT OBJ→                ; prompt for C with a default value of 6
0 0                                     ; push dummy values in the stack
```

```
→ A B C S TWOA                          ; store data in local variables A, B, and
                                        ; C.  Also store dummy zeros in local
                                        ; variables S and TWOA.
      «
      '√(SQ(B)-4*A*C)' EVAL             ; calculate square root of determinant
      'S' STO                           ; assign the square root of D to local variable S
      '2*A' EVAL                        ; calculate 2 A
      'TWOA' STO                        ; assign 2*A to the local variable TWOA
      '(-B+S)/TWOA' EVAL                ; obtain first root
      'ROOT1' →TAG                      ; attach a ROOT1 tag to the first root
      '(-B-S)/TWOA' EVAL                ; obtain second root
      'ROOT2' →TAG                      ; attach a ROOT2 tag to the second root
      »
»
```

The screen is shown below when prompting for the first coefficient:



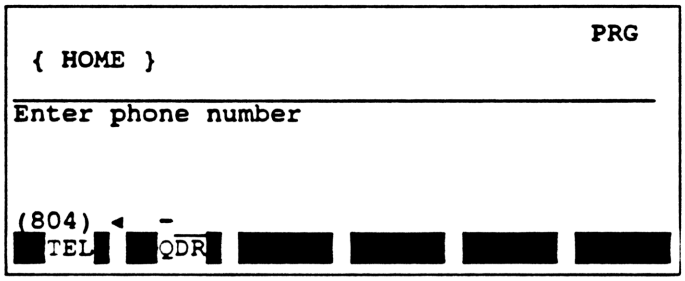
To avoid using the default value you must delete it. It will not automatically disappear if you press an alphanumeric key when prompted. Many microcomputers applications use the latter technique, which is superior to the HP48SX approach.

Manipulating the Default Input

The last program illustrated how a default input is supplied. I purposely chose a simple default value. The INPUT command allows you to position the cursor anywhere inside the default value and set insert or overwrite mode. This is shown by the next program that prompts you for your phone number. The prompt string for the INPUT statement is "Enter phone number". The second argument for INPUT is a list that contains a string and a real number. The string "(804) ___-____" displays a phone number template with a default area code. The number -7 is interpreted as "put the cursor 7 characters from the left side of the string." The negative value signals that the cursor is in overwrite mode. Had I used the positive number 7, the cursor would be in insert mode. For this kind of prompt, the insert mode is the appropriate mode. When the INPUT statement is executed, the solid block cursor (indicating that you are in overwrite mode) is located at the first underscore character. You may move the cursor to the area code data and type in another value. The underscore characters remind you of where you need to type non-default input. After you press the [Enter] key, the program extracts the area code and phone numbers, tags them, and leaves them in the stack. The listing is shown below:

```
; Listing 4.6.  Program to demonstrate default input.
«
  "Enter phone number"
  { "(804) ___-____" -7} INPUT          ; prompt for the phone number
  DUP                                   ; duplicate the input
  2 4 SUB                               ; extract the area code
  'AreaCode' →TAG                       ; tag a label to area code string
  SWAP                                  ; swap to access other input copy
  7 14 SUB                              ; extract phone number
 'Number' →TAG                          ; tag a label to phone number string
»
```

The screen is shown below when prompting for the phone number:



The rules for using this type of advanced prompting are:

- The argument for INPUT in level 1 is a list made up of a string and a real number.
- The string appears in the command line. Quoted string constants may appear in the list. However, variables containing strings must be first evaluated and the result included in the list.
- The absolute value of the real number represents the position of the cursor from the left side of the string. A value of 3 puts the cursor at the third string character. A zero indicates that the cursor is placed after the command-line string. The absolute value of the real number is rounded up. Thus, 7.6 is rounded up to 8, while 7.4 is rounded down to 7.
- The sign of the real number indicates the cursor mode. Positive values put the cursor in insert mode (the cursor appears as a left-pointing arrow). By contrast, negative values invoke the overwrite mode (the cursor appears as a block).

Programming Note

The absolute values of the real numbers are located. For example, -7.6 is rounded to the number -8.

Tag-Aided Input

Version 4.2 of the quadratic solver uses the PROMPT command to prompt you for all three coefficients. The last two versions of the quadratic solver use the INPUT command to individually prompt for each coefficient. You can use the INPUT command to prompt for several items at once. Tags can be used to assist in the input multiple items. The command line string contains the sequence of needed tags (one for each input datum). To display each tag on a separate line, each tag must be followed by a return character. What about the cursor? Where should it be located? The INPUT command allows you to select the row and column position of the cursor using a list of two reals (I will call this list the *input control list*) . The first real number is the tag index. Thus, 1 locates the cursor in the line of the first tag that appears in the command-line string. The number 3 positions the cursor at the second tag in the command-line string, and so on. If the tag index is greater than the actual number of tags, the cursor is located at the last tag. In addition, the sign of the tag index number selects the insert or overwrite mode. Positive values set the insert mode, while negative values select the overwrite mode.

The second real number in the list specifies the number of characters from the left side of a tag. Thus, { 2 1 } indicates that the cursor is located at the first character of the second tag. Assigning 0 to the character position causes the cursor to be located after the tag. Thus, { -4 0 } locates the cursor after the fourth tag and selects the overwrite mode.

> *Programming Note*
>
> The list { n m } used with the INPUT command places the cursor at the m'th character of the n'th line. If n is zero or greater than the actual number of lines, the cursor is located at the last line. If m is zero, or greater than the length of the n'th line, the cursor is located after the line's text.
>
> The HP48SX screen shows up to four tags at a time. When using more than four tags, you scroll through them using the up and down arrow keys.
>
> Prompting with more than three tagged items overwrites the prompt string.

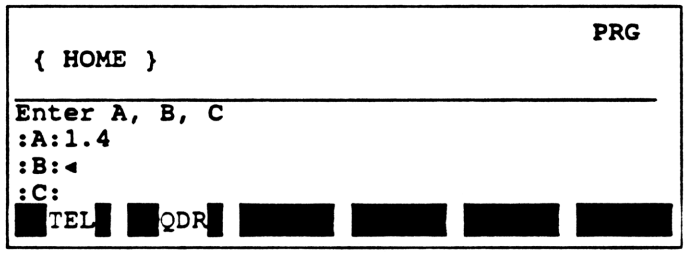
The next program shows a version of the quadratic solver that uses the INPUT command as discussed above (note: the symbol ■ in the listing is a carriage return):

```
; Listing 4.7.  Quadratic Solver version 4.6
«
"Enter A, B, C"                          ; display prompt string
{ ":A:■:B:■:C:" { 1 0 } }                ; set command-line string and cursor mode and
                                         ; location
INPUT OBJ→                               ; invoke INPUT command
```

```
0 0                                    ; push dummy values in the stack
→ A B C S TWOA                         ; store data in local variables A, B, and
                                       ; C.  Also store dummy zeros in local
                                       ; variables S and TWOA.
      «
      '√(SQ(B)-4*A*C)' EVAL            ; calculate square root of determinant
      'S' STO                          ; assign the square root of D to local variable S
      '2*A' EVAL                       ; calculate 2 A
      'TWOA' STO                       ; assign 2*A to the local variable TWOA
      '(-B+S)/TWOA' EVAL               ; obtain first root
      'ROOT1' →TAG                     ; attach a ROOT1 tag to the first root
      '(-B-S)/TWOA' EVAL               ; obtain second root
      'ROOT2' →TAG                     ; attach a ROOT2 tag to the second root
      »
»
```

The screen image of a sample session is shown below. The image shows the following:

- The tags A, B, and C.
- The number 1.4 was entered for the tag A.
- The cursor is currently located after the tag B.
- The tag C has yet to receive its input.



To move up or down between tags use the up and down arrow keys. Using these keys, you are free to move back and forth between the tagged data. You can also edit your input (and any default values) very easily.

Input Validation

The INPUT command offers a basic input validation feature. By including an unquoted V *parameter* in the input control list, your programs can detect input that does not correspond to any valid object type. Let's consider the last program. If you enter an ill-formatted number, say 1..4, the two-decimal number is not checked by INPUT. Consequently, the above program is halted later due to the attempt to handle the erroneous input. There are two cures: insert a set of commands that hunt for bad input, or use the V parameter. This parameter causes the INPUT command to make sure that the input after each tag corresponds to a valid object type. This is by no means a fool-proof measure, since, for example, you can enter a matrix where a complex number is expected. More effective validation is discussed in the chapter on error-handing. Thus, to utilize INPUT's data validation you need to change the following list:

```
{ ":A:■:B:■:C:" { 1 0 } }
```

into

```
{ ":A:■:B:■:C:" { 1 0 } V }
```

Programming Note

The members of the input control list can appear in any order.

Other Input Control Parameters

The HP48SX offers two more input control parameters. They are the ALG and the α parameters which must be appear unquoted in the input control list. The ALG parameter puts the HP48SX in algebraic-object mode. The α parameter locks the alpha input. This parameter is very useful for string input.

Controlling the Screen Output

The HP48SX offers a set of commands that permit you to control the screen output. The CLLCD command clears the LCD display of the machine. Once the screen is blanked, the DISP command is able to write to the screen using medium-size characters. The DISP command treats the LCD screen as having 7 rows and 22 columns. The rows are numbered 1 though 7, with one being the top screen row. The stack is redisplayed when the program ends or when a HALT command is executed. The following program uses the CLLCD and DISP commands to demonstrate these commands. The output displays strings "ROW # 1" through "ROW #7".

```
; Listing 4.8.  DISP command demo, version 4.1
«
  CLLCD                                  ; clear LCD
  "ROW # 1" 1 DISP                       ; display strings on rows 1 to 7
  "ROW # 2" 2 DISP
  "ROW # 3" 3 DISP
  "ROW # 4" 4 DISP
  "ROW # 5" 5 DISP
  "ROW # 6" 6 DISP
  "ROW # 7" 7 DISP
  5 WAIT                                 ; wait 5 seconds
»
```

The DISP command converts the usual LCD display into a 7-by-22 character message board. This permits the implementation of on-line help and condensed output.

The above program introduces the WAIT command. It suspends program execution for a specified number of seconds. The waiting period is taken from level 1. The above program

pauses for five seconds after displaying the seven strings, before the normal stack reappears.

While pausing a program for a specified period has its appeal, it also has drawbacks. What if the user was distracted during the pause? The remedy is to make the program pause until a the user presses a key. The WAIT command accommodates this requirement, too. By supplying an argument of zero, the WAIT command pauses the program until a key is pressed (the alpha and shift keys are excluded). It returns the code for the pressed key. If this information is of no value to the program, simply pop it off the stack. I have modified the last program to include the WAIT command with a zero argument. The seven strings remain visible until you press a key.

```
; Listing 4.9. DISP command demo, version 4.2
«
  CLLCD                                  ; clear LCD
  "ROW # 1" 1 DISP                       ; display strings on rows 1 to 7
  "ROW # 2" 2 DISP
  "ROW # 3" 3 DISP
  "ROW # 4" 4 DISP
  "ROW # 5" 5 DISP
  "ROW # 6" 6 DISP
  "ROW # 7" 7 DISP
  0 WAIT                                 ; wait until a key is pressed
  DROP                                   ; drop the keystroke code generated by the 0 WAIT command
»
```

The HP48SX offers the KEY command that works like 0 WAIT. The difference is that when KEY is invoked it returns 0 if no key was pressed; otherwise, the command returns the key code. The most common use of the KEY function includes the conditional DO-UNTIL loop (more about loops in Chapter 7). For now, don't worry about the loop if it looks ambiguous. The new program version is shown below:

```
; Listing 4.10.  DISP command demo, version 4.3
«
  CLLCD                                  ; clear LCD
  "ROW # 1" 1 DISP                       ; display strings on rows 1 to 7
  "ROW # 2" 2 DISP
  "ROW # 3" 3 DISP
  "ROW # 4" 4 DISP
  "ROW # 5" 5 DISP
  "ROW # 6" 6 DISP
  "ROW # 7" 7 DISP
  DO                                     ; use an empty DO-UNTIL loop
  UNTIL KEY END                          ; loop until a key is pressed
  DROP                                   ; drop the keystroke code generated by the KEY command
»
```

The HP48SX Bells and Whistles

The HP41C offered the BEEP and TONE commands to add sound to programs. Most BASIC implementations also have sound commands. The HP48SX offers the BEEP command that allows the data in the stack to specify the frequency and duration of the sound. The functionality of the HP48SX BEEP resembles the BEEP command in many BASIC

implementations. The argument in level 2 is the tone frequency. The argument in level 1 is the tone duration in seconds. The following version of the DISP command demo program beeps for 0.2 seconds after a key is pressed:

```
; Listing 4.11.  DISP command demo, version 4.4
«
  CLLCD                                   ; clear LCD
  "ROW # 1" 1 DISP                        ; display strings on rows 1 to 7
  "ROW # 2" 2 DISP
  "ROW # 3" 3 DISP
  "ROW # 4" 4 DISP
  "ROW # 5" 5 DISP
  "ROW # 6" 6 DISP
  "ROW # 7" 7 DISP
  0 WAIT                                  ; wait until a key is pressed
  DROP                                    ; drop the keystroke code generated by the 0 WAIT command
  1000 .2 BEEP                            ; toot the 48's horn!
»
```

Using Menus for Input

The latest generation of Hewlett-Packard hand-held programmable calculators extensively uses menus. In addition to a wealth of predefined menus, the HP48SX allows you to set up your own custom menu. This menu may be used to collect frequently used commands and functions. The custom menu can also be used for a menu-directed input.

Building Custom Menus: A Crash Course

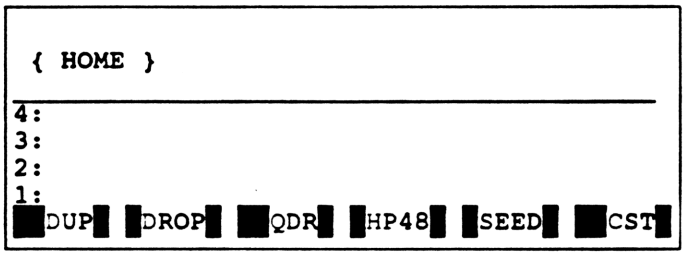
If you are familiar with creating custom menus, you may skip this subsection. The [CST] key invokes the custom menu of the HP48SX. Interestingly, there are variations in the process of creating custom menus. This is a quick rundown for these methods.

To create a custom menu you need to place a list of menu items in level 1 and invoke the MENU command (by pressing the [right shift][CST] keys and selecting the MENU option). The type of list members determine how sophisticated the custom menu is.

The simplest way to define a custom menu is to include a list of commands and other objects that appear verbatim in the custom menu. For example, consider the following list used with a MENU command:

```
{ DUP DROP QDR "HP48SX" SEED CST } MENU
```

The custom menu that appears when pressing the [CST] key is shown below:



The first two options are predefined commands. The third option invokes the quadratic solver program saved as the variable QDR. The fourth menu option pushes the string "HP48SX" in the stack. The fifth option recalls the contents of the global variable SEED (if it exists) or pushes the name 'SEED' in the stack. The last option is similar to the last one, except it works with the global variable CST. This variable is created by the machine when the MENU command was executed. The variable CST contains the list used to create the most recent custom menu. By storing the menu list in the CST variable, the HP48SX is able to offer you a separate custom menu for each directory! Moreover, if you have selected the custom menu and move to another directory, the HP48SX automatically displays the custom menu of the new directory.

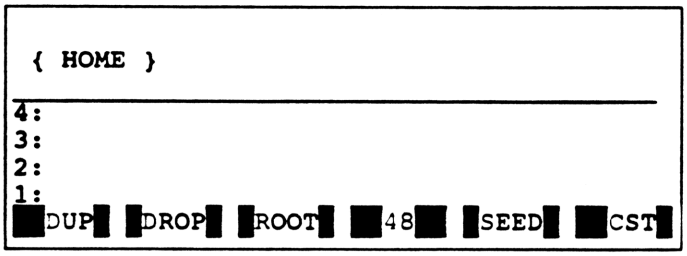
Programming Note

By storing the menu list in the CST variable, the HP48SX is able to offer you a separate custom menu for each directory! Moreover, if you have selected the custom menu and move to another directory, the HP48SX automatically displays the custom menu of the new directory.

The second genre of custom menus allows the menu options to be aliased using labels. This means that the menu displays a label name that is different from the actual name of the evaluated object. The menu labels are strings. Each menu option and its label must be enclosed in a list. The above custom menu list can be modified to use labels as shown below:

```
{ DUP DROP { "ROOT" QDR } { "48" "HP48SX" } SEED CST } MENU
```

The label ROOT is now used instead of the name of the program QDR. The string "HP48SX" is now replaced with an abbreviated label "48". The custom menu that appears when pressing the [CST] key is shown below:



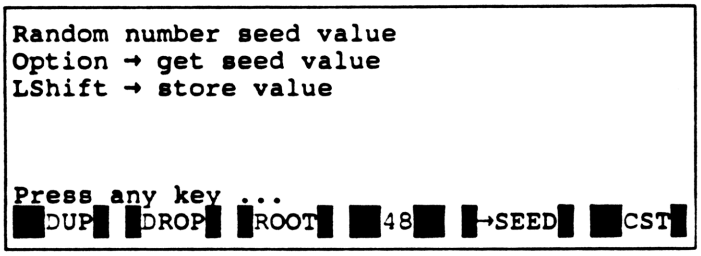
Using menu labels also makes it possible to associate a label with a program object. For example, the SEED option which recalls the contents of variable SEED can be changed to perform the reverse. A label "→SEED" is used with a program object that stores the data of level 1 in the variable SEED. Notice that the label uses the right arrow to signal that the option is used to store data into SEED. The modified custom menu list is shown below:

```
{ DUP DROP { "ROOT" QDR } { "48" "HP48SX" } { "SEED" « 'SEED' STO » CST }
```

The third type of custom menu allows you to define additional functionality with the right and left shift keys. To implement this feature a menu label must be followed by a list of three objects. The first object handles pressing the menu option. The second and third objects deal with the use of the left and right shift keys, respectively. The following list shows a new version of the custom menu list, with the →SEED option supporting the shift key functionality:

```
{
      DUP
      DROP
      { "ROOT" QDR }
      { "48" "HP48SX" }
      { "SEED"
         {
            « 'SEED' STO »
            « SEED »
            «     CLLCD
                  "Random number seed value" 1 DISP
                  "Option → get seed value" 2 DISP
                  "LShift → store value" 3 DISP
                  "Press any key ..." 7 DISP
                  0 WAIT DROP
            »
         }
      CST
} MENU
```

The three objects in the list following the "→SEED" label are program objects. The first program object stores data at level 1 in the variable SEED. This program runs when the menu option is pressed. The second program object recalls the value of the SEED variable. This takes place when the left shift key is pressed before menu key. The third object is a longer program that offers on-line instructions and help. The help screen is shown below:



Menu Input

Custom menus can be used to implement a special class of applications. These programs invoke the custom menu to display the names of the variables and the subprogram options. The custom menu becomes the focal point for entering data and running program code. Unlike the sequential input, using custom menus empower the user to selectively re-enter data. Of course, all of the data variable must be assigned values initially. After the first round of calculations, the end-user can ask *what-if* questions and update selected variables without going through the whole list. This method is even faster than a sequential input method that uses the last values you entered as default data.

To use menu-aided input, the host program must create and remove the custom menus as needed. The program section using menu-aided input needs to invoke the MENU command after defining the custom menu list. Here is a set of rules and suggested guideline for constructing the menu lists for menu-aided input:

1) Global variables are used to store and recall the data.
2) The global variables are created when the menu is constructed and are purged when the menu is no longer needed.
3) The menu labels associated with the global variables should start with the variable → symbol to signal that the menu option handles a variable (and does not execute a program object).
4) The menu option that handles a global variable should be set as follows:

- The unshifted option stores data in the variable.
- The left shift option recalls the value in the variable.
- The right shift option offers an on-line explanation.

5) The menu option that handles a program object should be set as follows:

- The unshifted option executes the program object.
- The right shift option offers an on-line explanation.

6) A menu option should be used to properly exit the program. The action taken by this option includes purging the global variables used and clearing the custom menu.

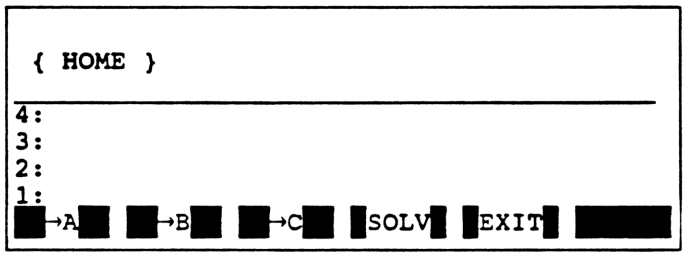
The above rules and guidelines are applied to the quadratic solver. The listing of this new

version is shown below:

```
; Listing 4.12.  Menu-aided version of the quadratic solver.
«
{
      {
      "→A"                              ; set label for coefficient A
            {
            « 'A' STO »                 ; --- unshifted menu option ---
                                        ; store value in global variable A
            « A »                       ; --- left shift menu option ---
                                        ; recall value from global variable A
            «                           ; --- right shift menu option ---
                                        ; provide with on-line help
            CLLCD                       ; clear display
            "COEFF. OF X^2" 1 DISP      ; show message in screen row 1
            0 WAIT DROP                 ; wait for a key to be pressed
            »
            }
      }
      {
      "→B"                              ; set label for coefficient B
            {
            « 'B' STO »                 ; --- unshifted menu option ---
                                        ; store value in global variable B
            « B »                       ; --- left shift menu option ---
                                        ; recall value from global variable B
            «                           ; --- right shift menu option ---
                                        ; provide with on-line help
            CLLCD                       ; clear display
            "COEFF. OF X" 1 DISP        ; show message in screen row 1
            0 WAIT DROP                 ; wait for a key to be pressed
            »
            }
      }
      {
      "→C"                              ; set label for coefficient C
            {
            « 'C' STO »                 ; --- unshifted menu option ---
                                        ; store value in global variable C
            « C »                       ; --- left shift menu option ---
                                        ; recall value from global variable C
            «                           ; --- right shift menu option ---
                                        ; provide with on-line help
            CLLCD                       ; clear display
            "CONST TERM" 1 DISP         ; show message in screen row 1
            0 WAIT DROP                 ; wait for a key to be pressed
            »
            }
      }
      {
      "SOLV"                            ; set label for the quadratic solver
            {
            «                           ; --- unshifted menu option ---
            A B C                       ; push the quadratic coefficients in the stack
            QR                          ; invoke QR, the quadratic solver subroutine
            »
            «                           ; --- left shift menu option ---
            »                           ; DO NOTHING!
            «                           ; --- right shift menu option ---
            "QUADRATIC SOLVER"          ; provide on-line help
```

```
            1 DISP
            0 WAIT DROP
            »
            }
      }
      {
      "EXIT"                          ; set label for exit option
            {
            «
            'A' PURGE                 ; purge the global variables A, B, and C
            'B' PURGE
            'C' PURGE
            { } MENU                  ; clear the custom menu
            2.01 MENU                 ; select the VAR menu
            »
            }
      }
} MENU                                ; build and activate the custom menu
»
```

The above program object should be stored in a variable, say, MQDR. When the MQDR program is executed is sets up the custom menu shown below:



The first three custom options manage the global variables A, B, and C involved with the quadratic solver. The menu options for these coefficients are coded very similarly. The unshifted option stores the number in level 1 of the stack in the corresponding global variable. The left shifted menu option recalls the content of that variable. The right shift option provides a verbose explanation of what that variable is.

The SOLV menu option offers two active choices. The unshifted option pushes the data in the global variables A, B, and C in the stack and then invokes the subroutine QR (QR can be any version of the quadratic solver in Chapter 3). The left shift option is made idle by using an empty program object. The right shift option has an program object that displays a verbose program title. The SOLV option shows that you can invoke program objects stored in other variables.

The EXIT option is the most interesting one. Its task is to remove the global variable used, remove the custom menu, and return to the VAR menu. Inspecting the CST variable after the action of this option returns an empty list. You can delay selecting this option until you need to reuse the custom menu. This means that you can move to other menus and return to the custom menu at your discretion.

The HP48SX also allows the use of temporary menus. They are more transient than custom menus. Moreover, using temporary menus does not overwrite custom menus. The TMENU command works like MENU. You can use the temporary custom menu if the use of the menu-aided program is just that -- temporary.

Menu-aided programs can also use a hierarchy of menus. This requires a more elaborate scheme of custom menu manipulation. It is worth mentioning that such programs simulate multiple levels in the same directory. The removal of variables can be done as you go up the menu levels -- the variables of the level left behind are removed. An alternate scheme is to remove all the global variables (created by the different menu levels) by an EXIT-type option located at the highest menu level.

Notes

5

Operators and Expressions

Operators and expressions for the different data objects will be examined in this chapter. Operators are special functions. They work with one or two operands and are either symbols (such as the + sign) or reserved keywords (such as MOD). Expressions contain one or more sets of operators and operands.

Mathematical Operators and Expressions

This class of operators and expressions is involved in computing a mathematical result. In this section I will discuss these operators applied to real numbers, complex numbers, binary integers, arrays, and matrices.

Real Numbers

The operators for real numbers are familiar to calculator users. These include the operators for the four operations, as well as the power and modulus operators. Figure 5.1 lists these operators and includes examples in algebraic objects. The figure also shows the priority levels of these operators when they appear in algebraic expressions. More about this next.

The HP41C user is familiar with RPN expressions --- the sequence of RPN commands that calculate a result. By contrast, the BASIC programmer is accustomed to algebraic expressions. The latter type of expression is generally easier to read, unless the expression is riddled with parentheses. Examples of both types of expressions are shown in Figure 5.2.

Case number 1 shows a simple expression containing a single operator. Case number 2 shows an algebraic expression with two + operators and a * operator. The equivalent RPN expression is written to account for the different priority levels between the + and * operators in the algebraic expressions. Figure 5.3 shows the sequence of evaluating the latter. This sequence matches that of the RPN expression. Case number 3 shows how parentheses are used to evaluate the priority of the first + operator in the algebraic expression. Figure 5.4 shows how that results in the sequential evaluation of the operators. Case number 4 also shows how parentheses are used to give the * operator a higher precedence over the power operator.

Operator	Function	Priority Level	Example
+	unary plus	4	'+5'
-	unary minus	4	'-5'
+	add	1	'5+8'
-	subtract	1	'22-5'
*	multiple	2	'4*45'
/	divide	2	'355/113'
^	raise to power	3	'2^3'
MOD	modulus	2	'355 MOD 7'

Note: Higher priority operators are performed before lower ones.

Figure 5.1. The operators for real numbers.

Complex Numbers

The operators of the complex numbers are the same ones for the real numbers, except the modulus operator. The unary negation (i.e., change of sign) works on both real and imaginary components of a complex number. The complex number operators have the same priority of evaluation in algebraic expressions.

Complex and real numbers can appear in the same expression. The following rules apply:

- Multiplication and division: the real number is applied to both parts of the complex number.
- Addition, subtraction, and raising to a power: the real number is first converted into a complex number with a zero imaginary part.

Case #	RPN Expression	Algebraic Expression	Result
1	5 6 +	5 + 6	11
2	3 4 5 * + 6 +	3+4*5+6	29
3	3 4 + 5 * 6 +	(3+4)*5+6	41
4	355 113 / 2 ^	(355/113)^2	9.87

Figure 5.2. Examples of algebraic expressions and their equivalent RPN expressions.

Binary Integers

The mathematical operators for binary integers are essentially limited to the four basic operators. Another limitation comes from the fact that binary integer constants cannot enter algebraic expressions. However, variables storing binary integers can appear in algebraic expressions. Figure 5.5 lists the mathematical operators. I have omitted the unary negation operator, since it has the same effect as a bitwise-NOT operator, presented later.

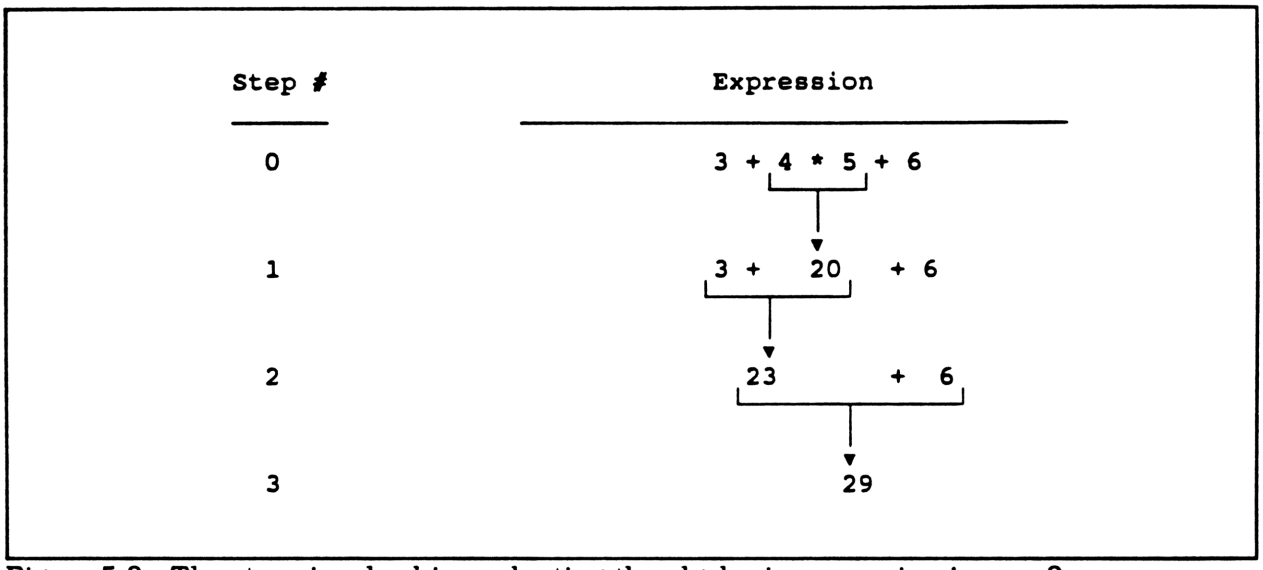


Figure 5.3. The steps involved in evaluating the algebraic expression is case 2.

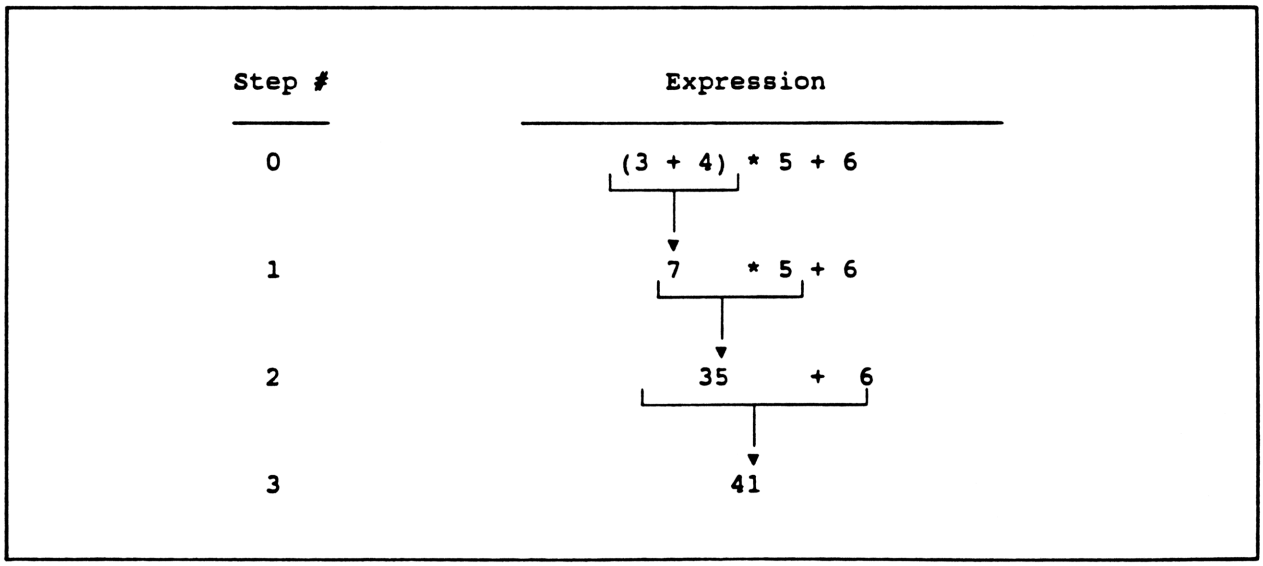


Figure 5.4. The steps involved in evaluating the algebraic expression is case 3.

Operator	Function	Priority Level	Example	Result
+	add	2	#12d #FFh +	267d
-	subtract	2	#FFFFh #1000d -	64535d
*	multiple	1	#27o #100b *	134o
/	integer divide	1	#355d #113d /	3d

Figure 5.5. The math operators for the binary integers.

RPL allows a limited interaction between binary integers and real numbers. By contrast, no interaction is allowed with complex numbers. If such operations are desired, the binary integers must be first converted into real numbers using the B→R command.

Real Arrays and Matrices

The mathematical operators for arrays work with real numbers, complex numbers, other arrays, and matrices. The following operations are supported:

- Negation of an array results in the negation of all its elements.
- Communicative multiplication between an array and a real number. The result is a real array.
- Communicative multiplication between an array and a complex number. The result is a complex array.
- Division of an array by a real number (but not the other way around). The result is an array of real numbers.
- Division of an array by a complex number (but not the other way around). The result is a complex array.
- Addition and subtraction of two arrays that have the same size.

Figure 5.6 lists the operators between arrays and real numbers, complex numbers, and other arrays.

Operator	Operand1	Operand2	Result	Examples
-	array		array	[1 1] NEG gives [-1 -1]
*	real	array	array	5 [1 1 1] * gives [5 5 5]
	array	real	array	[1 1 1] 5 * gives [5 5 5]
	complex	array	complex array	(1,1) [2 2] * gives [(2,2) (2,2)]
	array	complex	complex array	[2 2] (1,1) * gives [(2,2) (2,2)]
/	array	real	array	[5 5 5] 5 / gives [1 1 1]
	array	complex	complex array	[10 10] (2,2) / gives [(2.5,-2.5) (2.5,-2.5)]
+	array	array	array	[1 1] [2 2] + gives [3 3]
-	array	array	array	[5 5] [1 1] - gives [4 4]

Note: the HP48SX allows the addition and subtraction of arrays that have the same size. You can mix complex and real arrays.

Figure 5.6. The mathematical operations between arrays and real numbers, complex numbers, and other real arrays.

The mathematical operators for matrices are similar to those for arrays. The same operations for arrays are supported, plus the following:

- Multiplying an array and a matrix. The array must be in level 1 and the matrix in level 2. The size of the array must equal the number of matrix columns. The result is an array whose size is equal to the number of matrix rows.
- Dividing an array by a matrix. This essentially solves a system of linear equations.
- Multiplying two matrices. The first matrix has m rows and n columns. The second matrix has n rows and p columns. The result is a matrix with m rows and p columns.

Figure 5.7 shows the mathematical operators for real matrices. Examples are also included.

Operator	Operand1	Operand2	Result	Examples
-	matrix		matrix	[[1 1][1 1] NEG gives [[-1 -1][-1 -1]]
*	real	matrix	matrix	5 [[1 1][1 1]] * gives [[5 5][5 5]]
	matrix	real	matrix	[[1 1][1 1]] 5 * gives [[5 5][5 5]]
	complex	matrix	complex matrix	(1,1) [[2 2][1 1]] * gives [[(2,2) (2,2)][(1,1) (1,1)]]
	matrix	complex	complex matrix	[[2 2][1 1]] (1,1) * gives [[(2,2) (2,2)][(1,1) (1,1)]]
	matrix	array	array	[[1 1][1 1]] [2 2] * gives [4 4]
	matrix	matrix	matrix	[[1 1][1 1]] [[1 1][1 1]] * gives [[2 2][2 2]]
/	matrix	real	matrix	[[5 5][5 5]] 5 / gives [[1 1][1 1]]
	matrix	complex	complex matrix	[[2 2][2 2]] (1,1) / gives [[(1,-1) (1,-1)] [(1,-1) (1,-1)]]
	array	matrix	array	[1 2] [[1 -2][2 -1]] / gives [1 0]
+	matrix	matrix	matrix	[[1 1][1 1]] [[2 2][2 2]] + gives [[3 3][3 3]]
-	matrix	matrix	matrix	[[5 5][5 5]] [[1 1][1 1]] - gives [[4 4][4 4]]

Figure 5.7. The mathematical operators for real matrices.

Complex Arrays and Matrices

The operators and rules for complex arrays and matrices parallel those for their real counterparts. It seems that we make more of a distinction between real and complex numbers than the HP48SX does.

Relational Operators

This class of operators compares two similar objects and returns true/false values. If you program on the HP41C you most likely use instructions such as $x=y?$, $x>y?$, and $x=0?$. These instructions implement relational operators to compare the numbers in the X and Y stack

registers. The RPL language supports the relational operators found in the HP41C, BASIC, and other popular programming languages. It is important to point out that relational operators form Boolean (that is, logical) expressions that are either true or false. The false value is represented numerically by 0. The true value is represented by 1. This is one of the main differences between relational operators in the HP41C and the HP48SX. If you program in Microsoft BASIC interpreters or compilers on the IBM PC, keep this in mind --- the values -1 and 0 represent true and false on such implementations.

Programming Note

It is better to use relational operators that test 0 (false) than 1 (true). This is because 0 is universally accepted as false by any language and implementation that represents logical values using integers.

Figure 5.8 lists the relational operators. The ==, ≠, and SAME operators work with all object types. The rest of the operators work with real numbers, binary integers, strings, and units.

Boolean expressions can appear in either algebraic or RPN forms. Examples of the forms are shown below:

```
'X > 0'
'Y < 5.5'
```

The equivalent RPN forms are shown below:

```
X 0 >
Y 5.5 <
```

In the RPN form, the left and right operands are located in levels 2 and 1, respectively. The algebraic form is easier to read than the RPN form. Mathematical expressions (in both RPN and algebraic forms) can appear where real numbers are expected. Thus, the following are valid comparisons:

```
'(23.4*X+2) > (TAN(Y)+1.2)'
'((2*X+3)*X-5) == 0'
```

The equivalent RPN forms are:

```
23.4 X * 2 +
Y TAN 1.2 +
>
```

and

```
2 X * 3 + X * 5 -
0
==
```

Operator	Test	Example
==	operand1 equal operand2	'X == 0' 'Y == 10'
≠	operand1 is not equal to operand2	'X <> Y' 'Z <> -1'
>	operand1 is greater than operand2	'X > 0' 'X > Y'
<	operand1 is less than operand2	'X < 0' 'X < Y'
≤	operand1 is less than or equal to operand2	'X ≤ 0' 'X ≤ Y'
≥	operand1 is greater than or equal to operand2	'X ≥ 0' 'X ≥ Y'
SAME	operand1 and operand2 are equal objects	X Y SAME

Note: The command SAME is a function, not an operator. I have included it because (1) it is important to RPL, and (2) it is unique to the HP48SX.

Figure 5.8. The relational operators.

Programming Note

Relational operators and their operands can by chained only by using Boolean operators. This applies to both RPN and algebraic forms.

Boolean Operators

The discussion on Boolean expressions began in the last section. Relational operators need Boolean operators to build more complex conditions. The Boolean operators are AND, OR, XOR, and NOT. The first three are binary operators. The NOT operator is a unary operator that performs logical negation. The operands of the Boolean operators are Boolean values. Such

values are usually obtained by relational operators. You can use a real number as an operand to these operators. The operator considers the number as representing a false Boolean value if it is equal to 0; otherwise, it represents a logical true value. This is why the last programmer's note recommends the testing with 0. Figure 5.9 shows the truth table. This table is to logic what the multiplication table is to arithmetic.

Operand 1	Operand 2	Boolean Operator	Result
true	true	AND	true
true	false	AND	false
false	true	AND	false
false	false	AND	false
true	true	OR	true
true	false	OR	true
false	true	OR	true
false	false	OR	false
true	true	XOR	false
true	false	XOR	true
false	true	XOR	true
false	false	XOR	false
true		NOT	false
false		NOT	true

Figure 5.9. The Truth table.

The Boolean operators are familiar to BASIC programmers, but not to HP41C programmers. This does not mean that the HP41C programs cannot implement Boolean logic. Its just less obvious. If you are not familiar at all with the Boolean operators, here is a brief description of their functions:

- The AND operator declares a condition to be true if and only if its two sub-conditions (that is, the operands) are true.

- The OR operator declares a condition to be true if either or both of its sub-conditions are true.

- The XOR operator declares a condition to be true if only one of its sub-conditions is true. The X in XOR stands for eXclusively OR.

- The NOT operator toggles the logical value of its operand. It resembles the change-of-sign mathematical operator.

Examples of Boolean operators are shown below:

Algebraic Form	RPN Form
'X > 0 AND X < 11'	X 0 > X 11 < AND
'I > 1 AND I < 100 AND I ≠ J'	I 1 > I 100 < AND I J ≠ AND
'NOT A < 10'	A 10 < NOT

RPL applies the following rules on Boolean expressions:

- The relational operators have a higher priority than Boolean operators. Since relational operators are evaluated before Boolean ones, you need to enclose relational operators and their operands in parentheses (when using algebraic forms). In fact, when the HP48SX parses an algebraic object, it throws out the parentheses that contain relational and Boolean operators.

- The Boolean operators are evaluated from left to right in an algebraic object. You cannot use Boolean sub-expressions.

- The entire Boolean expression is evaluated.

The above rules should be noted by BASIC programmers. All BASIC implementations allow you to use parentheses to build more complex Boolean expressions, such as the ones shown below:

```
((I > 0) AND (I < 10) OR (X = 100))
(NOT ((I < 0) OR (I > 10)) AND ((J < 1) OR (J > 9)))
```

To implement such Boolean expressions in RPL you can mix between algebraic and RPN forms or use RPN form only. Applying the first solution to the first Boolean expression we get:

```
'I > 0 AND I < 10' EVAL          ; evaluate the first sub-expression
'X == 100' EVAL                  ; evaluate the second sub-expression
OR                               ; OR the two results in the stack
```

Applying the first solution to the second Boolean expression we get:

```
'I < 0 OR I > 10' EVAL           ; evaluate the first sub-expression
'J < 1 OR J > 9'                 ; evaluate the second sub-expression
AND                              ; AND the above results
NOT                              ; logically negate the last result
```

Applying the second solution to the first Boolean expression we get:

```
I 0 > I < 10 AND                 ; evaluate the first sub-expression
```

```
X 10  ==                          ; evaluate the second sub-expression
OR                                ; OR the two results in the stack
```

Applying the second solution to the second Boolean expression we get:

```
I 0 < I 10 > OR                   ; evaluate the first sub-expression
J 1 < J 9 > OR                    ; evaluate the second sub-expression
AND                               ; AND the above results
NOT                               ; logically negate the last result
```

Concatenation Operators

Traditionally, most languages and implementations support concatenation operators to join two similar objects. The most popular example is the BASIC + operator that joins strings and characters. RPL has extended the use of the + concatenating operator to include strings and lists. The concatenation operator is communicative. Figure 5.10 shows examples of concatenating strings with other object types. The last example shows the interesting result of concatenating a string and a list. RPL treats that operator as the list concatenation operator.

Operand 1	Operand 2	Result
"A"	" word"	"A word"
"Hello"	" World"	"Hello World"
"Cost = $"	123.55	"Cost = $123.55"
"Address is "	# 11h	"Address is # 11h"
"Volume = "	1_m^3	"Volume = '1_m^3'"
"Vector is "	[1 2 3]	"Vector is [1 2 3]"
"Root = "	(1,2)	"Root = (1,2);
"Me"	{ 4.5 (3,4) }	{ "Me" 4.5 (3,4) }

Figure 5.10. Examples of using the string concatenation operator to combine strings and other object types.

RPL allows you to concatenate a list with just about every object type, including other lists. However, concatenating a non-empty list with an empty does not result in the growth of the former list!

Bitwise Operators

This class of operators specializes in manipulating the zeros and ones of binary integers and contains two types of operators:

- Operators that manipulate the bits of one or two binary integers, resulting in a third binary integer. The operators are the AND, OR, XOR, and NOT. They parallel the Boolean operators that bear the same name. The first three bitwise operators take two operands, while the last one takes only one operand.

- Operators that manipulate the bits of a binary integer.

The first type of operators manipulates the entire bits of a binary integer. The change in bits uses the bitwise truth table, shown in Figure 5.11. This table is a version of the Boolean truth table where 0 replaces false, and 1 replaces true.

Before I proceed in discussing the bitwise operators I need to explain the orientation of bits in a binary integer. Figure 5.12 shows random bits of a 64-bit binary integer. The bits are numbered 0 to 63, and from right to left. The leftmost bits are called the most significant bits, while the rightmost bits are called the least significant bits. Figure 5.12 shows the maximum size of a *word* on the HP48SX. The RCLWS command returns the current number of bits in a word. The STOWS command enables you to store a new word size. For example, if you are using the HP48SX to do binary math and bit manipulations for an IBM PC assembler, set the word size to 16 bits. The valid range of arguments for the STPWS is 1 to 64. If the argument to the STOWS command exceeds 64, the HP48SX uses 64. If the argument for STOWS is less than 1, the HP48SX uses 1.

Bit # 1	Bit # 2	Bit Operator	Result
1	1	AND	1
1	0	AND	0
0	1	AND	0
0	0	AND	0
1	1	OR	1
1	0	OR	1
0	1	OR	1
0	0	OR	0
1	1	XOR	0
1	0	XOR	1
0	1	XOR	1
0	0	XOR	0
1		NOT	0
0		NOT	1

Figure 5.11. The Bit truth table.

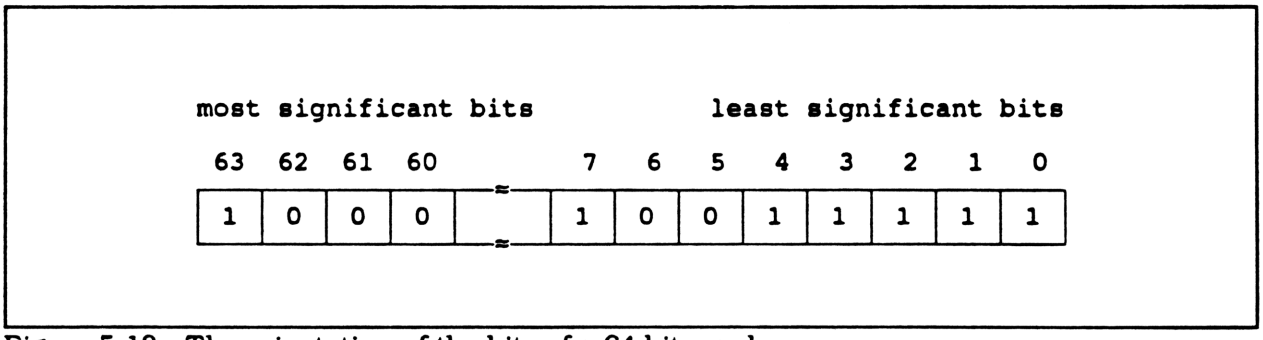


Figure 5.12. The orientation of the bits of a 64-bit word.

Figure 5.13 shows the AND, OR, XOR, and NOT operators. The examples of how these operators work are shown in hexadecimal and binary bases. The numbers in binary base give you an exact picture of how the operators work. The AND, OR, and XOR operators work by manipulating the corresponding bits of its operands. The NOT operator works by simply reversing the 1s and 0s of a binary integer.

Operator	Binary Integer Operands	Binary Base
AND	# 55h	# 1010101b
	# 4Ch	# 1001100b
	# 44h	# 1000100b
OR	# 55h	# 1010101b
	# 4Ch	# 1001100b
	# 5Dh	# 1011101b
XOR	# 55h	# 1010101b
	# 4Ch	# 1001100b
	# 19h	# 0011001b
NOT	# FFh	# 11111111b
	# FF00h	# 1111111100000000b

Figure 5.13. The bitwise operators AND, OR, XOR, NOT (the current word size setting is assumed to be 16 bits).

The other bitwise operators shift or rotate the bits of a binary integer. The set of shift operators are:

1) The SR Shift Right operator. This operator shifts the bits of a binary integer to the right by 1 bit. The most significant bit is assigned 0, while the least significant bit is lost. Figure 5.14 shows how the SR command works on an 8-bit word. The SR operator reduces the magnitude of the binary integer, since the leading bit becomes 0.

2) The SL Shift Left operator. This operator shifts the bits of a binary integer to the left by 1 bit. The most significant bit is lost, while the least significant bit is assigned 0. Figure 5.15 shows how the SL command works on an 8-bit word.

3) The SRB Shift Right Byte operator. This operator shifts the bits of a binary integer by 8 bits to the right. The most significant 8 bits (that is, the most significant byte) are assigned 0s, while the least significant 8 bits are lost. Figure 5.16 shows how the SR command works on an 8-bit word. The SRB operator reduces the magnitude of the binary integer, since the leading bits become 0.

4) The SLB Shift Left Byte operator. This operator shifts the bits of a binary integer by 8 bits to the left. The most significant 8 bits (that is, the most significant byte) are lost, while the least significant 8 bits are assigned 0s. Figure 5.17 shows how the SR command works on an 8-bit word.

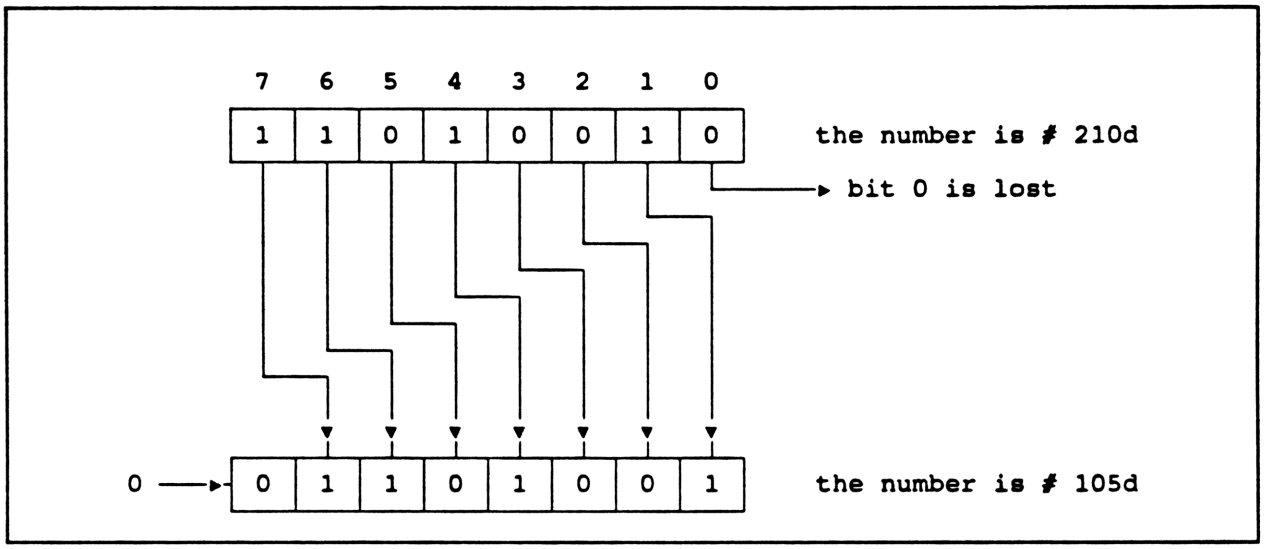


Figure 5.14. The SR shift right operator.

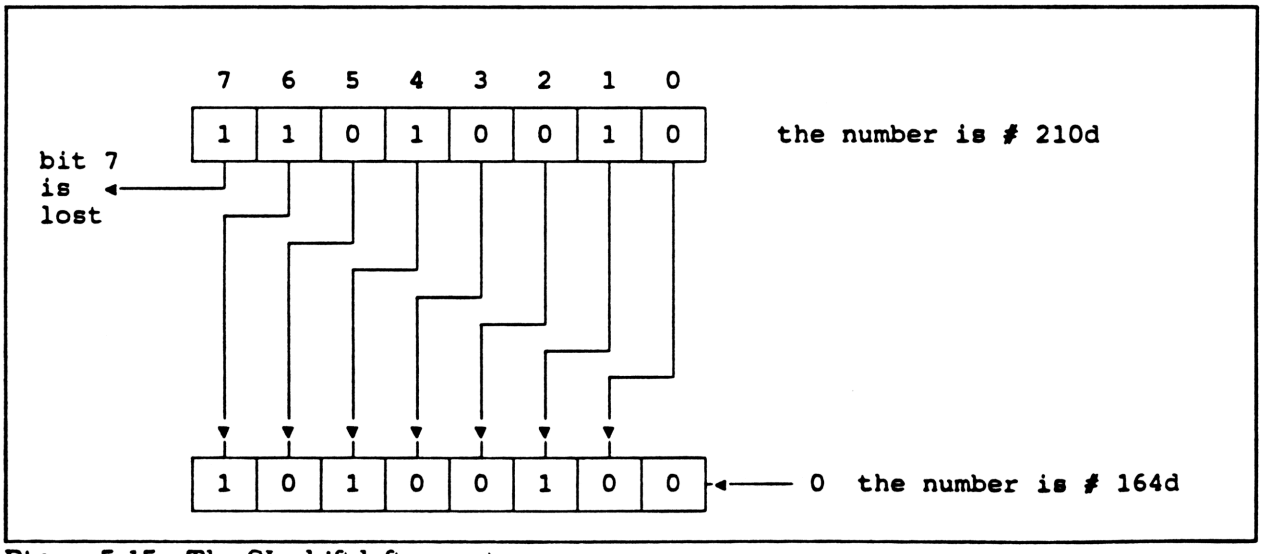


Figure 5.15. The SL shift left operator.

5) The ASR Arithmetic Right Shift operator. This is similar to the SR operator, except that the most significant bit is not affected. Figure 5.18 shows how the ASR operator works on an 8-bit word.

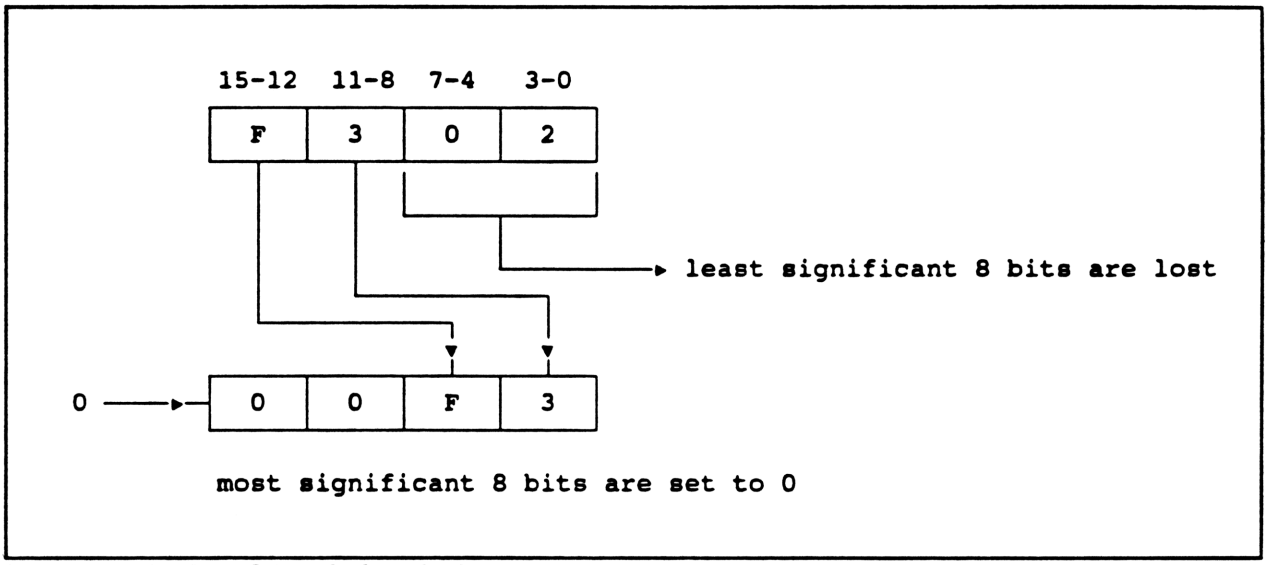


Figure 5.16. The SRB shift right byte operator.

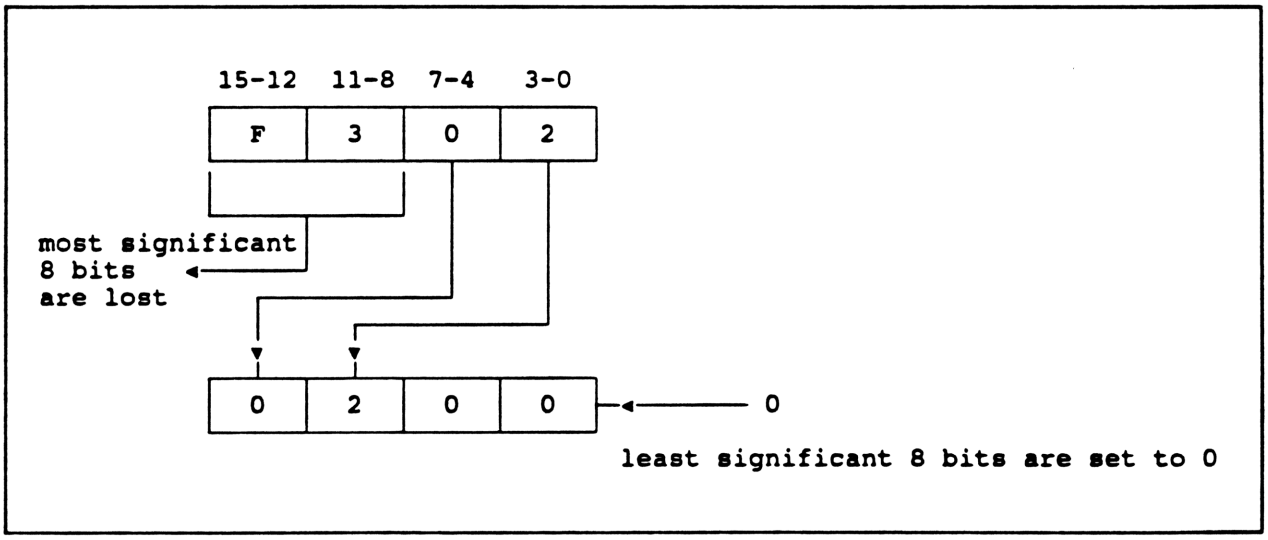


Figure 5.17. The SLB shift left byte operator.

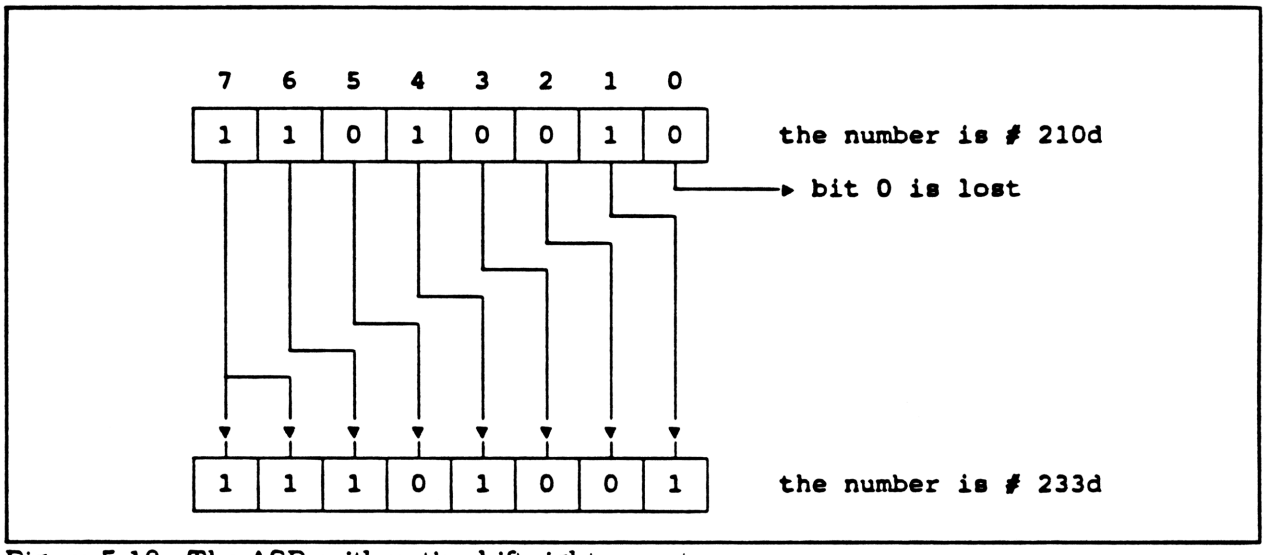


Figure 5.18. The ASR arithmetic shift right operator.

The set of bit rotation operators modifies a binary integer by rotating its bits either to the left or to the right. Unlike the shift operators, this set of operators preserves the bits, but changes their location. Thus, it is possible to restore the original value of a binary integer by repeatedly applying the rotation operators until the original bit configuration is restored. The operators are:

1) The RR Rotate Right operator. This operator rotates the bits of a binary integer to the right by 1 bit. In the process bit 0 becomes the most significant bit, while every other bit becomes less significant. Figure 5.19 shows how the RR operator works on the bits of an 8-bit word.

2) The RL Rotate Left operator. This operator rotates the bits of a binary integer to the left by 1 bit. In the process the most significant bit is moved to bit 0, while every other bit becomes more significant. Figure 5.20 shows how the RR operator works on the bits of an 8-bit word.

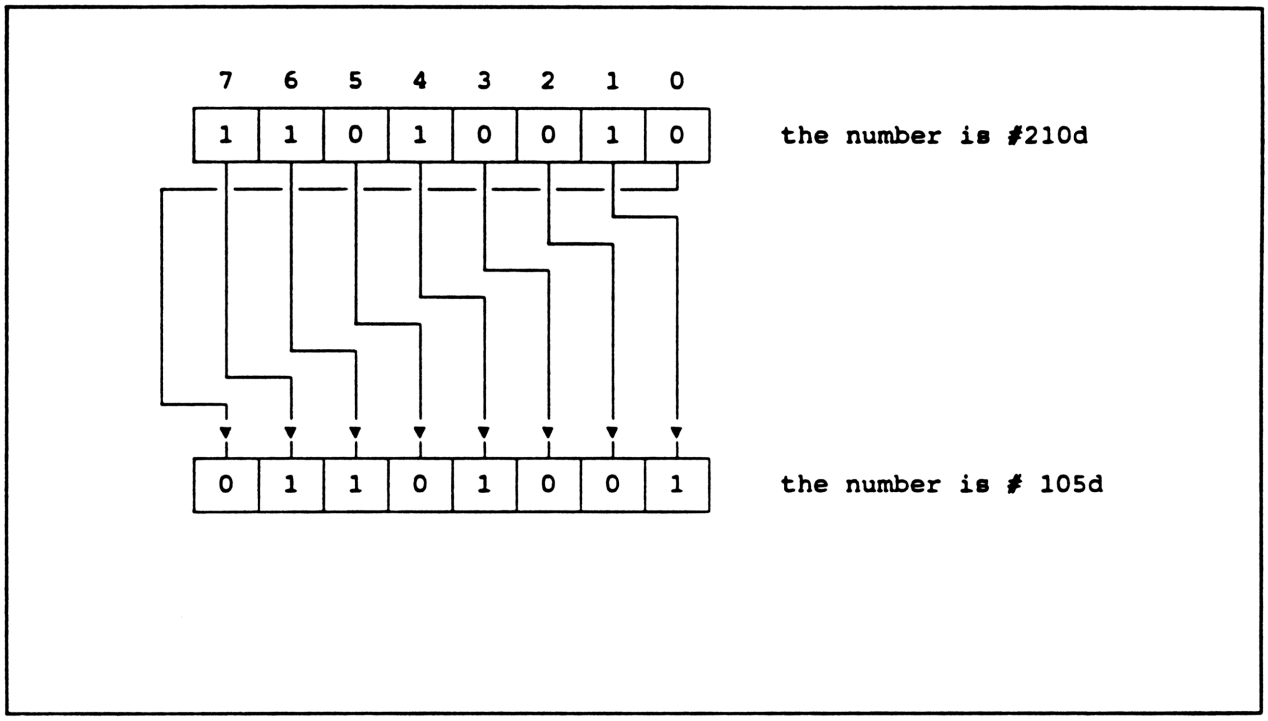


Figure 5.19. The RR rotate right operator.

3) The RRB Rotate Right Byte operator. This operator rotates the bits of a binary integer by 8 bits to the right. The least significant byte becomes the most significant one, while the other bytes become less significant. Figure 5.21 shows how the RRB command works on a 16-bit word.

4) The RLB Rotate Left Byte operator. This operator rotates the bits of a binary integer by 8 bits to the left. The most significant byte becomes the least significant one, while the other bytes become more significant. Figure 5.22 shows how the RLB command works on a 16-bit word.

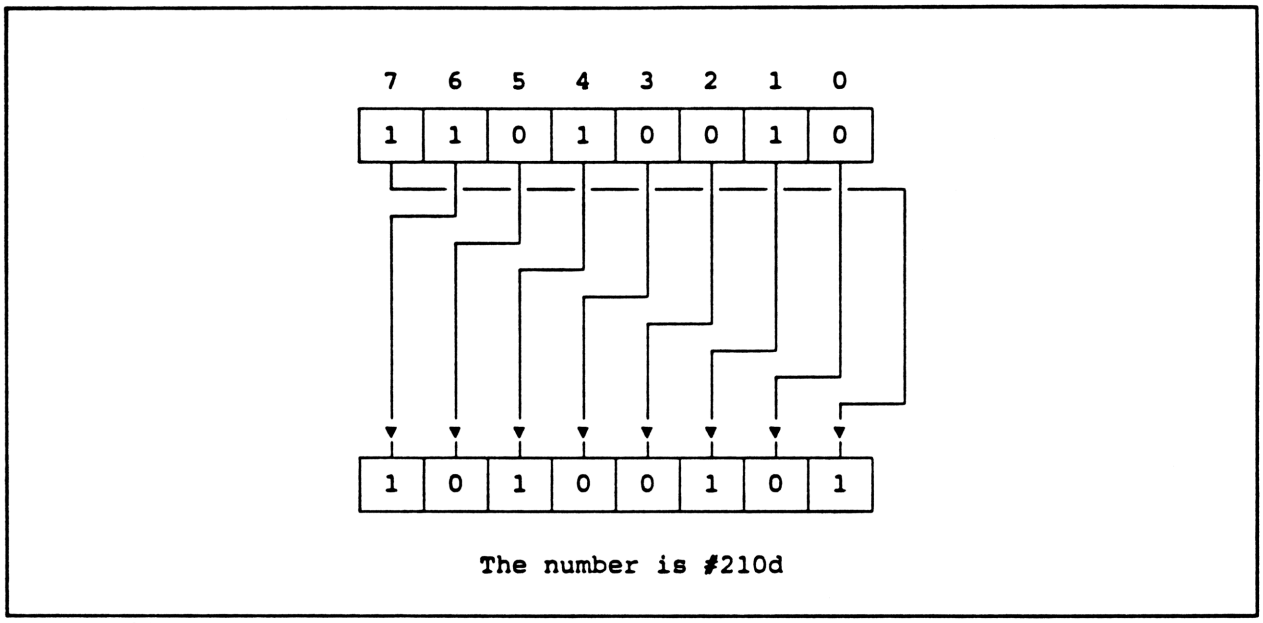


Figure 5.20. The RL rotate left operator.

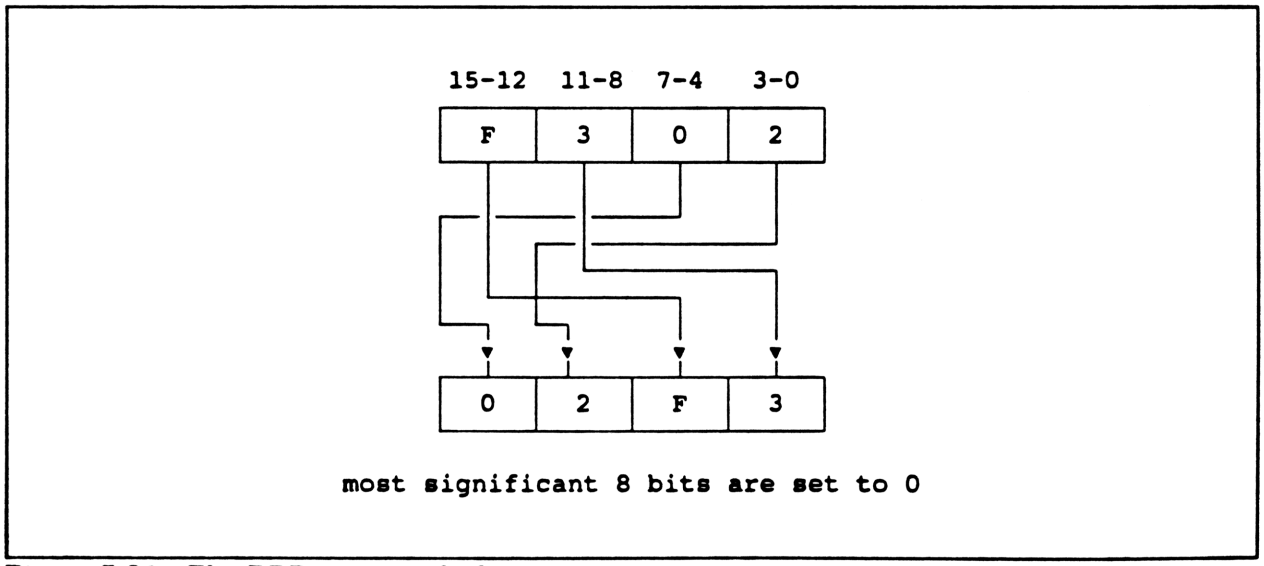


Figure 5.21. The RRB rotate right byte operator.

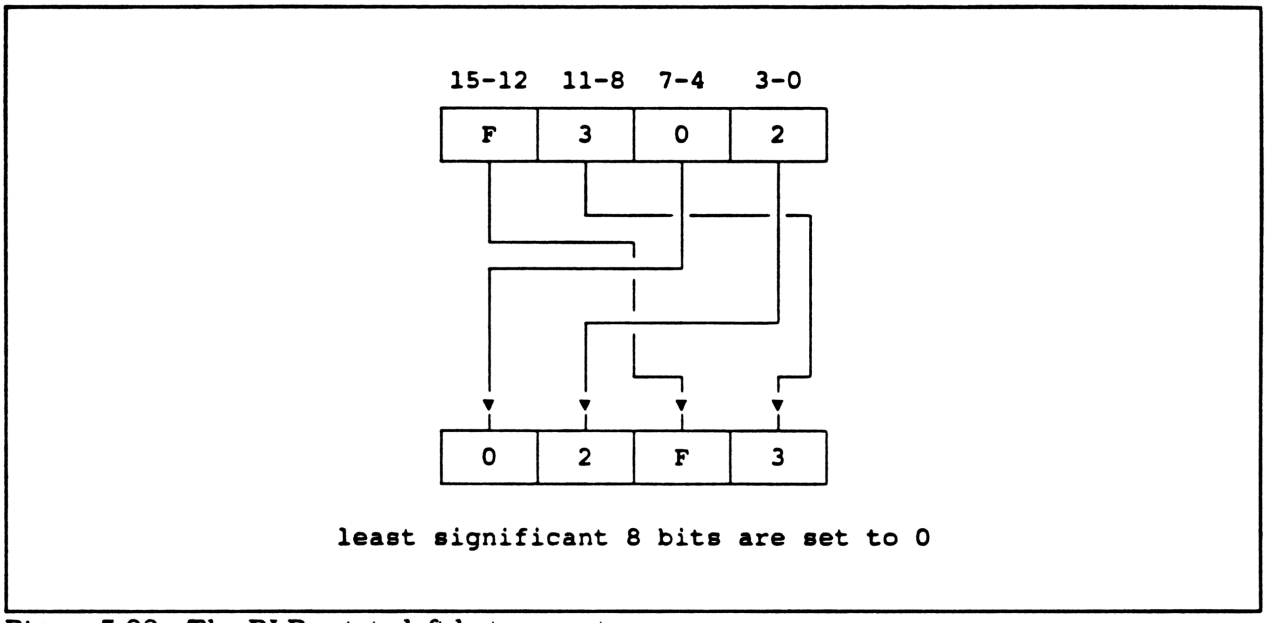


Figure 5.22. The RLB rotate left byte operator.

The EVAL Operator

I consider the EVAL command a special (but not unique, since a few programming languages implement similar operators) unary operator. The HP48SX EVAL command goes beyond simple numerical evaluation of algebraic expressions. In fact, when EVAL is invoked it tells its argument (whatever is on level 1) **to do its thing**. Here is a brief description of how EVAL works with various object types:

• Algebraic objects which may contain math expressions with names, commands, and other objects. The names are evaluated, the commands are executed, and the rest are pushed on the stack. Examples of evaluating algebraic objects are shown below:

Algebraic object	Result	Comments
'1+2'	3	
'2*X'	6	assuming X has a value of 3
'2*X'	'2*X'	there is no variable by the name of X
'SIN(45)'	0.70711	the command SIN is executed

• Global names are evaluated in various ways depending on their contents:

  • Numeric data, strings, lists, arrays are pushed on the stack.
  • Programs are executed.

- Directories become the current ones.
- Other objects are pushed on the stack.

Examples of evaluating global names are shown below:

Global Name	Result	Comments
'X'	5	if X stores 5
'ME'	"Namir"	if ME stores "Namir"
'Weekend'	{ Sat Sun }	if Weekend stores the list { Sat Sun }
'XARR'	[1 2 3]	if XARR stores the array [1 2 3]
'QD'	executes a program	if QD contains a program object
'STAT'	move to directory { HOME STAT LR }	if STAT contains a directory
'MYART'	pushes a graph object on the stack	if MYART stores a graphics image

- Local names have their contents recalled and if suitable, evaluated.

- Program objects are executed. Each item in the program object is evaluated individually (this is actually how an HP48SX program runs). Names are evaluated, commands executed, and the other objects are pushed on the stack.

- Lists are evaluated by individual list members. Names are evaluated, programs run, commands executed, and the other objects are pushed on the stack. Examples of evaluating lists are shown below:

List object	Result	Comments
{ 45 SIN }	0.70711	the number 45 is pushed on the stack and then the SIN command is executed
{ 25 SQRT INV }	0.2	the number 25 is pushed on the stack, the user-defined program, SQRT is run to return the square root of 25, and the INV command is executed, returning 0.2
{ ME X PURGE }	VAR menu updated	purges directory entries in the list.

Programming Note

You can use a list containing the names of programs, commands, and data to emulate batch programming available on mainframes and microcomputers. This is very suitable for chained programs that perform a time-consuming number crunching. Of course, these programs should maintain minimum interaction with the end-user.

Notes

6

Decision-Making

The last chapter presented Boolean expressions. Such expressions evaluate conditions, based on which one or more courses of action are taken. This chapter looks at the RPL decision-making constructs that use Boolean expressions. This includes the single-alternative IF-THEN-END, the dual-alternative IF-THEN-ELSE-END, and the multi-alternative CASE-END control structures.

The Single Alternative IF-THEN-END

The simplest decision-making construct is the single-alternative IF-THEN-END control structure. The structure executes a set of commands if a tested condition is true. Otherwise, no action is taken. The general syntax of the IF-THEN-END structure is:

IF *condition is true* THEN *sequence of commands* END

The following program contains an IF-THEN-END structure. It calculates the sale price of an item using the base price and the state sales tax. The program tests if the percentage of state tax is entered as a factional number or as a percent. Numbers greater than 1 are treated as percentages and need to be divided by 100. The listing is shown below:

```
; Listing 6.1.  Sale price calculator (RPN version)
«
  "Enter base sale price" ""
  INPUT OBJ→                          ; prompt for input
  "Enter tax rate" ""
  INPUT OBJ→
  DUP 1                               ; duplicate tax and push 1 in stack
  IF > THEN                           ; is tax rate greater than 1?
     100 /                            ; yes.  Divide tax rate by 100
  END
  1 +                                 ; calculate final sale price
  *
»
```

The body of the THEN clause contains two items: the real number 100 and the divide command. Let's single-step through two sample sessions to examine closer how the IF structure works. To proceed with the first sample session enter 'SALE' (assuming that this is the name of the variable string the sale program) in the stack, press the [PRG] key and select the CTRL and DBUG options. Single-step through the program using the SST menu option. The program tracing proceeds as follows:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	"Enter base sale price"	"Enter base sa...	prompt string is in level 1
2	""	""	command-line string is in level 1
3	INPUT	""	prompt for user input
4		"150"	your input representing $150
4	OBJ→	150	convert a real number
5	"Enter tax rate"	"Enter tax...	push prompt string in level 1
6	""	""	push command-line string in level 1
7	INPUT	""	prompt user for input
8		"4.5"	your input representing 4.5%
9	OBJ→	4.5	convert into a real number
10	DUP	4.5	duplicate number in level 1
11	1	1	push 1 in stack
12	IF	1	beginning of IF structure
13	>	1	test if real number in level 2 (i.e., the tax) is greater than 1. Since the condition is true, a 1 is placed in level 1
14	THEN	4.5	the program proceeds in the THEN clause
15	100	100	push 100 in stack
16	/	0.045	divide 4.5 by 100
17	END	0.045	end of IF structure
18	1	1	push 1 in the stack
19	+	1.045	add 1 to the tax
20	*	156.75	calculate sale price

```
21      »                       156.75          end of program
```

The input for the above session caused the THEN clause to be executed. To make the program bypass that clause, rerun it and enter 0.045 when prompted for the tax. As you trace the program in this case, steps 14 through 17 are bypassed.

The program version that uses an algebraic condition form is shown below:

```
; Listing 6.2.  Sale price calculator (algebraic version)
«
  "Enter base sale price" ""
  INPUT OBJ→                          ; prompt for input
  "Enter tax rate" ""
  INPUT OBJ→
  → I                                 ; store tax in local variable I
     «
     I                                ; push I in the stack
     IF 'I>1' THEN                    ; is tax rate greater than 1?
        100 /                         ; yes.  Divide tax rate by 100
     END
        1 +                           ; calculate final sale price
     *
     »
»
```

The major difference between the two versions is that the algebraic one stores the tax in a local variable. This is required for the algebraic condition. While both program versions are simple, they represent typical use of the IF structure. The structure is usually preceded and followed by other commands. The action taken by the IF structure adjusts data values before resuming with other calculations.

The single-alternative IF structure is available in all BASIC implementations. The following BASIC listing implements the same HP48SX sale calculator program:

```
' QuickBASIC version
CLS
INPUT "Enter base sale price : ";S
INPUT "Enter tax ";I
IF I > 1 THEN I = I / 100 ' divide the tax rate by 100
PRINT "Sale price = $"; S * (1 + I)
END
```

How does the RPN version of the above program compare to an HP41 version? I present the HP41 version along with the RPN version. The following set of listings attempts to match similar aspects of the two programs:

```
HP41 Program                HP48SX Program
LBL SALE                    «
"BASE PRICE"                "Enter base sale price" "" INPUT OBJ→
PROMPT
1
"TAX RATE"                  "Enter tax rate" "" INPUT OBJ→
```

```
PROMPT
                          DUP
                          1
X < Y?                    IF > THEN
GTO 00
100                            100
/                              /
LBL 00                    END
                          1
+                         +
*                         *
RTN                       »
```

The major differences between implementing single-alternative control structures on the HP41 and the HP48SX are:

- HP48SX objects that make up the tested condition are consumed by its evaluation. This is not the case with the HP41.

- The HP41 executes the instruction right after the test, if the test is true. GOTOs must be used when multiple imbedded instructions must be executed when the test is positive. By contrast, GOTOs are not needed in RPL. In fact, GOTOs are not even implemented in the HP48SX!

Programming Note

RPL does not implement GOTOs!

While programming languages (such as Pascal and C) support GOTOs but discourage you from using them, RPL does not implement GOTO at all!

Life Without GOTOs

The sale calculator program gives HP41 and BASIC programmers an example of structured programming without GOTOs. I have to admit that GOTOs have gotten a very bad press. Many programmers look down at other colleagues if their listings contain GOTOs!

RPL is a highly structured reverse-polish language. As with any structured programming language, you must view the sequence of commands as coherent and related blocks. Each block performs a specific task or subtask. Consequently, allowing you to jump in the middle of a block of commands is structured programming heresy (or is illogical, in the words of a famous sci-fi TV character). The designers of RPL have elected to resist this heresy to the fullest

by not offering GOTO statements. In return, they provide the typical structures for looping and decision-making that succeed in making life without GOTOs an easy one.

The Dual-Alternative IF-THEN-ELSE-END

The dual-alternative IF-THEN-ELSE-END structure empowers a program to examine a condition and take alternate actions. The general syntax of this form of IF structure is:

IF *condition is true*
THEN
    sequence of commands
ELSE
    alternate sequence of commands
END

The following HP48SX program illustrates the dual-alternative IF structure. The program solves for either the present or future value of an investment using:

$$FV = PV\,(1 + i)^N$$

The program prompts you to enter the following:

- The percent interest rate.
- The number of investment periods.
- The present value. To solve for this value enter 0.
- The future value. To solve for this value enter 0.

The listing is shown below:

```
; Listing 6.3.  Financial program to solve for future or present values.
«
  "% Interest" "" INPUT OBJ→          ; prompt for interest
   100 /                              ; convert percentage to fraction
  "Periods" "" INPUT OBJ→             ; prompt for investment periods
  "Present Value" "" INPUT OBJ→       ; prompt for present value
  "Future Value" "" INPUT OBJ→        ; prompt for future value
   → N I FV PV                        ; store input in local variables
     «
     IF 'P == 0'                      ; is present value a zero?
     THEN                             ; Yes.  Solve for present value
          'PV'                        ;   push PV tag
          'FV/(1+I)^N'
     ELSE                             ; No.  Solve for future value
          'FV'                        ;     push FV tag
          'PV*(1+I)^N
     END
     EVAL                             ; evaluate expression in level 1
     SWAP →TAG                        ; swap result and tag, then tag the result
```

»
»

Let's trace the above RPL program twice to get a closer view of how the IF structure works. Each session uses a set of data that results in executing either the THEN or ELSE clause of the IF structure. In the first debugging session I will solve for the present value of an investment given the following data:

Interest rate: 10%
Investment periods: 15 years
Future value: $1000.00

To proceed with the debugging session clear the stack and enter the name of the variable storing the RPL program. Press the [PRG] key and select the CTRL and DBUG menu options. Single-step through the program using the SST menu option. The program tracing proceeds as follows:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	"% Interest" "" INPUT	""	prompt for percent interest
2		"10"	your input representing 10%
3	OBJ→	10	convert into a real number
4	100 /	0.1	divide percent interest by 100
5	"Periods" "" INPUT	""	prompt for investment periods
6		"15"	your input, representing 15 years
7	OBJ→	15	convert into a real number
8	"Present Value" "" INPUT	""	prompt for present value
10		"0"	your input of 0, since you want to solve for PV
11	OBJ→	0	convert a real number
12	"Future Value" "" INPUT	""	prompt for future value
13		"1000"	your input representing 1000 dollars

14	OBJ→	1000	convert into a real number
15	→ N I FV PV		save data in local variables
16	«		
17	IF		
18	'PV==0'	'PV==0'	push condition in stack. The value of variable PV is compared with 0. Since PV is equal to 0, the condition returns 1. This result is removed from the stack and the THEN clause is executed
19	THEN		
20	'PV'	'PV'	push tag name in stack
21	'FV/(1+I)^N'	'FV/(1+I)^N'	push algebraic expression in the stack
22	END	'FV/(1+I)^N'	program jumps to END of structure
23	EVAL	239.39	evaluate expression
24	SWAP	'PV'	swap tag into level 1
25	→TAG	PV:239.39	tag the result
26	»		
27	»	PV:239.39	end of program.

In the second debugging session I solve for the future value of an investment given the following data:

Interest rate: 10%
Investment periods: 15 years
Present value: $1000.00

To proceed with the debugging session clear the stack and enter the name of the variable storing the RPL program. Press the [PRG] key and select the CTRL and DBUG menu options. Single-step through the program using the SST menu option. The program tracing proceeds as follows:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	"% Interest" "" INPUT	""	prompt for percent Interest
2		"10"	your input representing 10%
3	OBJ→	10	convert into a real number
4	100 /	0.1	divide percent interest by 100
5	"Periods" "" INPUT	""	prompt for investment periods
6		"15"	your input, representing 15 years
7	OBJ→	15	convert into a real number
8	"Present Value" "" INPUT	""	prompt for present value
10		"1000"	your input of 1000 dollars
11	OBJ→	1000	convert into a real number
12	"Future Value" "" INPUT	""	prompt for future value
13		"0"	your input 0, since you want to solve for FV.
14	OBJ→	0	convert into a real number
15	→ N I FV PV		save data in local variables
16	«		
17	IF		
18	'PV==0'	'PV==0'	push condition in stack. The value of variable PV is compared with 0. Since PV is not equal to 0, the condition returns 0. This result is removed from the stack and the ELSE clause is executed
19	ELSE		
20	'FV'	'FV'	push tag name in stack

21	'PV*(1+I)^N'	'PV*(1+I)^N'	push algebraic expression in the stack
22	END	'PV*(1+I)^N'	program jumps to END of structure
23	EVAL	4177.25	evaluate expression
24	SWAP	'FV'	swap tag into level 1
25	→TAG	FPV:4177.25	tag the result
26	»		
27	»	FV:4177.25	end of program.

The QuickBASIC version of the above RPL program is shown below:

```
' QuickBASIC version
CLS
INPUT "Enter interest rate : ";I
I = I / 100
INPUT "Enter investment periods : ";N
INPUT "Enter present value : ";PV
INPUT "Enter future value : ";FV
IF P = 0 THEN
   PRINT "PV = ";FV/(1+I)^N
ELSE
   PRINT "FV = ";PV*(1+I)^N
END IF
END
```

The HP41 version of the above RPL program is shown side by side with the RPL listing:

HP41	RPL
LBL "FINC"	«
"% INTEREST"	"% Interest" "" INPUT OBJ→
PROMPT	
100	100
/	/
STO 00	
"PERIODS"	"Periods" "" INPUT OBJ→
PROMPT	
STO 01	
"PV"	"Present Value" "" INPUT OBJ→
PROMPT	
STO 02	
"FV"	"Future Value" "" INPUT OBJ→
PROMPT	
STO 03	
	→ N I FV PV
	«
RCL 02	
X≠0?	IF 'PV == 0'

```
GTO 00
                                    THEN
"PV="                                  'PV'
RCL 03                                 'FV/(1+I)^N'
1
RCL 00
+
RCL 02
y^x
GOTO 01
LBL 00                              ELSE
"FV="                                  'FV'
RCL 02                                 'PV*(1+I)^N
1
RCL 00
+
RCL 01
y^x
LBL 01                              END
                                    EVAL
                                    SWAP
ARCL X                              →TAG
PROMPT
                                    »
RTN                          »
```

Looking at the HP41 version, you will notice that the test verifies whether or not the value in the X register is not zero. This is the opposite of the 'PV= =0' test. This is necessary to implement the THEN and ELSE clauses in the same sequence in both programs. If I use the *X=0?* test in the HP41 version, then I need to switch the instructions that make up the *informal* THEN and ELSE clauses. In implementing the logic of an IF-THEN-ELSE-END structure, the HP41 must resort to more GOTOs. Keep in mind that this example is simple. If the Boolean expression is more complicated, a battery of GOTO's and flags are most likely needed with the 41 version.

The Multi-Alternative CASE-END Structure

The RPL language offers the CASE-END structure to implement multiple-alternative decision-making. The general syntax for the CASE-END structure is:

```
CASE
    condition #1 is true
    THEN
        sequence #1 of commands
    END

    condition #2 is true
    THEN
        sequence #2 of commands
    END

    other tested conditions and THEN clauses

    condition #n is true
    THEN
        sequence #n of commands
    END

    default sequence of commands(optional)

END
```

The CASE-END structure empowers a program to sequentially test a battery of conditions. Each condition is accompanied by a THEN clause. An optional catch-all clause may be included. The CASE-END structure works by testing the conditions in sequence. If a condition is true, its accompanying THEN clause is executed and the program resumes after the END of the CASE structure. If no condition is true, the structure executes its default clause, if present.

> *Programming Note*
>
> To enhance the speed of executing a CASE structure, place the conditions in the order of their likelihood of being true.

To show how the CASE-END works let's go back to the last RPL program. That program was

limited to solving either the present or future investment value --- the values for the interest and investment periods must be always given. The next version of the program empowers you to solve for either variable. To select a variable, enter 0 when prompted for its value. The new program version is listed below:

```
; Listing 6.4.  Progran that uses the CASE-END for financial calculations.
«
  "% Interest" "" INPUT OBJ→          ; prompt for interest
   100 /                              ; convert percentage to fraction
  "Periods" "" INPUT OBJ→             ; prompt for investment periods
  "Present Value" "" INPUT OBJ→       ; prompt for present value
  "Future Value" "" INPUT OBJ→        ; prompt for future value
   → N I FV PV                        ; store input in local variables
     «
     CASE
         'P == 0'                     ; is present value 0?
          THEN                        ; Yes.  Solve for present value
              'PV'                    ;   push PV tag
              'FV/(1+I)^N'
          END
         'FV==0'                      ; is future value 0?
         THEN                         ; Yes.  Solve for future value
              'FV'                    ;     push FV tag
              'PV*(1+I)^N'
         END
         'I==0'                       ; is interest 0?
         THEN                         ; Yes.  Solve for interest
              'I'                     ;    push I tag
              '(FV/PV)^(1/N)-1'
         END
                                      ; ******** DEFAULT CLAUSE ********
              'N'                     ; solve for investment periods
              'LN(FV/PV) / LN(1 + I)'
     END
     EVAL                             ; evaluate expression in level 1
     SWAP →TAG                        ; swap result and tag, then tag the result
     »
»
```

Let's trace the above RPL program to get a closer view of how the CASE-END structure works. Each session uses a set of data that results in executing a THEN clause and the default clause. In the first debugging session I will solve for the interest rate of an investment, given the following data:

Investment periods: 15 years
Present value: $250.00
Future value: $1000.00

To proceed with the debugging session clear the stack and enter the name of the variable storing the RPL program. Press the [PRG] key and select the CTRL and DBUG menu options. Single-step through the program using the SST menu option. The program tracing proceeds as follows:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	"% Interest" "" INPUT	""	prompt for percent interest
2		"0"	your input, 0.
3	OBJ→	0	convert string into a real number
4	100 /	0	divide percent interest by 100
5	"Periods" "" INPUT	""	prompt for investment periods
6		"15"	your input, representing 15 years
7	OBJ→	15	convert string into a real number
8	"Present Value" "" INPUT	""	prompt for present value
10		"250"	your input of representing $250
11	OBJ→	250	convert into a real number
12	"Future Value" "" INPUT	""	prompt for future value
13		"1000"	your input representing $1000
14	OBJ→	1000	convert string into a real number
15	→ N I FV PV		save data in local variables
16	«		
17	CASE		
18	'PV==0'	'PV==0'	push condition in stack. The value of variable PV is compared with 0. Since PV is not equal to 0, the condition returns 0. This result is removed from the stack and the next condition is evaluated.
19	'FV==0'	'FV==0'	push condition in stack. The value of variable FV is compared with 0. Since FV is not equal to 0, the condition returns 0. This result is removed from the stack and the next condition is evaluated.

20	'I==0'	'I==0'	push condition in stack. The value of variable I is compared with 0. Since I is equal to 0, the condition returns 0. This result is removed from the stack and the THEN clause is executed.
21	THEN		
22	'I'	'I'	push tag name in stack
23	'(FV/PV)^(1/N)-1'	'(FV/PV...	push algebraic expression in the stack
22	END	'(FV/PV...	program jumps to END of structure
23	EVAL	0.0968	evaluate expression
24	SWAP	'I'	swap tag into level 1
25	→TAG	I:0.0968	tag the result
26	»		
27	»	I:0.0968	end of program.

The QuickBASIC version of the above RPL program is shown below:

```
' QuickBASIC version
CLS
INPUT "Enter interest rate : ";I
I = I / 100
INPUT "Enter investment periods : ";N
INPUT "Enter present value : ";PV
INPUT "Enter future value : ";FV
IF PV = 0 THEN
   PRINT "PV = ";FV/(1+I)^N
ELSEIF FV = 0 THEN
   PRINT "FV = ";PV*(1+I)^N
ELSEIF I = 0 THEN
   PRINT "I = ";(FV/PV)^(1/N) - 1
ELSE
   PRINT "N = ";LOG(FV/PV)/LN(1 + I)
END IF
END
```

Notice that the QuickBASIC version uses a battery of IF and ELSEIF structures instead of a CASE statement. This is due to the fact that the CASE structure in QuickBASIC (and most of other languages) works differently from the CASE structure in RPL. The difference is seen from the general syntax of CASE in QuickBASIC:

```
SELECT CASE variable or expression
        CASE list # 1 of values
                sequence of statements
        CASE list # 2 of values
                sequence of statements

        other case labels

        CASE ELSE
                sequence of statements
END SELECT
```

Thus, the RPL CASE structure allows you to test multiple conditions that are not syntactically related to each other. The QuickBASIC CASE examines the values of a specific variable or expression.

The HP41 version of the above RPL program is shown side by side with the RPL listing:

```
  HP41                          RPL
-------------------------------------------------------
LBL "FINC"                      «
"% INTEREST"                    "% Interest" "" INPUT OBJ→
PROMPT
100                             100
/                               /
STO 00
"PERIODS"                       "Periods" "" INPUT OBJ→
PROMPT
STO 01
"PV"                            "Present Value" "" INPUT OBJ→
PROMPT
STO 02
"FV"                            "Future Value" "" INPUT OBJ→
PROMPT
STO 03
                                → N I FV PV

                                    «
RCL 02                              CASE
X≠0?                                'PV == 0'
GTO 01
                                    THEN
"PV="                                  'PV'
RCL 03                                 'FV/(1+I)^N'
1
RCL 00
+
RCL 02
y^x
GTO 00                              END
LBL 01
```

```
RCL 03                          'FV==0'
X≠0?
GTO 02                          THEN
"FV="                                 'FV'
RCL 02                                'PV*(1+I)^N
1
RCL 00
+
RCL 01
y^x
GTO 00                          END
LBL 02
RCL 01
X≠0?                            'N==0'
GTO 03
                                THEN
"N="                               'N'
RCL 03                             'LN(FV/PV)/LN(1+I)'
RCL 02
/
LN
1
RCL 00
+
LN
/
GTO 00                          END
LBL 03
"I="                            'I'
RCL 03                          '(FV/PV)^(1/N)-1'
RCL 02
/
RCL 01
1/X
y^x
1
-
LBL 00                          END
                                EVAL
                                SWAP
ARCL X                          →TAG
PROMPT
                                »
RTN                         »
```

Once again we see that the HP41 uses an increasing number of GOTOs to implement the logic of a CASE structure. Using algebraic objects with the various clauses of a CASE structure makes the RPL version far more readable than the HP41 version.

Nested Decision-Making Structures

The various decision-making structures can be nested to provide more advanced control logic. The last RPL program that demonstrated the CASE structure has a logical weakness: what if

a user enters 0 for more than one variable? The result is a runtime math error. To remedy this potential problem the CASE structure is placed inside an IF structure that detects multiple zero assignment using the following Boolean expression:

'N==0 XOR I==0 XOR FV==0 XOR PV==0'

The above condition is true if and only if one of the variables is zero. The CASE structure is located in the THEN clause of the outer IF structure. An ELSE clause can be added to warn the user of his or her bad input. The listing of this new program version is show below:

```
; Listing 6.5.  Program that uses the XOR operator in finanical calculations.
«
  "% Interest" "" INPUT OBJ→          ; prompt for interest
   100 /                              ; convert percentage to fraction
  "Periods" "" INPUT OBJ→             ; prompt for investment periods
  "Present Value" "" INPUT OBJ→       ; prompt for present value
  "Future Value" "" INPUT OBJ→        ; prompt for future value
   → N I FV PV                        ; store input in local variables
     «
                                     ; is there only one variable that
                                     ; have a zero value?
     IF 'N==0 XOR I==0 XOR FV==0 XOR PV==0'
     THEN                            ; yes!  proceed in solution
       CASE                          ; start CASE structure
        'P == 0'                     ; is present value 0?
         THEN                        ; Yes.  Solve for present value
             'PV'                    ;   push PV tag
             'FV/(1+I)^N'
         END
        'FV==0'                      ; is future value 0?
        THEN                         ; Yes.  Solve for future value
             'FV'                    ;     push FV tag
             'PV*(1+I)^N'
        END
        'I==0'                       ; is interest 0?
        THEN                         ; Yes.  Solve for interest
             'I'                     ;    push I tag
             '(FV/PV)^(1/N)-1'
        END
                                     ; ******** DEFAULT CLAUSE ********
             'N'                     ; solve for investment periods
             'LN(FV/PV) / LN(1 + I)'
       END                           ; end of CASE structure
       EVAL                          ; evaluate expression in level 1
       SWAP →TAG                     ; swap result and tag, then tag the result
     ELSE                            ; warn user of bad input
       1000 .4 BEEP
       "Bad input!"
     END
     »
»
```

There are no rules on how decision-making structures are nested. The process is subject to the

application's need.

The HP48SX Flags

Most of the Hewlett-Packard programmable calculators support logical flags. Flags are essentially logical switches. A flag is either set (that is, turned on) or clear (that is, turned off). One or more flags are used to preserve or indicate a specific condition or state. There is no explicit equivalent of flags in BASIC. BASIC programs can easily emulate flags by storing 0 or -1 (the bitwise inverse of 0) in integer-typed variables. The HP48SX flags are very similar to those of the HP41. There are, however, the following differences:

- The user defined flags have positive numbers, while the system flags have negative numbers. There is no flag 0!
- To manipulate a flag you must place its number in level 1 of the stack.
- The result of testing the status of a flag is placed in level 1. A 1 indicates that the tested status is true, while a 0 indicates the reverse.
- The command following a test of a flag status is always executed, regardless of the outcome of the test.

The system flags are listed in Appendix E of the HP48SX manual. I will focus on the user-flags. The HP48SX flag commands are shown below:

Flag Command	Action	Examples
SF	set flag	3 SF I SF
CF	clear flag	4 CF J CF
FS?	is flag set?	3 FS? K ABS FS? 2 FS? 3 FS? AND 10 FS? 'I>J' OR
FC?	is flag clear	4 FC? INDEX FC? 1 FC? 4 FC? XOR 'I>0' 'J>0' AND 12 FC? AND
FS?C	clear flag after testing if it is set	6 FS?C FLGIDX FS?C 'I<10' FS?C 3 XOR
FC?C	clear flag after	01 FC?C

```
testing if it is        K FC?C
clear                   1 FC?C 2 FC?C XOR
```

The flag status test can be involved in the conditions of IF-THEN, IF-THEN_ELSE and CASE-END structures. The flag status test contributes a 0 or 1 to the tested condition.

The next program illustrates flags. It calculates the circumference and area of a circle, then prompts you for the following:

- The value of the radius.
- Whether or not you want the results tagged.
- Whether or not you want to print the results.
- Whether or not you want the results returned in a list.

When prompted for the three choices listed above, a default answer Y is provided. Press the [ENTER] key for a yes, or type N and then press [ENTER] for a no. The program uses flags to convert your input into a logical form. These flags are used after the calculation phase to fine-tune the output. The program is listed below;

```
; Listing 6.6.  Program that calculates the circumference and area of a circle.
; Version 1
«
  0 ""                                  ; push dummy data in the stack
  → RADIUS S                            ; create local variables
    «
    "Enter radius" {"1" -1} INPUT       ; prompt for the radius
    OBJ→
    'RADIUS' STO
    "Tag results?" {"Y" -1} INPUT       ; do you want to tag the results?
    'S' STO
    IF 'S' "Y" ==                       ; is answer a Y or a y?
       'S" "y" == OR
    THEN
        1 SF
    ELSE
        1 CF
    END
    "Print results?" {"Y" -1} INPUT     ; want to print results?
    'S' STO
    IF 'S' "Y" ==                       ; is answer a Y or a y?
       'S" "y" == OR
    THEN
        2 SF
    ELSE
        2 CF
    END
    "Store results in a list?"          ; store the results in a list?
    {"Y" -1} INPUT OBJ→
    'S' STO
    IF 'S' "Y" ==                       ; is the answer a Y or a y?
       'S" "y" == OR
    THEN
```

```
    3 SF
ELSE
    3 CF
END
RADIUS 2 * π →NUM                    ; calculate circumference
RADIUS SQ π *                        ; calculate the area
IF 1 FS?C                            ; tag results?
THEN
    'AREA'
    →TAG
    SWAP
    'CIRCUMF'
    TAG→
    SWAP
END
IF 2 FS?C                            ; print results?
THEN
    PR1
    SWAP
    PR1
    SWAP
END
IF 3 FS?C                            ; put results in a list?
    { }
    +
    +
END
»
»
```

The RCLF command recalls the system and user flags by pushing a list of two binary integers in the stack. The first and second numbers in the list represent the current status of the system and user flags, respectively. The STOF command enables you to restore or alter the status of all flags or just the system flags. This command takes one of two arguments from the stack:

- A list of two binary integers. This is similar to the list produced by a RCLF command. The first number sets the system flags, while the second one sets the user flags.

- A binary integer that sets the system flags.

To calculate the value of the binary integer needed to alter the user and system flags, use the following equations:

$$\textit{Binary integer} = \sum_{i=1}^{64} 2^i \qquad \textit{for all flag } i \textit{ that is set}$$

$$\textit{Binary integer} = \sum_{i=-1}^{-64} 2^{|i|} \quad \textit{for all flag } i \textit{ that is set}$$

The last program did not preserve and restore the status of flags 1, 2, and 3. The following is a modified version that uses the RCLF and STOF commands. The list obtained by the RCLF is stored in a new local variable FFLAG. At the end of the program the FFLAG STOF command restores the previous flag status. Notice that the IF structure that tests for the status of the flags now use the FS? test instead of the FS?C. In version 1 of the program I was assuming that the flags 1, 2 and 3 were clear before the program started. Therefore, version 1 uses the FS?C to test and clear the flags to the original values. In version 2 I need not clear these flags because the STOF command takes care of restoring the status of these flags to whatever they were. Version 2 has the strong advantage of NOT ASSUMING ANYTHING about the status of the flags before the program starts. This is the recommended approach for flag management by programs.

Programming Note

Use the RCLF and STOF to save and restore the status of the HP48SX flags at the beginning and at the end of a program object, respectively. A local variable should be used to store the flag status in the case of smooth-running programs. Otherwise, use a global variable so that you can manually restore the flags.

The program is listed below:

```
; Listing 6.7.  Program that calculates the circumference and area of a circle.
; Version 2
«
  RCLF 0 ""                              ; recall status of flags and push dummy data in the stack
  → FFLAG RADIUS S                       ; create local variables
    «
    "Enter radius" {"1" -1} INPUT        ; prompt for the radius
     OBJ→
    'RADIUS' STO
    "Tag results?" {"Y" -1} INPUT        ; do you want to tag the results?
    'S' STO
    IF 'S' "Y" ==                        ; is answer a Y or a y?
       'S" "y" == OR
    THEN
        1 SF
```

```
    ELSE
        1 CF
    END
    "Print results?" {"Y" -1} INPUT ; want to print results?
    'S' STO
    IF 'S' "Y" ==                   ; is answer a Y or a y?
       'S" "y" == OR
    THEN
        2 SF
    ELSE
        2 CF
    END
    "Store results in a list?"      ; store the results in a list?
    {"Y" -1}
    INPUT OBJ→
    'S' STO
    IF 'S' "Y" ==                   ; is the answer a Y or a y?
       'S" "y" == OR
    THEN
        3 SF
    ELSE
        3 CF
    END
    RADIUS 2 * π →NUM               ; calculate circumference
    RADIUS SQ π *                   ; calculate the area
    IF 1 FS?                        ; tag results?
    THEN
        'AREA'
        →TAG
        SWAP
        'CIRCUMF'
        TAG→
        SWAP
    END
    IF 2 FS?                        ; print results?
    THEN
        PR1
        SWAP
        PR1
        SWAP
    END
    IF 3 FS?                        ; put results in a list?
        { }
        +
        +
    END
    FFLAG STOF                      ; restore original status of flags
    »
»
```

Programming Note

I recommend that you reserve a number of flags for the following:

- Program objects that implement logical functions. Such functions would implement, for example, a yes/no prompt.
- Program results that are supplemented by flags. For example, flags can indicate any error that occurred during computation.

I suggest that higher flags be used for this purpose. These flags should not be considered as free flags. Their status need not be stored and restored as is the case with the rest of the user flags.

Notes

7

Loop Structures

Loops enable programs to repeat commands. There are three types of loops: fixed iteration, conditional, and open. Fixed-iteration loops (the HP48SX manual calls them definite loops) iterate for a predetermined number of times. Conditional loops iterate while or until a tested condition is true. Open loops iterate indefinitely. This chapter deals with the HP48SX fixed and conditional loops. While open loops are not explicitly implemented in RPL, they can be easily mimicked by existing loops.

The FOR-NEXT Fixed Loop Structure

The FOR-NEXT loop is among perhaps the most popular loop in many languages. The general syntax for the FOR-NEXT loop is:

start finish
FOR *loop control variable*
    sequence of commands
NEXT

The start and finish are numeric limits that define the number of iterations (which is equal to finish - start + 1). The start and finish parameters are located in levels 2 and 1, respectively, just before the loop start executing. Typically the start parameter is assigned 1 and the finish parameter is assigned the number of iterations. The loop control variable is local to the body of the FOR-NEXT loop and represents a measure of the loop progress. When the loop starts executing, the control variable is assigned the value of start. As the loop reaches the NEXT keyword, the loop control variable is increased by one. The loop reiterates as long as the control variable is less than or equal to the finish value.

> *Programming Note*
>
> If the start parameter is greater than the finish parameter, the loop iterates once. This is because the control variable is compared with the finish parameter after the loop's NEXT is encountered.

This loop structure should be very familiar if you have programmed in any BASIC implementation. The general syntax for the FOR-NEXT loop in BASIC is:

```
FOR loop control variable = start TO finish
      sequence of statements
NEXT
```

The HP41C is able to emulate a FOR-NEXT loop using the following:

bbb.eee

STO *nn*
LBL *mm*
...
...
ISG *nn*
GTO *mm*

The *bbb.eee* is a real number than contains the loops limits. The loop begins with *LBL mm* and ends with the *GTO mm* instruction. The *ISG nn* instruction handles the loop iteration control.

The next program demonstrates the FOR-NEXT loop in a small program that calculates factorials. The program prompts you for an input. The program verifies that your input is not negative before calculating the factorial using a FOR-NEXT loop. The loop body uses the value of the control variable to obtain the factorial. The program is listed below:

```
; Listing 7.1.  Program calculates factorial using FOR-NEXT loop.
«
  "Enter number" "" INPUT OBJ→        ; prompt for input
   DUP                                ; duplicate input for IF structure
   0
   IF >=                              ; is your input equal to or greater than 0?
   THEN                               ; yes.  Calculate factorial

    1 SWAP                            ; push and swap 1 for factorial product
    1 SWAP                            ; push and swap 1 for start parameter of loop
    FOR I                             ; begin FOR-NEXT loop.  I is the control variable
        I *                           ; multiply factorial product by I
    NEXT                              ; end of FOR-NEXT loop
   ELSE                               ; handle bad input
    1000 0.2 BEEP                     ; beep and put a message in level 1
    "Bad input"
   END
»
```

Let's single-step through the above program. The following tracing session is for an input of 3:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	"Enter number" "" INPUT	""	prompt user for input
2		"3"	your input, 3
3	OBJ→	3	convert into real number
4	DUP	3	
5	0	0	
6	IF >=	1	is 3 > 0?
7	THEN	3	
8	1	1	push and swap factorial product
9	SWAP	3	
10	1	1	push and swap loop start
11	SWAP	3	
12	FOR I	1	begin loop
13	I	1	******* 1st iteration
14	*	1	
15	NEXT	1	
16	I	2	******* 2nd iteration
17	*	2	
18	NEXT	2	
19	I	3	******* 3rd iteration
20	*	6	
21	NEXT	6	
22	END	6	exit from loop and jump to end of IF structure
23	»	6	end of program

The QuickBASIC program version is shown below. Except for how the FOR loop is syntactically initialized, notice how similar the FOR-NEXT loops are in BASIC and RPL.

```
' QuickBASIC version
INPUT "Enter number : ";N
IF N >= 0 THEN
  FACT = 1
  FOR I = 1 TO N
      FACT = FACT * I
  NEXT
  PRINT FACT
ELSE
  PRINT "Bad input"
END
```

The HP41C version of the above program is shown below (in a side-by-side comparison with the RPL version):

HP41C	RPL
LBL "FACT"	«
"ENTER NUMBER"	"Enter number" "" INPUT OBJ→
PROMPT	
	DUP
	0
X<0?	IF >=
GTO 02	
	THEN
INT	1 SWAP
1E3	1 SWAP
/	
1	
+	
1	
STO 00	
LBL 01	FOR I
RCL 00	I
INT	
*	*
ISG 00	
GTO 01	NEXT
GTO 10	
LBL 02	ELSE
BEEP	1000 0.2 BEEP
"BAD INPUT"	"Bad input"
PROMPT	
LBL 10	END
RTN	»

Another example of using the FOR loop is one that handles arrays. Fixed loops are suitable structures for handling arrays. The next program object implements a function that returns the average of real elements in a one-dimensional array. The program takes its array argument

from level 1 and pushes the calculated average in the stack. The program is listed below:

```
; Listing 7.2.  Program that calculates the average value of an array.
«
  0 0 → A SUM SUMX                    ; assign array to local variable A and create/initialize
                                      ; the statistical summations
    «
    A SIZE 1 GET                      ; get the array size
    'SUM' STO                         ; store array size in local variable SUM
    1 SUM                             ; define limits of FOR-NEXT loop
    FOR I                             ; begin FOR-NEXT loop, with I as the control
                                      ; variable
        A I GET                       ; obtain the I'th array element
        'SUMX' STO+                   ; update sum of elements
    NEXT
    SUMX SUM /                        ; calculate the average
    'AVERAGE'                         ; push tag in the stack
    →TAG                              ; tag the result
    »
»
```

The FOR-STEP Fixed Loop Structure

The FOR-NEXT loop structure increments the loop control variable by 1 at the end of each iteration. The FOR-STEP loop is a variation of the FOR-NEXT that increases the control variable by a value other than 1. The increment may be positive or negative. Practically, the FOR-NEXT loop is used more frequently than the FOR-STEP loop. The general syntax for the FOR-STEP loop structure is:

start finish
FOR *loop control variable*
    sequence of commands
    increment
STEP

The FOR-STEP loop adds an increment parameter before the STEP keyword. This increment is evaluated the first time it is encountered and is stored. When the loop starts executing, the control variable is assigned the value of start. As the loop reaches the STEP keyword, the loop control variable is increased by the specified increment. For positive increment values, the loop reiterates as long as the control variable is less than or equal to the finish value. For negative increment values, the loop reiterates as long as the control variable is greater than or equal to the finish value.

The BASIC FOR loop supports a STEP clause that makes it work like RPL's FOR-STEP loop. The general syntax for the FOR-NEXT with the STEP clause is:

```
FOR loop control variable = start TO finish STEP increment
      sequence of statements
NEXT
```

The HP41C is able to emulate an upward-counting FOR-STEP loop using the following:

bbb.eeeii

STO *nn*
LBL *mm*
...
...
ISG *nn*
GTO *mm*

The *ii* part in the *bbb.eeeii* number represents the loop increment value.

The HP41C is also capable of emulating an downward-counting FOR-STEP loop using the following:

bbb.eeeii

STO *nn*
LBL *mm*
...
...
DSZ *nn*
GTO *mm*

The next program illustrates the FOR-STEP loop in a small program that calculates factorials. This program resembles the FOR-NEXT version, except a downward-counting loop is used to obtain the factorial. The program prompts you for an input. The program verifies that your input is not negative before calculating the factorial using a FOR-START loop. The loop body uses the value of the control variable to obtain the factorial. The program is listed below:

```
; Listing 7.3.  Program calculates factorial using FOR-STEP loop, version 2.
«
  "Enter number" "" INPUT OBJ→       ; prompt for input
   DUP                               ; duplicate input for IF structure
   0
   IF >=                             ; is your input equal to or greater than 0?
   THEN                              ; yes.  Calculate factorial

    1 SWAP                           ; push and swap 1 for factorial product
    1                                ; push 1 for start parameter of loop
    FOR I                            ; begin FOR-STEP loop.  I is the control variable
        I *                          ; multiply factorial product by I
        -1                           ; the loop increment value
```

```
    STEP                                    ; end of FOR-STEP loop
   ELSE                                     ; handle bad input
    1000 0.2 BEEP                           ; beep and put a message in level 1
    "Bad input"
   END
»
```

Let's single-step through the above program. The following tracing session is for an input of 3:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	"Enter number" "" INPUT	""	prompt user for input
2		"3"	your input, 3
3	OBJ→	3	convert into real number
4	DUP	3	
5	0	0	
6	IF >=	1	is 3 > 0?
7	THEN	3	
8	1	1	push and swap factorial product
9	SWAP	3	
10	1	1	push loop start
11	FOR I	1	begin loop
12	I	3	******* 1st iteration
13	*	3	
14	NEXT	3	
15	I	2	******* 2nd iteration
16	*	6	
17	NEXT	6	
18	I	1	******* 3rd iteration
19	*	6	
20	NEXT	6	

```
21          END                     6               exit from loop and jump
                                                    to end of IF structure

22          »                       6               end of program
```

The QuickBASIC program version is shown below:

```
' QuickBASIC version
INPUT "Enter number : ";N
IF N >= 0 THEN
  FACT = 1
  FOR I = N TO 1 STEP -1
      FACT = FACT * I
  NEXT
  PRINT FACT
ELSE
  PRINT "Bad input"
END
```

The HP41C version of the above program is shown below (in a side-by-side comparison with the RPL version):

HP41C	RPL
LBL "FACT"	«
"ENTER NUMBER"	"Enter number" "" INPUT OBJ→
PROMPT	
	DUP
	0
X<0?	IF >=
GTO 02	
	THEN
0.001	1 SWAP
+	1
STO 00	
LBL 01	FOR I
RCL 00	I
INT	
*	*
DSE 00	-1
GTO 01	STEP
GTO 10	
LBL 02	ELSE
BEEP	1000 0.2 BEEP
"BAD INPUT"	"Bad input"
PROMPT	
LBL 10	END
RTN	»

Another example of the FOR-STEP loop performs the numerical integration using Simpson's rule:

$$\int_{X_0}^{X_n} y\,dx = (\Delta X/3)[\,Y_1 + 4Y_2 + 2Y_3 + \ldots + 2Y_{n-2} + 4Y_{n-1} + Y_n\,]$$

Simpson's rule calculates the approximate area of an odd number of function values taken at regular x intervals. The array of y and the value of Δx must be supplied in levels 2 and 1, respectively. The program pushes the result in the stack. A single FOR-STEP loop is used to obtain separate sums for the array elements with odd and even indices. The FOR-STEP loop is incremented by 2. Its limits are between 2 and the size of an array. The program is listed below:

```
; Listing 7.4.  Program that uses Simpson's method to integrate the function y(x)
«
  0 0 0 0 → Y DX SUMODD SUMEVEN      ; store array and Δx in local variables
            AREA ARSIZE              ; assign 0 to sum of even and odd terms
    «
    A SIZE 1 GET                     ; obtain the array size
    'ARSIZE' STO                     ; store array size in the local variable ARSIZE
    A 1 GET                          ; get the first array element
    A ARSIZE GET                     ; get the last array element
    -                                ; store the difference in variable AREA
    'AREA' STO
    2 ARSIZE                         ; define the FOR-STEP loop limits
    FOR I                            ; begin FOR loop
        A I GET                      ; obtain I'th element
        'SUMEVEN' STO+               ; update the even-terms summation
        A 'I+1' EVAL GET             ; obtain (I+1)'th element
        'SUMODD' STO+                ; update the odd-terms summation
        2                            ; set the loop increment
    STEP
    4 'SUMEVEN' STO*                 ; multiply the even-terms sum by 4
    2 'SUMODD' STO*                  ; multiply the odd-terms by 2
    'SUMEVEN + SUMODD' EVAL          ; get the sum of even and odd terms
    'AREA' STO+                      ; add to variable AREA
    DX 3 /                           ; calculate the approximate area
    'AREA' STO*
    AREA
    'AREA'                           ; push the tag in the stack
    →TAG                             ; tag the result
    »
»
```

If you compare the code of the above program with the above equation you discover that the following revised form of Simpson's equation is implemented:

$$\int_{X_0}^{X_n} y\,dx = (\Delta X/3)[\,Y_1 - Y_n + 4Y_2 + 2Y_3 + \ldots + 2Y_{n-2} + 4Y_{n-1} + 2Y_n\,]$$

This allows a single FOR-NEXT loop to obtain the sum of an equal number of odd and even terms that are later multiplied by 2 and 4, respectively.

Manipulating FOR Loop Iteration

By manipulating the FOR loop control variable a program can reduce or extend the originally planned number of iterations. While this type of action seems to defeat the purpose of having a predetermined number of iterations, there are valid cases. The most prominent circumstance is handling critical conditions. When such conditions arise they require that the loop stops iterating. The technique is rather simple --- just assign the value of the loop's *finish* parameter to the loop control variable. This makes the loop iteration mechanism think that its done.

The next program is an example that manipulates a loop control variable to exit from FOR-NEXT loop. The program object is a function that searches for a number in an array. The input to the function is the array in level 2 and the search element is level 1. The function output is pushed in the stack and represents the index of the first matching array element, or 0 if no match is found.

```
; Listing 7.5.  Array search function.
; Version 1
«
  0                                   ; push zero to reserve a space for the local variable
                                      ; LEN
  → ARY ELM LEN                       ; store data in local variables:
                                      ;     array in ARY
                                      ;     element in ELM
                                      ;     0 in LEN
    «
    1 CF                              ; clear flag 1 which is used to indicate if a match
                                      ; is found
    ARY SIZE 1 GET                    ; obtain the size of the array
    'LEN' STO                         ; store array size in local variable LEN
    1 LEN                             ; specify the limits of the FOR-NEXT loop
    FOR I                             ; begin FOR-NEXT loop with I as the control variable
        ARY I GET                     ; obtain the I'th array element
        ELM                           ; recall the search element
        IF == THEN                    ; is I'th array member equal to search element?
            I                         ; yes!  push result in stack
            LEN 'I' STO               ; set loop control variable to the loop's upper limit
            1 SF                      ; set found flag
        END
    NEXT
    »
  IF 1 FC?C THEN                      ; no match found?
    0                                 ; then push 0 in the stack
  END
»
```

The START-NEXT and START-STEP Fixed Loop Structures

The RPL language implements a second class of fixed loops, namely, the START-NEXT and START-STEP loops. These loops are very similar to the FOR-NEXT and FOR-STEP loops. The main difference is that the START loop does not let you define and use a loop control variable. In reality, these loops do use control variables, but a program is denied their access. The START loops are not as common as FOR loops among popular languages. What purpose do they serve? The START loops are used to repeat sequences of commands that do not require loop control variables. The general syntax for the START-NEXT loop is:

start finish
START
    sequence of commands
NEXT

The next program illustrates using the START-NEXT loop to obtain the basic statistics for the built-in random number generator. The program requires that the number of iterations be in level 1. This type of application does not require the use of a loop control variable, making the START-NEXT loop an ideal candidate. The two tagged results are the mean value and the standard deviation in levels 2 and 1, respectively. The theoretical results for the mean and standard deviation are 0.5 and 0.28, respectively. The program is shown below:

```
; Listing 7.6. Program that obtains the basic statistics of the built-in random
; number generator.
; Version 1
«
  0 0 0 → SUM SUMX SUMXX X          ; initialize statistical summations and reserve space for
                                    ; the auxiliary local variable X
  «
  1 SUM                             ; set limits for START-NEXT loop
  START
    RAND                            ; obtain a random number between 0 and 1
    'X' STO                         ; store random number in X
    X 'SUMX' STO+                   ; update sum of X
    X SQ 'SUMXX' STO+               ; update sum of X squared
  NEXT
  'SUMX / SUM' EVAL                 ; calculate the mean
  'MEAN' →TAG                       ; push tag in the stack
                                    ; calculate the standard deviation
  '√((SUMXX - SUMX * SUMX / SUM) / (SUM - 1))' EVAL
  'SDEV' →TAG                      ; push the tag in the stack
  »
»
```

The START-STEP loop is a variation of the START-NEXT loop. Like the FOR-STEP loop, the START-STEP uses an increment that is added to the internal loop counter. The general syntax for the START-STEP loop is:

start finish
START
    sequence of commands
    loop increment
STEP

The DO-UNTIL Conditional Loop Structure

The RPL language supports two conditional loops, namely the DO-UNTIL and the WHILE-REPEAT loops. The DO-UNTIL is a conditional loop that iterates until a tested condition is true. The general syntax for this loop is:

DO
    sequence of commands
UNTIL *condition* **END**

Since the tested condition is located at the end of the loop, the DO-UNTIL loop executes at least once. It is therefore used when the loop's command must be executed at least once. The HP48SX DO-UNTIL should be familiar to you if you have programmed in QuickBASIC. The QuickBASIC DO-UNTIL loop has the following general syntax:

```
REM *** QuickBASIC ***
DO
    sequence of statements
LOOP UNTIL condition
```

By contrast, the HP41C does not explicitly support conditional loops. The logic of a DO-UNTIL loop is implemented, in general, as follows:

```
LBL mm
sequence of commands
reverse condition
test
GTO mm
```

The scope of the DO-UNTIL loop on the HP41C is defined by the label and GTO command. The condition tested must be the reverse of that in RPL.

The first example of using a DO-UNTIL is a very simple application of Newton's algorithm. The program calculates the square root of a positive number, N, using the following algorithm:

$$X_{n+1} = (N / X_n + X_n) / 2$$

The program takes the number from level 1 and returns the square root (8 decimal laces accurate) in level 1. The initial guess for the square root is taken as N/2 and is provided by the program itself. The program is listed below:

```
; Listing 7.7. Program to iteratively solve for the square root using Newton's
; algorithm and a DO-UNTIL loop
«
  ABS DUP 2 / → N X                    ; store number in local variable N
                                       ; store N/2 the initial guess for the square root in
                                       ; the local variable X
    «
    DO                                 ; begin the DO-UNTIL loop
       '(N/X+X)/2' EVAL                ; refine the guess for the square root
       'X' STO                         ; store refined guess back in X
    UNTIL 'ABS(X*X-N) < 1E-8' EVAL     ; is the refined guess accurate enough?
    END
    X                                  ; push guess in the stack
    »
»
```

Let's trace this program to obtain the square root of 25. I am using a perfect square so that it becomes easier to get a feel of how close we're getting to the answer in each iteration. The use of algebraic expressions bypasses the secondary intermediate results and focuses on the primary ones. To single step the program, enter 5 and the name of the variable storing the program in the stack. Press the [PRG] key and select the CRTL and DBUG menu options. Use the SST menu option to single-step through the program. The program tracing proceeds as follows:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	ABS	25	take the absolute value of the square
2	DUP	25	
3	2	2	
4	/	12.5	get the initial guess for the square root
5	→ N X		assign the data to local variables
6	«		
7	DO		
8	'(N/X+X)/2' EVAL	7.25	evaluate refined guess
9	'X' STO		store refined guess
10	UNTIL		
11	'ABS(X*X-N) < 1E-8' EVAL	0	evaluate loop condition

```
12          END
13          '(N/X+X)/2' EVAL              5.3491379     evaluate refined guess
14          'X' STO                                     store refined guess
15          UNTIL
16          'ABS(X*X-N) < 1E-8' EVAL      0             evaluate loop condition
17          END
18          '(N/X+X)/2' EVAL              5.0113941     evaluate refined guess
19          'X' STO                                     store refined guess
20          UNTIL
21          'ABS(X*X-N) < 1E-8' EVAL      0             evaluate loop condition
22          END
23          '(N/X+X)/2' EVAL              5.000012953   evaluate refined guess
24          'X' STO                                     store refined guess
25          UNTIL
26          'ABS(X*X-N) < 1E-8' EVAL      0             evaluate loop condition
27          END
28          '(N/X+X)/2' EVAL              5             evaluate refined guess
29          'X' STO                                     store refined guess
30          UNTIL
31          'ABS(X*X-N) < 1E-8' EVAL      1             evaluate loop condition
                                                        since the condition is
                                                        1, the loop will stop
                                                        iterating.
32          END
33          X                             5             push result in the stack
34          »                             5
35          »
```

The QuickBASIC version is shown below:

```
INPUT "Enter a number : ";N
N = ABS(N)
X = N / 2
DO
  X = (N / X + X) / 2
UNTIL ABS(X*X-N) < 1E-8
PRINT X
END
```

The HP41C version is presented below with a side-by-side comparison to the RPL version:

HP41C	RPL
LBL "SQQRT"	«
ABS	ABS
STO Y	DUP
2	2
/	/
STO 00	→ N X
RDN	
STO 01	
	«
LBL 00	DO
RCL 01	'(N/X+X)/2' EVAL
RCL 00	
/	
RCL 00	
+	
2	
/	
STO 00	'X' STO
STO *X	UNTIL 'ABS(X*X-N) < 1E-8' EVAL
RCL 01	
-	
ABS	
1E-8	
x<=y?	
GTO 00	END
RCL 00	X
	»
RTN	»

The above HP41C program implements the DO-UNTIL logic using the commands between LBL 00 and GTO 00. The test located before GTO 00 checks whether or not the absolute value of (XX*-N) is greater or equal to the error tolerance of 1E-8. This test is the reversed version (or logically negative, if you prefer) of the one in the RPL version.

The second example of the DO-UNTIL loop is a program object that implements a Yes/No

response function. You should find this function very useful in your own programs. The function takes a prompt message from the stack. An INPUT command is used to prompt you the message and a default answer "Y". A DO-UNTIL loop ensures that your answer begins with the letters Y or N. Flag 64 is set if your answer begins with a Y. The same flag is cleared if your answer begins with an N. The program is listed below:

```
; Listing 7.8.  Yes/No function.
;
; INPUT:
;        Message in level 1
;
; USER INPUT; Yes/No answer which includes Y, N, y, n, YES, Yes, No, and no.
;
; OUTPUT:
;        flag 64 is set   => answer was positive
;        flag 64 is clear => answer was negative
;
«
  ""
  → MSG ANSWER                          ; store prompt message in variable MSG
    «
    DO                                  ; begin DO-UNTIL loop
        MSG {"Y" -1} INPUT              ; prompt user with default answer "Y"
        'ANSWER' STO                    ; store input in variable ANSWER
        ANSWER SIZE 0                   ; is the size of the input string > 0?
        IF >
        THEN                            ; yes!  Obtain the first input character
            ANSWER 1 1 SUB
        ELSE
            " "                         ; no! push a space in the stack
        END
        'ANSWER' STO                    ; store level 1 in ANSWER
    UNIT ANSWER "Y" ==                  ; DO loop condition:
         ANSWER "y" == OR               ;     is ANSWER a Y, y, N, or n?
         ANSWER "N" == OR
         ANSWER "n" == OR
    END
    IF ANSWER "Y" ==                    ; is ANSWER a Y or y?
       ANSWER "y" == OR
    THEN                                ; yes.  Set flag 64
        64 SF
    ELSE
        64 CF                           ; no.  Clear flag 64
    END
    »
»
```

The third example of the DO-UNTIL is a program that solves for the root of a function using Newton's algorithm:

$$X_{n+1} = X_n - f(X_n) / f'(X_n)$$

The $f(X)$ and the $f'(X)$ are the target function and its first derivative with respect to X. The

derivative can be approximated using the following:

$$f'(X) \approx (f(X + h) - f(X - h)) / 2h$$

where h is a small increment of X. I use the following rules to obtain h:

$$h = 0.01 \qquad \text{if } |X| \leq 1$$
$$h = 0.01\ X \qquad \text{if } |X| > 1$$

The program requires that the initial guess, the error tolerance, and the name of the variable storing the function be placed in levels 3, 2, and 1, respectively. The program can iterate up to 50 times. If after 50 times no solution is found, flag 1 is set. The program is listed below.

```
; Listing 7.9.  Program solves for the root of a function using Newton's method.
; Version 2
; INPUT:
;           guess for root
;           error tolerance
;           name of variable storing target function
;
; OUTPUT:
;           refined root guess
;           flag 1
;            set   => no solution was found
;            clear => a solution was found
«
  1 CF                                  ; clear flag 1
  50 0 0 0                              ; push the maximum number of iterations in the stack
                                        ; push dummy data
  → X TOLER FX MAXITER ITER H DIFF      ; assign data to local variables:
                                        ; initial guess to variable X
                                        ; error tolerance to variable TOLER
                                        ; name of variable storing target function to variable FX
                                        ; 50 to variable MAXITER
                                        ; 0 to the iteration counter ITER
                                        ; 0 (dummy data) to increment variable H
                                        ; 0 (dummy data) to guess improvement variable DIFF

    «
    DO                                  ; start Newton's iterations
       .01                              ; push 0.01 in the stack
       IF 'ABS(X)>1'                    ; is the absolute value of X greater than 1?
       THEN                             ; yes.  calculate increment as 0.01 * X
          X *
       END
       'H' STO                          ; store increment value in H
       2 H *                            ; calculate guess refinement
       X FX EVAL *                      ;     calculate f(X)
       X H + FX EVAL                    ;     calculate f(X+h)
       X H - FX EVAL                    ;     calculate f(X-h)
       -
       /
```

```
        DUP                                ; duplicate guess refinement
        'DIFF' STO                         ; store refinement in variable DIFF
        X SWAP -                           ; calculate refined guess and store in X
        'X' STO
        'ITER' INCR DROP                   ; increment iteration counter
                                           ; test loop condition:
                                           ;   1) is absolute value of guess refinement
                                           ;      less than error tolerance?
                                           ; OR
                                           ;   2) has the number of iteration reached
                                           ;      or exceeded the maximum allowable
                                           ;      limit?
    UNTIL 'ABS(DIFF) < TOLR OR ITER ≥ MAXITER'

    END
    IF 'ITER ≥ MAXITER'                    ; solution diverged?
    THEN                                   ; yes.  Set flag 1
        1 SF
    END
    X                                      ; push latest guess for the root in the stack
    'Root'                                 ; tag the result
     →TAG
    »
»
```

The above program includes the INCR command used to increment the loop iteration counter ITER. This command takes the name of the variable, increments its value, and pushes the new value in the stack. Since the program needs to merely increment ITER, a DROP command is used to promptly remove the updated increment from the stack.

The DECR command decrements the values stored in variables acting as counters. This command works like INCR by taking the name of the variable from level 1, decrementing it, and pushing the new value back to the stack.

The WHILE-REPEAT Conditional Loop Structure

The WHILE-REPEAT loop iterates as long as a tested condition is true. The general syntax for the WHILE-REPEAT is:

WHILE *condition* **REPEAT**
    sequence of commands
END

Since the tested condition comes before the loop's commands, the WHILE-REPEAT loop may not iterate if the condition is false. This is an important difference between the WHILE-REPEAT and DO-UNTIL loops.

Many programmers feel that the WHILE loop is *the* loop and all other loops are variations of it. These programmers claim that they can replace any loop with the WHILE loop. This claim is

technically valid. The reason other loops exist is that they offer a more readable and more logical form. The following general syntax shows how a WHILE-REPEAT loop works like a FOR-DO loop:

```
FOR-NEXT loop                   WHILE-REPEAT loop

start finish                    start 'control_variable' STO
FOR control_variable            WHILE 'control_variable ≤ finish' REPEAT
      sequence of commands            sequence of commands
                                      1 'control_variable' STO+
NEXT                            END
```

And similarly, the next general syntax shows how a WHILE-REPEAT loop emulates a FOR-STEP loop:

```
FOR-NEXT loop                   WHILE-REPEAT loop

start finish                    start 'control_variable' STO
FOR control_variable            WHILE 'control_variable ≤ finish' REPEAT
      sequence of commands            sequence of commands
      increment                       increment 'control_variable' STO+
STEP                            END
```

The WHILE-REPEAT loop can also mimic the DO-UNTIL loop, as shown below:

```
DO-UNTIL loop                   WHILE-REPEAT loop

DO                              WHILE condition is false REPEAT
      sequence of commands            sequence of commands
UNTIL condition is true
END                             END
```

The ability to replace the various types of loops with the WHILE-REPEAT loop occasionally becomes useful. Most common cases for such a substitution arise when you utilize a FOR-NEXT loop --- you start with the FOR loop and realize that there is an additional condition (aside from the loop's start and finish parameters) that dictates whether or not the loop iterates. The WHILE-REPEAT loop comes in handy.

The WHILE-REPEAT loop should be familiar to you if you have used QuickBASIC, BASICA, or GW-BASIC. The general syntax for the WHILE-WEND loop available to these BASIC implementation is:

```
WHILE condition
      sequence of statements
WEND
```

QuickBASIC offers a second form of WHILE loop:

```
DO WHILE condition
      sequence of statements
LOOP
```

Concerning the HP41C, there is no formal WHILE loop. Using tests, GOTOs, and labels, you can easily implement the logic of a WHILE loop. This is shown by the following general syntax:

```
LBL mm
reverse condition
test
GTO nn
sequence of commands
GTO mm
LBL nn
```

Notice that the logic of the WHILE loop uses twice as many GOTOs and labels as does that of the DO-UNTIL loop.

The first example of using the WHILE-REPEAT loop is a number guessing game. Using the random number generator, the program produces a number between 0 and 99. The program prompts you to guess the number. As you enter guesses that do not match the number, the program tells you if your guess was high or low. You are allowed 10 trials. The use of the WHILE loop handles the case where your first guess is the lucky one. In this instance, the program needs not give you a hint for your next guess! Also notice that while INPUT commands are used inside and outside the WHILE loop, the prompt messages are different. This justifies using the INPUT command located before the WHILE loop. The program is listed below:

```
; Listing 7.10.  A number guessing game that uses a WHILE-REPEAT loop.
«
  100 RAND * IP                          ; generate number to be guessed
  9                                      ; the number of maximum iterations minus one
  0                                      ; dummy data
  → NMBR MAX GUESS                       ; assign data to local variables
    «
    "Enter guess" "" INPUT OBJ→          ; first prompt for guess
    'GUESS' STO                          ; store first guess in variable GUESS
    WHILE                                ; start WHILE loop
        'GUESS ≠ NMBR AND MAX > 0'       ; test condition:
                                         ;   1) is guess not equal to number?
                                         ; AND
                                         ;   2) did you run out of number of tries?
    REPEAT
        'M' DECR DROP                    ; decrement iteration counter
        CASE                             ; use CASE-END to examine current guess
              'GUESS > NMBR'             ; your guess is high
            THEN
              "Guess is high" ""
              INPUT OBJ→                 ; prompt for lower value
              'G' STO
            END
              'GUESS < NMBR'             ; your guess is low
            THEN
              "Guess is low" ""
              INPUT OBJ→                 ; prompt for a higher guess
              'G' STO
            END
```

```
        END                              ; *** END of CASE structure ***
    END                                  ; *** END of WHILE loop ***
    IF 'GUESS == NMBR'                   ; did you guess the number?
    THEN                                 ; yes!
        "Good guess!"
    ELSE
        "Number is " N +                 ; nooooo
    END
    »
»
```

The following is a sample tracing session. I use the word *sample session* to indicate that it is not likely to be repeated easily due to the use of random numbers. As you single-step through the program you will be able to see the number created by the program. Enter a high guess, a low guess, and then the number itself. This makes your session somewhat parallel to the one shown below:

Step #	Statement executed/ Object evaluated	Value in level 1	Comment
1	100 RAND * IP	30	generated the number to be guessed
2	9	9	number of trial - 1
3	0	0	
4	→ NMBR MAX GUESS		
5	«		
6	"Enter guess" "" INPUT	"50"	prompt for input. My input was "50"
7	OBJ→	50	convert to real number
8	'GUESS' STO		
9	WHILE		
10	'GUESS ≠ NMBR AND MAX ≥ 0'		test loop condition
11	REPEAT		
12	'MAX' DECR	8	
13	DROP		
14	CASE		
15	'GUESS > NMBR'		is guess > number?
16	THEN		yes. Hint at next guess

```
17      "Guess is high" "" INPUT     "25"        my input was "25".
18      OBJ→                          25         convert to real number
19      'G' STO
20      END                                      END of THEN clause
21      END                                      END of CASE structure
22      END                                      END of WHILE loop
23      'GUESS ≠ NMBR AND MAX ≥ 0'               test loop condition
24      REPEAT
25      'M' DECR                      7
26      DROP
27      CASE
28      'GUESS > NMBR'                           is guess > number?
29      THEN
30      END                                      no!
31      'GUESS < NMBR'                           is guess < number?
32      THEN
33      "Guess is low" "" INPUT      "30"        yes.  Hint at next guess.
                                                 "30" was my input
34      OBJ→                          30         convert to real number
35      'G' STO
36      END                                      END of THEN clause
37      END                                      END of CASE structure
38      END                                      END of WHILE loop
39      'GUESS ≠ NMBR AND MAX ≥ 0'               test loop condition
40      END                                      loop terminates, since
                                                 condition is false
41      IF                                       do I guess the number?
42      'GUESS == NMBR'
43      THEN                                     yes!
```

```
44        "Good guess!"
45        END
46        »
47        »
```

The QuickBASIC version is shown below:

```
NUMBER = RND(100)
MAX = 9
INPUT "Enter guess : ";GUESS
WHILE (GUESS <> NUMBER) AND (MAX >= 0)
  MAX = MAX - 1
  IF GUESS > NUMBER THEN
      INPUT "Enter lower guess : ";GUESS
  ELSEIF GUESS < NUMBER THEN
      INPUT "Enter higher guess : ";GUESS
  END IF
WEND
IF GUESS = NUMBER THEN
  PRINT "Good guess!"
ELSE
  PRINT "Number is ";NUMBER
END IF
END
```

The HP41C version of the number guessing game is show below along with the RPL version:

HP41C	RPL
LBL "GAME"	«
100	100
XEQ "RNG"	RAND
*	*
INT	IP
STO 00	
9	9
STO 01	
	0
	→ NMBR MAX GUESS
	«
"GUESS?"	"Enter guess" ""
PROMPT	INPUT OBJ→
STO 02	'GUESS' STO
LBL 00	WHILE
RCL 02	'GUESS ≠ NMBR AND MAX > 0'
RCL 00	
x=y?	
GTO 10	
RCL 01	
x=0?	
GTO 10	

```
DSE 01
STO X

RCL 00
RCL 02
x<=y?
GTO 01

"GUESS HI"
PROMPT
STO 02

LBL 01
x=y?
GTO 03

"GUESS LO"
PROMPT
STO 02

LBL 03
GTO 00
"NUMBER IS "
ARCL 00
RCL 00
RCL 02
x=y?
"GOOD GUESS"
PROMPT

RTN
```

```
REPEAT
  'M' DECR DROP

  CASE
    'GUESS > NMBR'


    THEN
      "Guess is high" ""
      INPUT OBJ→
      'G' STO
    END

    'GUESS < NMBR'

    THEN
      "Guess is low" ""
      INPUT OBJ→
      'G' STO
    END
  END
END
IF 'GUESS == NMBR'

THEN
  "Good guess!"
ELSE
  "Number is " N +
END
»
»
```

The HP41C version assumes that the machine has a program labeled RNG to generate the random numbers. The WHILE loop is defined by the commands between LBL 00 and GTO 00. The logic of the CASE structure in the HP41C version lacks a defined start, but ends with LBL 03. The way that the HP41C displays the result does not translate the IF structure of the RPL version.

The second example of using the WHILE-REPEAT loop is a version of the array search function shown earlier. This version uses the WHILE loop to search for the first element that matches a given number. The condition of the WHILE loop is made up of two ANDed subconditions:

1) The array index must be less than or equal to the array's size.
2) The search flag must be clear, indicating that no match has been found yet.

This new version implements the linear search algorithm that works with un-unordered arrays and lists. The program is shown below:

```
; Listing 7.11.  Array search function.
; Version 2
«
  0 0                                   ; push zero to reserve a space for the local variable LEN
```

```
  → ARY ELM LEN I                    ; store data in local variables:
                                     ;     array in ARY
                                     ;     element in ELM
                                     ;     0 in LEN
                                     ;     0 (dummy) in search index I
  «
    1 CF                             ; clear flag 1 which is used to indicate if a match
                                     ; is found
    ARY SIZE 1 GET                   ; obtain the size of the array
    'LEN' STO                        ; store array size in local variable LEN
    1 'I' STO                        ; assign 1 to the search index variable I
    WHILE                            ; begin WHILE-REPEAT loop
        'I ≤ LEN'                    ; is search index within legal limit?
        1 FC?                        ; is flag 1 clear?
        AND                          ; are both of the above conditions true?
    REPEAT
        ARY I GET                    ; obtain the I'th array element
        ELM                          ; recall the search element
        IF == THEN                   ; is I'th array member equal to search element?
            1 SF                     ; yes!  Set flag 1 (the found flag)
        END
        'I' INCR DROP                ; increment the search index
    END
  »
  IF 1 FC?C THEN                     ; no match found?
    0                                ; then push 0 in the stack
  END
»
```

Nesting Loops

Non-trivial applications that perform repetitive tasks often use nested loops. Loops can be nested in any combination. The only rule that must be observed is that nested loops cannot overlap. Instead, they must be contained within each other. In this section I will show two examples. The first nests two FOR-NEXT loops to assign data to a matrix. The second example nests three loops, namely, a WHILE-REPEAT, a DO-UNTIL, and a FOR loop to sort a list of strings.

The first program searches the element of a matrix to determine the largest number. A pair of nested FOR-NEXT loops are used in this search. Nested FOR-NEXT loops are typically used when handling matrices. The variable storing the largest element is initialized with the value of the matrix element in the first row and column. The nested loops compare the matrix elements with the current value of the local variable BIG. The program is shown below:

```
; Listing 7.12.  Program that returns the largest element in a matrix.
«
  DUP SIZE DUP                       ; duplicate matrix and obtain its size
  1 GET                              ; obtain the number of rows
  2 GET                              ; obtain the number of columns
  0                                  ; push a dummy value
  → M ROWS COLS BIG                  ; assign data to local variables:
                                     ; Matrix in variable M
```

```
                                    ; Number of rows in variable ROWS
                                    ; Number of columns in variable COLS
                                    ; 0 (dummy value) in variable BIG
    «
    M { 1 1 } GET                   ; get M[1,1]
    'BIG' STO                       ; store it in BIG
    1 ROWS                          ; define limits of FOR loop that scan the matrix rows
    FOR I
        1 COLS                      ; define limits of FOR loop that scan the matrix columns
        FOR J
            M                       ; push matrix M in the stack
            { } I + J +             ; get M[I,J]
            GET
            DUP                     ; duplicate matrix element
            BIG                     ; get BIG
            IF >                    ; is M[I,J] > BIG?
            THEN                    ; yes.  Set BIG = M[I,J]
                'BIG' STO
            ELSE
                DROP                ; drop M[I,J] from the stack
            END
        NEXT
    NEXT
    BIG                             ; push BIG in the stack
    »
»
```

The second program sorts a list of strings or real numbers in an ascending order. The program takes the target list from level 1 and returns the sorted list in the stack. The Shell-Metzner method is used. This uses cycles of comparisons. These cycles start by comparing distant elements and swapping them if needed. Once the elements of a cycle are in order, the algorithm moves to the next cycle to compare closer elements. The last cycle compares neighboring elements. Once it is finished, the list is in order. The program is listed below:

```
; Listing 7.13. Program that uses the Shell-Metzner algorithm to sort a
; list of strings or numbers
«
  DUP SIZE DUP 0                    ; obtain list size and duplicate it
  → L LEN OFS I                     ; store data in local variables:
                                    ; List in variable L
                                    ; The list size in variable LEN
                                    ; The comparison offset in variable OFS
                                    ; 0 (dummy) data in I
    «
    WHILE                           ; start WHILE-REPEAT loop
          'OFS > 1'                 ; loop while offset is greater than 1
    REPEAT
        OFS 2 / 'OFS' STO           ; reduce the offset by half
        1 CF                        ; clear swap flag
        DO                          ; start DO-UNTIL loop
            1 LEN OFS -             ; assign limits for the FOR-NEXT loop
            FOR J                   ; J is the leading list element
                J OFS +             ; calculate the index I for the trailing list element
                'I' STO
```

```
                L J GET                 ; get the J'th list element
                L I GET                 ; get the I'th list element
                DUP2                    ; duplicate both elements
                IF >                    ; test if J'th element > I'th element
                THEN                    ; yes.  Swap elements
                    L SWAP              ;    insert and swap list
                    J SWAP              ;    insert and swap index J
                    PUT                 ;    put the I'th element in location J
                                        ; *** NOTE: the list is still in the stack ***
                    SWAP                ;    swap list and J'th element
                    I SWAP              ;    insert and swap index I
                    PUT                 ;    put J'th element in location I
                    'L' STO             ; updated the list variable L
                    1 SF                ; set the swap flag
                ELSE                    ; no. Just drop the objects in levels 1 and 2
                    DROP2
                END
            NEXT                        ; *** END of FOR loop ***
        UNTIL 1 FC?C                    ; test condition of DO loop.  Is swap flag clear?
        END                             ; *** END of DO loop ***
    END                                 ; *** END of WHILE loop ***
    L                                   ; push sorted list in the stack
    »
»
```

Open Loops: Who Needs Them?

Open loops iterate indefinitely! This statement seems to convey more of a warning than a description. Why would any application want to use an open loop? The answer is that there are indeed applications that put open loops to very good use. Consider the operating system of the HP48SX (or any other computer). Such an operating system is nothing but a program that keeps on looping. In each iteration, the machine awaits your instructions and executes them when they are available. Aha! Now open loops make sense, don't they? Just think if operating systems did not use any open loops --- you have to turn the power off and then on every time you want to execute a command!

Practically, open loops come with mechanisms that allow a program to exit such loops if needed. The RPL language does not implement open loops, but they are not hard to emulate. A WHILE-REPEAT loop can easily accommodate such a need. The following is a general syntax for using a WHILE loop as a true-blue open loop:

```
WHILE '1 = = 1'
REPEAT
      sequence of commands
END
```

The test '1 = = 1' can be replaced by any other test that is always true. The only way to stop such a loop on the HP48SX is to press the attention key.

Notes

8

Error Handling

In the previous chapters the program examples used little or no measures to handle runtime errors. The general assumption that a program will run without errors is probably confined to only the most trivial programs. Programs that cannot handle anticipated errors lack the professional look. In this chapter I will discuss the error-handling mechanism of the HP48SX.

The RPL language implements an error-handling mechanism that conforms to structured programming. This allows your program applications to handle a variety of mishaps --- from bad input to computational error. There are two basic categories of error-handling methods: defensive programming and error trapping. Defensive programming works by using ordinary decision-making structures (mostly the IF structure) to detect error-generating conditions. Thus, the defensive programming essentially applies preventive measures. The error trapping method is more bold in its approach. Its basic strategy is the following:

- Set an error trap.
- Let the error happen.
- Apply the trap to handle the error.

This enables the program to continue running even if an error has occurred. Compared to the defensive programming approach, the error trapping method is superior.

HP48SX errors are associated with a numeric code and an error message string. This information is managed by a number of commands that enable you to set, reset, and query errors. They are:

1) The ERR0 command. This resets the error code and error message to 0 and a null string, respectively.

2) The ERRN command. It returns the error code associated with the last runtime error. A #0 means either that there has been no error so far, or that the error code was cleared using the ERR0 command.

3) The ERRM command. It yields the error message associated with the value returned by ERRN.

4) The DOERR command. This command empowers a program to raise a runtime error. If no error trap is set, the DOERR command halts the program. Two types of arguments may be supplied to the DOERR, namely:

- A user-defined error message string. When the DOERR detects a string it automatically relates it with the #70000h error code.

- A binary integer or a real number that corresponds to an HP48SX runtime code number. A subsequent call to ERRN and ERRM return the argument of DOERR and the accompanying error message, respectively.

RPL provides the IFERR-THEN-END and the IFERR-THEN-ELSE-END structures to trap errors. They are discussed in the next sections.

The IFERR-THEN-END Structure

This structure is the error-handling version of the IF-THEN-END structure. The general syntax is shown below:

```
IFERR
      trap-clause
THEN
      error clause
END
```

The specialty of the IFERR-THEN-END structure is that if an error occurs while executing the trap-clause, the program jumps to the THEN clause. For the HP41C programmer, the HP48SX error-handling is superior to using the error-ignore flag 25. If you program in QuickBASIC you will find that the HP48SX offers a more structured approach to error-handling, compared to BASIC's ON ERROR GOTO statements.

The following simple example illustrates how the IFERR-THEN-END structure works. The program object implements a version of the factorial function that uses error trapping to handle negative numbers. The trap-clause contains the ! command. If the function is supplied a negative number, an error condition arises. Consequently, the program resumes in the THEN clause. In this case a 0 is pushed in the stack --- such an invalid factorial value signals that the argument was invalid to begin with. The program is listed below:

```
; Listing 8.1.  Factorial function that traps negative arguments using error
; trapping.
«
  IFERR !
  THEN
       0
  END
»
```

The above example is a typical solution for mathematical functions that may experience computational errors. These functions contain the calculating commands in the trap-clause. The error clause contains commands pushing a special number in the stack. The number should be either one that the function does not generate or a number of extreme magnitude.

For the sake of comparison, I present the defensive programming version below:

```
; Listing 8.2.  Factorial function that traps negative arguments using defensive
; programming.
«
  DUP
  0
  IF >=
  THEN
        !
  ELSE
        0
  END
»
```

Notice that the defensive programming version uses an IF-THEN-ELSE-END structure to handle negative numbers.

The error-trapping version is an example of handling arguments that raise a runtime error and halt the program in the absence of error-handling. It is interesting to point out that since the HP48SX also supports complex math, some of the math function errors that occur in HP41C and BASIC programs are not on the HP48SX. For example, the negative values supplied to the square root and logarithm functions generate complex results on the HP48SX. If such results are not acceptable to your applications, then you need to raise a runtime error. The next program implements a LN function that considers negative arguments as erroneous input. Since these arguments yield complex numbers, the program checks for their presence and accordingly raises a runtime error. I have chosen -1 to be the result returned by the function when a bad argument is supplied. In addition to protecting against negative arguments, the next program also guards against a genuine runtime error when the argument is 0. In this case, error trapping is triggered as soon as the LN command is executed. The program execution is then diverted to the THEN error-clause. The program is shown below:

```
; Listing 8.3.  Natural log function that guards against non-positive arguments.
«
  IFERR
    LN                                  ; take the LN of the number in level 1
    DUP                                 ; duplicate the result
    TYPE 1                              ; obtain the type code and compare with 1 (the type
                                        ; of complex numbers)
    IF ==
    THEN                                ; result is complex due to a non-positive argument
        # 12Fh DOERR                    ; raise "Non-Real Result" error
    END
  THEN                                  ; error clause
    DROP                                ; drop result from the stack
    -1                                  ; push -1
  END
»
```

The IFERR-THEN-ELSE-END Structure

This structure is the error-handling version of the IF-THEN-ELSE-END structure. The general syntax is shown below:

IFERR
    trap-clause
THEN
    error clause
ELSE
    normal-clause
END

The above error-handling extends the IFERR-THEN_END structure by adding an ELSE clause. The commands in the latter clause are executed if no error occurs in the trap-clause. The next program demonstrates the IFERR-THEN-ELSE-END structure. It is a version of the quadratic solver that traps division by zero error (that arise when the coefficient of the X square is zero). The program first resets the error code by using the ERR0 command. The trap-clause in the IFERR structure contains the algebraic expressions that are vulnerable to a division-by-zero error. The error-clause drops the intermediate results, pushes an error message in the stack, and beeps. The latter two steps are optional, especially if the program is called as a subroutine. In this case, the caller must compare ERRN with 0 to detect any error. The normal-clause tags the results when the calculations proceed without any error. The program is listed below:

```
; Listing 8.4.  Quadratic solver QS with a math error handling.
«
  ERR0                                  ; reset error code
  0 0
  → A B C D TWOA                        ; assign data to local variables
    «
    '√(B*B-4*A*C)' EVAL                 ; calculate the square root of the determinant
    'D' STO                             ; store result in the local variable D
    '2*A' EVAL                          ; calculate 2A
    'TWOA' STO                          ; store 2A in the local variable TWOA
    IFERR                               ; start trap-clause
        '(-B+D)/TWOA' EVAL              ; calculate the first root.  If A=0 the divide-by-zero
                                        ; error occurs here
        '(-B-D)/TWOA' EVAL              ; calculate the other root.
    THEN                                ; error clause
        DROP2                           ; drop the two intermediate results
        "ERROR: A = 0"                  ; optional error message to the user
        1000 .2 BEEP                    ; optional beep
    ELSE                                ; normal-clause
        'Root1' →TAG                    ; tag the valid results
        SWAP
        'Root2' →TAG
    END
    »
»
```

Error-Proof Input

I stated earlier that computational errors and input error are among the general classes of errors. The examples in this chapter have focused so far on computational error. In this section I will discuss input error, especially when using the INPUT command. One of the most common errors is the lack (total or partial) of actual input from the user. The scheme for error-free input is to repeat the input command(s) until the sought input is obtained. The general syntax for this scheme is shown below:

```
DO
      IFERR
            input command
            data storage commands
            1
      THEN
            drop any partial input from the stack
            0
      END
UNTIL
END
```

The nested commands are enclosed in a DO-UNTIL conditional loop. The loop's UNTIL clause performs no explicit test. Instead, it relies on 0 or 1 values passed by the clauses of an IFERR structure. The trap-clause of the IFERR structure contains the commands for INPUT and input verification (this can be simply the commands that store the input in variables). The last command in the trap-clause pushes 1 in the stack for the UNTIL clause to examine. The error-clause drops any data from a partial input and pushes 0 for the UNTIL clause.

The above scheme is put to work in the next program. The program prompts you to enter the coefficients of a quadratic equation. A single INPUT command is used to prompt you for input. The DO-UNTIL loop iterates until you enter a value for each coefficient. Once the input is complete the program calls the above version of the quadratic solver. The program is listed below:

```
; Listing 8.5. Program that prompts for the coefficients of a quadratic equation.
«
  0 0 0                                  ; push dummy data to reserve space for local variables
  → A B C                                ; assign data to local variable
    «
    DO                                   ; start DO-UNTIL loop
       IFERR                             ; start trap-clause
            "" {":A:▪:B:▪:C:" 3 0}       ; prompt for coefficients
            INPUT OBJ→
            'C' STO                      ; store input in local variables
            'B' STO                      ; insufficient data error can occur in the
            'A' STO                      ; STO commands
            1                            ; made it ok!  Push 1 in the stack
```

```
        THEN
            DROP                        ; drop partial input
            0                           ; push 0 in the stack
        END
    UNTIL                               ; re-iterate if number in level 1 is 0, exit if 1
    END
    A B C QS                            ; call subroutine QS
    »
»
```

The last two programs enjoy error-handling features for both the input and calculations.

9

Special and Non-Numerical Arrays

The HP48SX offers advanced numeric array and matrix manipulation. This includes popular matrix and matrix-vector operations, such as matrix math, matrix inversion, and solving systems of equations. This prepares the HP48SX for scientific, engineering, statistical, and numerical analysis calculations. By contrast, non-numerical arrays, such as arrays of strings, are not explicitly implemented. However, the designers of the machine have provided the basics of creating such arrays. In particular, lists (as implemented in the HP48SX) are able to act as pseudo-arrays. Thus, lists can mimic arrays of strings, compound information, numerical arrays, numerical matrices, and other lists.

This chapter looks at a few special list-based arrays:

1) Arrays of strings.
2) Arrays of compound data.
3) Jagged numerical matrices.
4) Hash tables.

The second and last structures provide you with a simple but practical databases. They can be used to store phone numbers, addresses, and other useful data.

Arrays of Strings

The HP48SX array objects are limited to storing real and complex data. Strings are excluded. Arrays of strings can be implemented as lists of strings, because you can access a list member by index (a typical feature of an array). The basic functionality of string arrays includes storing and recalling a string from the list. Secondary functionality includes sorting the string array, and searching for a specific string. The search can be either linear (for unsorted arrays) or binary (a faster way used with sorted arrays). The general form of the list-based array is:

{ "*string_1*" "*string_2*" ... "*string_n*" }

Storing Strings

Storing a string in the list-based array (from now on I will simply call it *array*) involves placing the string at the specific index. If the index is within the array's current size, the operation is straightforward. By contrast, when the index is beyond the array size the array must be first expanded to the index value. This is accomplished by adding new members to the list. Since the list implements an array of strings, null strings are suitable as fillers. What if the supplied index is less than one (the first array element)? There are two choices: trigger an error condition or substitute the bad index with a good one. The first alternative raises an error condition by

either setting a special flag or pushing an error-flagging data object into the stack. The second alternative replaces the value of the bad index with a valid one, say 1. I will implement both versions for storing strings in an array. The first version is shown below:

```
; Listing 9.1.  Program STOR to store a string in an array of strings.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing strings
;           Index
;           String to be stored in array
;
; OUTPUT:
;           Flag 64 is clear => index value > 0
;           Flag 64 is set   => index value < 1
;
«
0 → SARR INDEX S ARRSIZE                ; assign parameters to local variables
                                        ; string array to variable SARR
                                        ; index to variable INDEX
                                        ; stored string to variable S
                                        ; 0 (dummy value) to variable ARRSIZE
      «
      IF 'INDEX < 1'                    ; is the index valid?
      THEN                              ; yes, proceed
            SARR SIZE                   ; obtain the array size and store it in variable ARRSIZE
            'ARRSIZE' STO
            IF 'INDEX > ARRSIZE'        ; is the index greater than the array size?
            THEN                        ; yes.  Increase the array size by adding null strings
              'INDEX-ARRSIZE' EVAL      ; evaluate the difference in size
              1 SWAP                    ; begin a START-NEXT loop to concatenate null strings
              START
                  "" +
              NEXT
            END
            INDEX S PUT                 ; insert string at the INDEX element
            64 CF                       ; clear error flag
      ELSE
            64 SF                       ; set error flag
      END
      »
»
```

The second version of the string storing program is presented next:

```
; Listing 9.2.  Program STOR to store a string in an array of strings.
;
; Version 2
; Date 12/12/90
;
; INPUT:
;           List containing strings
;           Index
;           String to be stored in array
;
; OUTPUT:
;
«
0 → SARR INDEX S ARRSIZE                ; assign parameters to local variables
                                        ; string array to variable SARR
                                        ; index to variable INDEX
                                        ; stored string to variable S
                                        ; 0 (dummy value) to variable ARRSIZE
      «
      IF 'INDEX < 1'                    ; is the index valid?
      THEN                              ; yes, adjust index value
            1 'INDEX' STO               ; assign 1 to variable INDEX
      END
      SARR SIZE                         ; obtain the array size and store it in variable ARRSIZE
      'ARRSIZE' STO
      IF 'INDEX > ARRSIZE'              ; is the index greater than the array size?
      THEN                              ; yes.  Increase the array size by adding null strings
            'INDEX-ARRSIZE' EVAL        ; evaluate the difference in size
            1 SWAP                      ; begin a START-NEXT loop to concatenate null strings
            START
                  "" +
            NEXT
      END
      INDEX S PUT                       ; insert string at the INDEX element
      »
»
```

The above version assumes that client programs check the index value before calling STOR. Examples of using the second version of STOR are shown below. The first example illustrates a typical use of STOR, where the storage index is within the array size:

INPUT STACK	OUTPUT STACK
3: { "DO" "FOR" "NEXT" }	
2: 2	
1: "END"	1: { "DO" "END" "NEXT" }

The second example shows the resulting array when the storage index is greater than the array size. In this case, the resulting array contains two new null strings:

INPUT STACK	OUTPUT STACK
3: { "DO" "FOR" "NEXT" }	
2: 6	
1: "END"	1: { "DO" "FOR" "NEXT" "" "" "END" }

Recalling Strings

Recalling a string from an array is a simpler operation. You simply check whether the index of the sought element is valid (that is, its value is between 1 and the array size). If the index value is valid, retrieve the sought element using a GET statement. If the index is out of range, return a null string. The program is shown below:

```
; Listing 9.3.  Program RCAL to recall a string from an array of strings.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing strings
;           Index
;
; OUTPUT:
;           string
;
«
SWAP DUP SIZE                          ; obtain the array size
→ INDEX SARR ARRSIZE                   ; assign parameters to local variables
                                       ; index to variable INDEX
                                       ; string array to variable SARR
                                       ; array size to variable ARRSIZE
      «
      IF                               ; is the index valid?
         'INDEX > 0 AND INDEX ≤ ARRSIZE'
      THEN                             ; yes, adjust index value
           SARR INDEX GET              ; obtain string from the INDEX'th element
      ELSE
           ""                          ; return a null string
      END
      »
»
```

Examples of using RCAL are shown below. The first example shows a typical use of the RCAL program. Here the access index is within the array size:

INPUT STACK	OUTPUT STACK
3: { "DO" "FOR" "NEXT" }	
1: 2	1: "FOR"

The second example shows the null string obtained by using an index that exceeds the array size. The same result is obtained if the index value is less than 1:

INPUT STACK	OUTPUT STACK
2: { "DO" "FOR" "NEXT" }	
1: 4	1: ""

Sorting Strings

Sorting an array of strings causes the elements to be rearranged in either ascending or descending order. The ascending order is most commonly used. There are various sorting methods to choose from, each with its own efficiency. Most of the times, I use the Shell-Metzner sort method because it offers good speed and does not require additional stack or memory space. The QuickSort method is among the fastest, but requires stack space because it uses recursion (that is, its routines call themselves).

The Shell-Metzner method compares and swaps array elements. This is done in cycles. Each cycle compares members that are N elements apart. At first, N is half the array size. During each other cycle the value of N is cut in half, until N becomes 1. Within each cycle, multiple passes are performed to compare the array members and ensure that they are in order. The last cycle compares immediate neighboring elements. When it is finished, the array is in order.

The following program sorts the elements of a string array in an ascending order and places the result back on the stack:

```
; Listing 9.4.  Program SORT to sort the elements of a string array.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing strings
;
; OUTPUT:
;           Sorted list of strings
;
«
DUP SIZE                                ; obtain the array size
DUP                                     ; duplicate size
```

```
0
RCLF                                    ; recall flags

→ SARR ARRSIZE OFS I FLAGS              ; assign parameters to local variables
                                        ; string array to variable SARR
                                        ; array size to variable ARRSIZE
                                        ; special sort offset to variable OFS
                                        ; 0 (dummy) to variable I
                                        ; flags to variable FLAGS
      «
      WHILE 'OFS>1'                     ; main sort loop
      REPEAT
            OFS 2 / 'OFS' STO           ; halve OFS
            1 SF                        ; set in-order flag
            DO
                1 ARRSIZE OFS -         ; set the limits of a FOR-NEXT loop
                FOR J
                    J OFS +             ; calculate I
                    'I' STO
                    SARR J GET          ; obtain array element J
                    SARR I GET          ; obtain array element I
                    DUP2                ; duplicate elements
                    IF >                ; is element J > element I ?
                    THEN                ; swap them
                        SARR SWAP J     ; swap element with list in stack
                        SWAP PUT
                        SWAP I
                        SWAP PUT
                        'SARR' STO      ; update variable SARR
                        1 CF            ; clear in-order flag
                    ELSE
                        DROP2           ; drop the copies of elements I and J
                    END
                NEXT
            UNTIL 1 FS?                 ; repeat until in-order flag is true (set)
      END
      FLAGS STOF                        ; restore flags to original status
      SARR                              ; push string array in the stack
      »
»
```

The program uses flag 1 to monitor the order of the array. The original status of the flags is stored in variable FLAGS. When the program ends, the original flag states are restored. Since flag 1 conveys no information relevant to client programs. An example of using the SORT program is shown below:

INPUT STACK	OUTPUT STACK
1: { "NEXT" "FOR" "DO" }	1: { "DO" "FOR" "NEXT" }

Searching for Strings

Searching through an unordered array requires a linear method, usually searching the array from the front to the end. The search stops when either a match is located or the end of the array is reached without finding a match. A successful search returns the index of the matching element; otherwise, a zero is returned. This type of linear search assumes that strings in the

array are unique, since no attempt is made to locate subsequent occurrences of the same string. Thus, searching for the string "Stewart" in the list { "Nelson" "Stewart" "Cox" "Stewart" } always returns 2, the location of the first occurrence. The linear search program is shown below:

```
; Listing 9.5.  Program LSRCH to search for a matching string in the array.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing strings
;           Search string
;
; OUTPUT:
;           index to the matching string,
;     OR    0 when no matching is found
;
«
SWAP DUP SIZE 1
RCLF                                    ; recall flags

→ S SARR ARRSIZE I FLAGS                ; assign parameters to local variables
                                        ; search string to variable S
                                        ; string array to variable SARR
                                        ; array size to variable ARRSIZE
                                        ; 0 (dummy) to variable I
                                        ; flags to variable FLAGS
      «
      1 SF                              ; set the not-found flag
      WHILE 'I ≤ ARRSIZE' 1 FS? AND     ; start the WHILE loop and test for:
                                        ;   1) if index I is less than or equal to the array size
                                        ; AND
                                        ;   2) if the not-found flag is still set
      REPEAT
          SARR I GET                    ; obtain the I'th array element
          S                             ; push the search string in the stack
          IF ≠                          ; does the I'th array element differ from the search string?
          THEN                          ; Yes.  Increment index I
            1 'I' STO+
          ELSE                          ; No. Clear the not-found flag
              1 CF
          END
      END
      IF 1 FC?                          ; is the not-found flag clear
      THEN                              ; Yes.  Return index for matching element
          I
      ELSE                              ; No. Return 0 for a no-match search
          0
      END
      FLAGS STOF                        ; restore original flags
      »
»
```

Examples of using LSRCH are shown below. Consider locating the string "DO" in the list { "IF" "FOR" "DO" }, as shown below:

INPUT STACK	OUTPUT STACK
2: { "IF" "FOR" "DO" }	
1: "DO"	1: 3

A request to find the string "Do" returns 0 (since the string "Do" is not in the list), as shown below:

INPUT STACK	OUTPUT STACK
2: { "IF" "FOR" "DO" }	
1: "Do"	1: 0

The binary search method is applied to ordered arrays. The method is much faster than the linear scheme and uses the same approach of the root-solving bisection method in numerical analysis. The binary search method starts by examining the median element of the array. This enables the scheme to skip half of the array and examine the relevant half. Successive searches continue to reduce the size of the search sub-array by examining the median of the current sub-array. The search stops until either a match is found or the size of the sub-array just examined is 1.

Program object BSRCH performs a binary search on the ordered string array.

```
; Listing 9.6. Program BSRCH to binary search for a matching string in the array.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing strings
;           Search string
;
; OUTPUT:
;           index to the matching string,
;     OR    0 when no matching is found
;
«
SWAP DUP SIZE 1 0 ""
→ S SARR HIGH LOW MEDIAN ELEM        ; assign parameters to local variables
                                     ; search string to variable S
                                     ; string array to variable SARR
                                     ; array size to variable HI
                                     ; 1 to variable LOW
                                     ; 0 (dummy) to variable MEDIAN
                                     ; null string (dummy) to variable ELEM
      «
      DO
            LOW HIGH + 2 / IP        ; calculate the median value
```

```
            DUP 'MEDIAN' STO          ; store in variable MEDIAN
            SARR SWAP GET             ; obtain the median element
            'ELEM' STO                ; store in variable ELEM
            IF 'S < ELEM'             ; compare search and median strings
            THEN
                  M 1 -               ; select lower sub-array
                  'HIGH' STO
            ELSE
                  M 1 +               ; select upper sub-array
                  'LOW' STO
            END
      UNTIL
            'S==ELEM OR LOW> HIGH'    ; test for end of search
      END
      IF 'S==ELEM'                    ; found a match?
      THEN                            ; Yes. Return the median
            MEDIAN
      ELSE                            ; No. Return zero
            0
      END
      »
»
```

The BSRCH program uses flag 1 to indicate whether or not a match is found. The original status of the flag is stored in variable FLAGS and then restored near the end of the program.

Examples of using BSRCH are shown next. Consider the array:

{ "Bear" "Bird" "Dog" "Lion" "Tiger" "Zebra" }

Searching for "Bird", "Lion", "Cat" returns 2, 4, and 0 respectively. The BSRCH and LSRCH programs return the same results when dealing with sorted arrays. The difference is that BSRCH is faster than LSRCH.

Compound Arrays

The list-based array of strings is made of a un-nested list containing strings. Such a list is useful when each string member represents a complete piece of information. What about the applications where multiple data fields are needed? The simple list-based arrays can be expanded to include *compound* data. A compound array is a nested list with the following general format:

{ { key_1 data_12 data_13 ... data_1n} ... { key_n data_n2 ... data_nm } }

A compound element is a list of data where the first list member is the key-information field. The key field must be the same for all compound elements in the array. You can use either a real number, a binary integer, or a string as the key field. Other data in compound elements can vary in type and number. If you have been exposed to structured languages you may see a similarity between compound elements and records or structures. Unlike such records and structures, there is no formal definition of the compound data type. Your applications can create them at will. With such a privilege comes the responsibility of using consistent keys, or

risk a variety of runtime errors!

The compound array is an expanded version of the simple string array presented earlier. Added complexity comes from the fact that we are dealing with nested lists. This means that operations store and recall must deal with extra steps of obtaining data from nested lists and putting them back.

Compound arrays enable you to maintain a flexible little database on the HP48SX. For example you can store mailing data or phone numbers. The last name can be used as the key field. So the next time you want to call your favorite HP dealer (to order some more accessories for the HP48SX), locate his/her phone number or address by supplying the last name!

Storing Compound Elements

Storing a compound element in a compound array is very similar to the simpler string array version.

The version of the string storing program is presented below:

```
; Listing 9.7.  Program STOR to store a compound element in a compound array.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           Nested list representing the compound array
;           Index
;           compound data list to be stored in array
;
; OUTPUT:
;
«
0 → CARR INDEX S ARRSIZE                ; assign parameters to local variables
                                        ; compound array to variable CARR
                                        ; index to variable INDEX
                                        ; stored compound element to variable S
                                        ; 0 (dummy value) to variable ARRSIZE
      «
      IF 'INDEX < 1'                    ; is the index valid?
      THEN                              ; yes, adjust index value
            1 'INDEX' STO               ; assign 1 to variable INDEX
      END
      CARR SIZE                         ; obtain the array size and store it in variable ARRSIZE
      'ARRSIZE' STO
      IF 'INDEX > ARRSIZE'              ; is the index greater than the array size?
      THEN                              ; yes.  Increase the array size by adding null strings
            'INDEX-ARRSIZE' EVAL        ; evaluate the difference in size
            1 SWAP                      ; begin a START-NEXT loop to concatenate null strings
            START
                  "" +
            NEXT
      END
      INDEX S PUT                       ; insert compound element at the INDEX element
      »
```

»

The above version assumes that client programs check the index value before calling STOR. Examples of using STOR are shown below. The first example illustrates a typical use, where the storage index is within the array size:

INPUT STACK	OUTPUT STACK
3: { { "FOR" 1 } { "DO" 10 } }	
2: 2	
1: { "IF" 5 }	1: { { "FOR" 1 } { "IF" 5 } }

The second example shows the resulting array when the storage index is greater than the array size. In this case, the resulting array contains two new null strings:

INPUT STACK	OUTPUT STACK
3: { { "DO" 3 } }	
2: 4	
1: { "END" 9 }	1: { { "DO" 3 } "" "" { "END" 9 } }

Recalling Compound elements

Recalling a compound element is very similar to the string array version. An out-of-range index results in a null string being pushed to the stack. Checking for null strings after invoking RCAL enables your program to detect bad-index calls. The program is shown below:

```
; Listing 9.8.  Program RCAL to recall a compound element from an array of
; compound data.
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing compound elements
;           Index
;
; OUTPUT:
;           compound element
;
«
SWAP DUP SIZE                           ; obtain the array size
→ INDEX CARR ARRSIZE                    ; assign parameters to local variables
                                        ; index to variable INDEX
                                        ; compound array to variable CARR
                                        ; array size to variable ARRSIZE
      «
      IF                                ; is the index valid?
          'INDEX > 0 AND INDEX ≤ ARRSIZE'
      THEN                              ; yes, adjust index value
          CARR INDEX GET                ; obtain compound element from the INDEX'th element
      ELSE
          ""                            ; return a null string
```

```
      END
      »
»
```

Examples of using RCAL are shown below. The first example shows a typical use of the RCAL program, here, the access index is within the array size:

INPUT STACK	OUTPUT STACK
2: { { "DO" 3 } { "FOR" 4 } }	
1: 2	1: { "FOR" 4 }

The second example shows the string obtained by using an index that exceeds the array size. The same result is obtained if the index value is less than 1:

INPUT STACK	OUTPUT STACK
2: { { "DO" 3.14 } }	
1: 4	1: ""

Sorting Compound elements

Sorting an array of compound elements is similar to sorting a string array. An added complication comes from the fact that a two-step extraction must be carried out to reach the key fields: the first obtains the compound element from the compound array, and the second retrieves the first list member of the compound element. In addition, the compound elements must be stored in separate variables for the sake of (potential) swapping.

The following program sorts the elements of a compound array in an ascending order and places the result back on the stack:

```
; Listing 9.9.  Program SORT to sort the elements of a compound array.
;
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing compound elements
;
; OUTPUT:
;           Sorted list of compound elements
;
«
DUP SIZE                                ; obtain the array size
DUP                                     ; duplicate size
0
RCLF                                    ; recall flags
{ } DUP
→ ARRSIZE OFS I FLAGS EI EJ             ; assign parameters to local variables
                                        ; array size to variable ARRSIZE
                                        ; special sort offset to variable OFS
                                        ; 0 (dummy) to variable I
```

```
                                        ; flags to variable FLAGS
                                        ; empty list (dummy) to variable EI
                                        ; empty list (dummy) to variable EJ
    «
    WHILE 'OFS>1'                       ; main sort loop
    REPEAT
          OFS 2 / 'OFS' STO             ; halve OFS
          1 SF                          ; set in-order flag
          DO
              1 ARRSIZE OFS -           ; set the limits of a FOR-NEXT loop
              FOR J
                J OFS +                 ; calculate I
                'I' STO
                DUP DUP
                J GET                   ; obtain array element J
                'EJ' STO                ; store it in variable EJ
                I GET                   ; obtain array element I
                'EI' STO                ; store it in variable EI
                EJ 1 GET                ; obtain the key field of element J
                EI 1 GET                ; obtain the key field of element I
                IF >                    ; is element J > element I ?
                THEN                    ; swap them
                      J EI PUT          ; swap element with list in stack
                      I EJ PUT
                      1 CF              ; clear in-order flag
                END
              NEXT
          UNTIL 1 FS?                   ; repeat until in-order flag is true (set)
    END
    FLAGS STOF                          ; restore flags to original status
    »
»
```

While the above implementation of SORT resembles that of the string array, you might have noticed that I am not using a local variable to store the compound array. Instead, the implementation leaves the array in the stack.

An example of using the SORT program is shown below:

INPUT STACK	**OUTPUT STACK**
1: { { "NEXT" 3 } { "FOR" 9 } { "DO" 4 } }	1: { { "DO" 4 } { "FOR" 9 } { "NEXT" 3 } }

Searching for Compound elements

I will present two programs that implement linear and binary searching on compound arrays. The implementations are similar to those of string arrays.

```
; Listing 9.10.  Program LSRCH to search for a matching compound element in the
; array.
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing compound elements
;           Search element
;
; OUTPUT:
;           index to the matching compound element,
;     OR    0 when no matching is found
;
«
SWAP DUP SIZE 1
RCLF                                    ; recall flags

→ S CARR ARRSIZE I FLAGS                ; assign parameters to local variables
                                        ; search key to variable S
                                        ; compound array to variable CARR
                                        ; array size to variable ARRSIZE
                                        ; 0 (dummy) to variable I
                                        ; flags to variable FLAGS
      «
      1 SF                              ; set the not-found flag
      WHILE 'I ≤ ARRSIZE' 1 FS? AND     ; start the WHILE loop and test for:
                                        ;   1) if index I is less than or equal to the array size
                                        ; AND
                                        ;   2) if the not-found flag is still set
      REPEAT
          CARR I GET                    ; obtain the I'th array element
          1 GET                         ; extract the key field
          S                             ; push the search key in the stack
          IF ≠                          ; does the key field of the I'th array element differ from
                                        ; the search key?
          THEN                          ; Yes.  Increment index I
            1 'I' STO+
          ELSE                          ; No. Clear the not-found flag
              1 CF
          END
      END
      IF 1 FC?                          ; is the not-found flag clear
      THEN                              ; Yes.  Return index for matching element
          I
      ELSE                              ; No. Return 0 for a no-match search
          0
      END
      FLAGS STOF                        ; restore original flags
      »
»
```

Examples of using LSRCH are shown below. Consider locating the compound element "DO" in the list { { "IF" 4 } { "FOR" 5 } { "DO" 6 } }, as shown below:

INPUT STACK	OUTPUT STACK
2: { { "IF" 4 } { "FOR" 5 } { "DO" 6 } }	
1: "DO"	1: 3

A request to find the compound element "Do" returns 0 (since the compound element "Do" is not in the list), as shown below:

INPUT STACK	OUTPUT STACK
2: { { "IF" 4 } { "FOR" 5 } { "DO" 6 } }	
1: "Do"	1: 0

Program object BSRCH performs a binary search on the ordered compound array.

```
; Listing 9.11.  Program BSRCH to binary search for a matching compound element
; in the array.
; Version 1
; Date 12/12/90
;
; INPUT:
;           List containing compound elements
;           Search key
;
; OUTPUT:
;           index to the matching compound element,
;     OR    0 when no matching is found
;
«
SWAP DUP SIZE 1 0 ""
→ S CARR HIGH LOW MEDIAN ELEM        ; assign parameters to local variables
                                     ; search key to variable S
                                     ; compound array to variable CARR
                                     ; array size to variable HI
                                     ; 1 to variable LOW
                                     ; 0 (dummy) to variable MEDIAN
                                     ; null string (dummy) to variable ELEM
      «
      DO
            LOW HIGH + 2 / IP        ; calculate the median value
            DUP 'MEDIAN' STO         ; store in variable MEDIAN
            CARR SWAP GET            ; obtain the median element
            1 GET                    ; get the key field of the median element
            'ELEM' STO               ; store in variable ELEM
            IF 'S < ELEM'            ; compare search key and key of median compound element
            THEN
                  M 1 -              ; select lower sub-array
                  'HIGH' STO
            ELSE
```

```
                M 1 +                   ; select upper sub-array
                'LOW' STO
        END
    UNTIL
        'S==ELEM OR LOW>HIGH'           ; test for end of search
    END
    IF 'S==ELEM'                        ; found a match?
    THEN                                ; Yes. Return the median
        MEDIAN
    ELSE                                ; No. Return zero
        0
    END
    »
»
```

Compound arrays are very powerful data structures. Their usefulness is indeed limited by our imagination!

Hash Tables

The superiority of binary search over linear search stems from the fact that it takes advantage of the array's sorted elements. Computer scientists also discovered that, surprisingly, fast searches can also be attained with arrays whose elements are in perfect disorder. How is that possible, you might ask? The basic scheme, called *hashing*, converts the key portion of the inserted data (usually a string) into a unique index. The inserted data is then stored at that index, without examining other array elements. Thus, an array that uses hashing to store and retrieve data is called a *hash table*. The unique index is calculated using a *hashing function* that produces a random-number index. Ideally, the hashing function should not produce the same index for two different data. This deviation, which causes data *collision*, occurs for two reasons:

1) The randomness of the result given by the hashing function.
2) The ratio between the number of inserted data and hash table size. The more crowded the hash table is, the more likely is collision.

There are various methods to handle colliding data. The best method is called *chaining* and can be easily implemented on the HP48SX. It works by creating a list of the colliding data at the offending index. The ideal hash table that stores strings can be implemented as a simple list. The ideal hash table that stores compound data is implemented as a nested list:

{ { key13 ... } { key99 ... } { key55 ... } ... { key67 ... } }

To implement chaining, another level of nested lists must be added. The general form of the hash table that chains colliding data is shown below:

{ { { key13 ... } { key45 ... } } { { key50 ... } { key67 ... } } ... }

The compound data with keys key13 and key45 are hashed into the same index, 1. Similarly, the data with keys key50 and key67 are hashed into index 2. Using lists that chain colliding

data enables a variable number of colliding data to be stored.

While hash tables succeed in providing fast data insertion and search, they are unable to readily provide sorted data. You can still obtain sorted data from a hash table but there is a cost! You need to search for all possible entries --- this works if the key data are limited and finite.

I present the implementation of a hash table system that creates hash tables, inserts data, deletes data, and retrieves data. The hash table has the following features:

1) It handles compound data. This enables you to stores and quickly retrieve more meaningful data, such as address and phone numbers.
2) The key field must be a string.

Creating a Hash Table

The first step in the life of a hash table is its creation. The following program creates a two-level nested list with the number of inner lists equal to the specified hash table size. This enables the list-based hash table to obtain its size by using the SIZE command.

```
; Listing 9.12.  Program CRTHT to create an empty hash table.
;
; Version 1
; Date 12/13/90
;
; INPUT:
;           Hash table size
;
; OUTPUT:
;           List-based empty hash table
;
;
«
{ }                                     ; insert the main list in the stack
SWAP 1 SWAP                             ; define the limits of a FOR-NEXT loop
FOR I                                   ; start FOR-NEXT loop
      "" +                              ; concatenate a null string to the main list
      I { } PUT                         ; overwrite the null string with an empty nested list
NEXT
»
```

The CRTHT program is straightforward. It appends nested null lists by first appending null strings and then overwriting them with empty nested lists. Attempting to directly append empty lists yields a single-level empty main list! An example of using program CRTHT to create a 3-entry table is shown below:

INPUT STACK	OUTPUT STACK
1: 3	1: { { } { } { } }

Hashing Function

The hashing function HASHF calculates the hash index for a given string and hash table size. The method used is shown in pseudo-code below:

1) Let SUM = 0,
2) Let L = length of input string
3) Let TS = input hash table size
4) For I = 1 to L Do
    4.1) Let C = I'th string character
    4.2) Let ASC = the ASCII code of character C
    4.3) Let SUM = SUM + (19 * ASC + 13) MOD TS
5) Let SUM = SUM MOD TS + 1
6) Return SUM as the hash index

```
; Listing 9.13.  Function HASHF that returns the hash index.
;
; Version 1
; Date 12/13/90
;
; INPUT:
;           String
;           Hash table size
;
; OUTPUT:
;           Hash index
;
«
SWAP DUP SIZE → TSIZE S LEN          ; assign parameters to local variables
                                     ; hash table size to variable TSIZE
                                     ; string to variable S
                                     ; the length of string S to variable LEN
      «
      0                              ; insert the initial value of the hash index
      1 LEN                          ; set the limits of the FOR-NEXT loop
      FOR I
      S I I SUB                      ; obtain the I'th character of string S
      NUM                            ; obtain its ASCII code
      19 * 13 + TSIZE MOD +          ; update the hash index
      NEXT
      TSIZE MOD 1 +                  ; calculate the final value for the hash index
      »
»
```

Inserting Data

Inserting compound data in a hash table involves the following steps:

1) Calculating the insertion index using the hashing function.
2) Making sure that the target list does not contain a compound element with the same key. If so, the insertion is halted. This scheme guards against compound data with duplicate keys.
3) Inserting the compound data in the target nested list.

The insertion program INSHT is shown below:

```
; Listing 9.14.  Program INSHT to insert compound data in the hash table.
;
; Version 1
; Date 12/13/90
;
; INPUT:
;           List-based hash table
;           Compound data to be inserted
;
; OUTPUT:
;           Update hash table
;
; CALL:
;           HASHF to obtain hash index
;           LSRCH to locate any duplicate keys in the hash table. This is the
;           same routine for compound data.
;
«
0 DUP { } ""
→ HT REC TSIZE INDEX HELM ELEM      ; assign parameters to local variables
                                    ; hash table to variable HT
                                    ; compound data to variable REC
                                    ; 0 (dummy) to hash table size variable TSIZE
                                    ; 0 (dummy) to hash index variable INDEX
                                    ; empty list (dummy) to hash entry variable HELM
                                    ; null string (dummy) to variable ELEM
      «
      HT SIZE 'TSIZE' STO           ; obtain hash table size and store it in variable TSIZE
      REC 1 GET 'ELEM' STO          ; obtain the data key and store it in variable ELEM
      ELEM TSIZE HASHF              ; calculate the hash index
      'INDEX' STO                   ; store hash index in variable INDEX
      HT INDEX GET DUP              ; obtain the INDEX'th sublist
      'HELM' STO                    ; store one copy into variable HELM
      ELEM LSRCH                    ; search for duplicates in the hash entry
      0
      IF ==                         ; any duplicate keys?
      THEN                          ; not found. Insert the data
            HT INDEX                ; push hash table and entry index into the stack
            HELM "" +               ; concatenate an empty string to the target sublist
            DUP SIZE                ; obtain the size of the sublist
            REC PUT                 ; append new compound data to the sublist
            PUT                     ; update the hash table
      ELSE                          ; handle duplicate
            HT                      ; push unchanged hash table back in the stack
      END
      »
»
```

An example of inserting information in the hash table is shown below:

INPUT STACK	OUTPUT STACK
2: { { } { } { } }	
1: { "JIM" "555-1234" }	1: { { } { } { { "JIM" "555-1234" } } }

Searching for Data

Searching for data in a hash table is usually faster than binary search in a list. The advantage of hash table searches is maximized when colliding elements are minimized. The steps involved in hash table searching are:

1) Calculate the hash index of the search key string.
2) Locate the sublist with the same index.
3) If the sublist is not empty, perform a linear search.
4) A successful linear search returns the matching compound element. Otherwise, it returns an empty list to indicate that no match was found.

The SRCHT program is shown below:

```
; Listing 9.15.  Program SRCHT to search for data in a hash table.
;
; Version 1
; Date 12/13/90
;
; INPUT:
;           List-based hash table
;           Search key string
;
; OUTPUT:
;           Compound data with the matching search key,
;     OR    an empty list if no match is found
;
; CALL:
;           HASHF to obtain hash index
;           LSRCH to locate matching data.
;
«
0  { }
→ HT S TSIZE HELM                    ; assign parameters to local variables
                                     ; hash table to variable HT
                                     ; key search string to variable S
                                     ; 0 (dummy) to hash table size variable TSIZE
                                     ; empty list (dummy) to hash entry variable HELM
      «
      HT SIZE 'TSIZE' STO            ; obtain hash table size and store it in variable TSIZE
      S TSIZE HASHF                  ; calculate the hash index
      HT SWAP GET DUP                ; obtain the target sublist
      'HELM' STO                     ; store one copy into variable HELM
      ELEM LSRCH                     ; search for duplicates in the hash entry
      DUP 0
      IF >                           ; found a match?
      THEN                           ; yes! Extract the data
            HELM SWAP GET            ; get matching compound element
      ELSE                           ; no!
            DROP                     ; drop the 0 search result
            { }                      ; push an empty list in the stack
      END
      »
»
```

An example of searching for data in the hash table is shown below. The request is made to find the record with the key field "JIM":

INPUT STACK	OUTPUT STACK
2: { { } { } { { "JIM" "555-1234" } } }	
1: "JIM"	1: { "JIM" "555-1234" }

Deleting Data

Updating data with string arrays and compound arrays is simple --- you store the new information at a target index. By contrast, my implementation of hash tables prevent you from directly updating data, to guard against duplicate keys. Moreover, since hash tables create the index for storing data, you haven't the vaguest idea where the sought data is. The answer to these problems is the deletion of the older version of the data, before the new one is inserted.

The steps involved in deleting an existing compound element are:

1) Calculate the hash index given the key string of the sought element.
2) If hash index points to an occupied sublist proceed with a linear search of that sublist.
3) If the linear search finds a matching element remove that element. This is done by (a) overwriting the deleted element with last element, and then (b) extracting all of the list elements except the last one (which gives the net effect of deleting the sought element).

The DELHT program is shown below:

```
; Listing 9.16.  Program DELHT to delete a compound element from the hash table.
;
; Version 1
; Date 12/13/90
;
; INPUT:
;           List-based hash table
;           Search key string
;
; OUTPUT:
;           Update hash table
;
; CALL:
;           HASHF to obtain hash index
;           LSRCH to locate matching data.
;
«
0  DUP DUP2 { }
→ HT S TSIZE INDEX I LEN HELM        ; assign parameters to local variables
                                     ; hash table to variable HT
                                     ; key search string to variable S
                                     ; 0 (dummy) to hash table size variable TSIZE
                                     ; 0 (dummy) to the hash index variable INDEX
                                     ; 0 (dummy) to search index variable I
                                     ; 0 (dummy) to the sublist length variable LEN
                                     ; empty list (dummy) to hash entry variable HELM
```

```
      «
      S
      HT SIZE DUP 'TSIZE' STO          ; obtain hash table size and store it in variable TSIZE
      HASHF                            ; calculate the hash index
      'INDEX' STO                      ; store the hash index in variable INDEX
      HT INDEX GET DUP                 ; obtain the target sublist
      'HELM' STO                       ; store one copy into variable HELM
      SIZE 'LEN' STO                   ; obtain the sublist length and store it in variable LEN
      HELM S LSRCH                     ; search a match
      'I' STO                          ; store result of search in variable I
      IF 'I > 0'                       ; found a match?
      THEN                             ; yes! Delete the data
            HT INDEX                   ; push hash table and hash index in the stack
            HELM DUP LEN GET           ; obtain the last sublist member
            I SWAP PUT                 ; put last member in element I
            1 LEN 1 - SUB              ; remove the last list element
            PUT                        ; update the hash table
      ELSE                             ; no!
            HT                         ; push the unchanged hash table back in the stack
      END
      »
»
```

An example of deleting an entry in the hash table is shown below. The request is made to remove the record with the key field "JIM":

INPUT STACK	OUTPUT STACK
2: { { } { } { { "JIM" "555-1234" } } } 1: "JIM"	1: { { } { } { } }

Converting to Compound Arrays

While hash tables search data quickly, obtaining their element in an ordered manner is very difficult and, in some cases, is impossible. An easy solution is to convert the compound-data hash table into a compound array. The latter can be sorted using the SORT program presented earlier for compound data.

The program →SL, shown below, converts a hash table into a compound array. This is accomplished by visiting every sublist of the hash table and extracting all of the compound elements. The elements of the compound array maintain the same sequence of the hash table.

```
; Listing 9.17.  Program →SL to convert a hash table into a compound array.
;
; Version 1
; Date 12/13/90
;
; INPUT:
;           List-based hash table
;
; OUTPUT:
;           Compound array
;
«
{ } "" { }
```

```
→ HT SL LST                          ; assign parameters to local variables
                                     ; hash table to variable HT
                                     ; empty list to the resulting array variable SL
                                     ; empty list (dummy) to the sublist variable LST
      «
      1 HT SIZE                      ; define the limits of the FOR-NEXT loop to examine
                                     ; all of the sublists of the hash table
      FOR I
            HT I GET DUP             ; get the I'th sublist
            'LST' STO                ; store it in variable LST
            SIZE DUP 0               ; get the sublist size
            IF >                     ; is the sublist occupied?
            THEN
                1 SWAP               ; define the limits of the FOR-NEXT loop that extracts
                                     ; each sublist member
                FOR J
                    SL "" +          ; expand the resulting array by one element
                  DUP SIZE
                  LST J GET          ; append the compound element to the resulting array
                  PUT
                  'SL' STO           ; update the resulting list-based array
                NEXT
            ELSE
                DROP                 ; drop an intermediate result
            END
      NEXT
      SL                             ; push the resulting compound array in the stack

      »
»
```

An example of converting a hash table into a compound array with program →SL is shown below:

INPUT STACK	OUTPUT STACK
1: {	1: {
{ { "JOHN" "555-5555" } }	{ "JOHN" "555-5555" }
{ { "PAUL" "555-4321" } }	{ "PAUL" "555-4321" }
{ { "JIM" "555-1234" } } }	{ "JIM" "555-1234" } }

Jagged Matrices

Matrices are typically square or rectangular in shape. This means that the number of elements in each matrix row is consistent. The HP48SX supports this type of matrix. In this section, I will present list-based jagged matrices. Such matrices are characterized by varying row sizes (or column sizes, depending on how you look at it). Lower triangular matrices (such as those used in the LU matrix-decomposition method) are a special case of jagged matrices. The typical jagged matrix has no restrictions on the individual row size (except available memory!).

The implementation of jagged matrices is carried out based on the following features:

1) The family of real arrays, representing the jagged matrix rows, are enclosed in a list.
2) The number of matrix rows can be extended at runtime.
3) The smallest row is [0] by default.
4) The size of each matrix row can be expanded by a redimensioning process. This enables the jagged matrices to grow by rows and columns at runtime.

The general form of the jagged matrix is shown below:

{ [row_1] [row_2] ... [row_n] }

The number of rows is equal to the size of the host list.

The above specification depicts an extremely flexible jagged array. The operations associated with this particular implementation of jagged matrices are:

1) Storing an element at a specified row and column. Expanding the rows and columns of the jagged matrix are carried out if need be.
2) Recalling a jagged matrix element from a specified row and column. If either or both of the access row and column are out of range, a 0 is returned.
3) Storing a new row to replace an existing one. This enables a program to perform faster data editing on a jagged matrix.
4) Recalling an existing row enables row-wise data retrieval.
5) Searching for a matrix element at a specified row and starting with a specified column index. This feature enables you to find duplicate data in the same row.

Notice that the above operations exclude an explicit one for creating a jagged matrix. This is not required since the jagged matrix is initially an empty list which dynamically grows by calling the storage operations.

Storing Jagged Matrix Elements

The basic notion of storing data in a jagged matrix is the same as in an ordinary square or rectangular matrix. You store a given number at a specified row and column of the jagged matrix. Here is where the similarity between the jagged and ordinary matrix ends. The specified row and column can lie outside the current dimension of the dynamically growing jagged matrix. Rather than flagging such values as erroneous, the jagged matrix is extended to match the specified row and column values. Naturally, excessively high value will drain the machine's memory.

In the case when the specified data insertion row is greater than the current rows of the jagged matrix new rows are added. A set of [0] arrays are inserted as filler rows when the insertion row exceeds the current number of rows by more than 1.

The value of the insertion column index is compared with the size of the insertion row (that is, the size of the array representing the specified jagged matrix row). If the column value exceeds the array size, the array is redimensioned. This is carried out using the RDM command. The progrm STOIJ is listed next:

```
; Listing 9.18.  Program STOIJ to store an element in a jagged matrix.
;
; Version 1
; Date 12/14/90
;
; INPUT:
;           Jagged matrix
;           Row
;           Column
;           Element
;
; OUTPUT:
;           Updated jagged matrix
;
«
0 0 → JM ROW COL ELM LEN ARR          ; store parameters in local variables
                                      ; jagged matrix in variable JM
                                      ; insertion row in variable ROW
                                      ; insertion column in variable COL
                                      ; inserted element in variable ELM
                                      ; 0 (dummy) in variable LEN
                                      ; 0 (dummy) in variable ARR
      «
      IF 'ROW < 0'                    ; adjust non-positive row values to 1
      THEN
            1 'ROW' STO
      END
      IF 'COL < 0'                    ; adjust non-positive column values to 1
      THEN
            1 'COL' STO
      END
      JM SIZE 'LEN' STO               ; get the number of matrix rows and store it in variable LEN
      IF 'LEN < ROW'                  ; insertion row beyond the current number of rows?
      THEN                            ; Yes.  Expand the number of jagged matrix rows
            JM                        ; push the jagged matrix in the stack
            1 ROW LEN -               ; specify the limits of the START-NEXT loop
            START
                  [ 0 ] +             ; append [ 0] arrays to the list-based jagged matrix
            NEXT
            'JM' STO                  ; update the jagged matrix
      END
      JM ROW GET DUP                  ; obtain the insertion row
      'ARR' STO                       ; store it in variable ARR
      SIZE 1 GET                      ; get the size of the insertion row
      'LEN' STO                       ; store it in variable LEN
      IF 'LEN < COL'                  ; is the insertion column beyond the row size?
      THEN                            ; Yes! Expand the row
            ARR                       ; push the array in the stack
            { COL }                   ; push a list with the new array size in the stack
            RDM                       ; redimension the array
      ELSE                            ; No! use current array
            ARR                       ; push current array in the stack
      END
      COL ELEM PUT                    ; insert new element in jagged matrix row
      JM SWAP ROW SWAP PUT            ; insert row back in jagged matrix
      »
»
```

Examples of storing data in a jagged matrix are presented next. The first example stores the number 10 in row 1 and column 1 of an empty jagged matrix:

```
INPUT STACK          OUTPUT STACK

4: { }
3: 1
2: 1
1: 10                1: { [ 10 ] }
```

The second example stores the number 55 at row 2 and column 3 of an empty jagged matrix:

```
INPUT STACK          OUTPUT STACK

4: { }
3: 2
2: 3
1: 55                1: {
                          [ 0 ]
                          [ 0 0 55 ] }
```

The above shows that the specified row/column values resulted in first row being the array [0] and the first two elements of the second row also being zero.

The third example shows the storage of the number 55 within the current boundaries of a jagged matrix:

```
INPUT STACK                OUTPUT STACK

4: { [ 1 2 ] [ 3 4 ] [ 6 7 ] }
3: 3
2: 2
1: 55                      1: {
                                [ 1 2 ]
                                [ 3 4 ]
                                [ 6 55 ] }
```

Recalling Matrix Elements

Recalling jagged matrix elements is also similar to that of an ordinary matrix — you specify the matrix and the accessed row and column. Accessing jagged matrix elements is simpler and faster that storing them since no matrix expansion is required. The values for the row and column must be within the current limits of the jagged matrix. The program RCLIJ is shown next:

```
; Listing 9.19.  Program RCLIJ to recall a jagged matrix element.
;
; Version 1
; Date 12/14/90
;
; INPUT:
;           Jagged matrix
;           Row
;           Column
;
; OUTPUT:
;
;           Jagged matrix element,
;     OR    0, for out-of-bound row/column
;
«
0 0 → JM ROW COL LEN                    ; store parameters in local variables:
                                        ; jagged matrix in variable JM
                                        ; accessed row in variable ROW
                                        ; accessed column in variable COL
                                        ; 0 (dummy) in variable LEN
      «
      JM SIZE 'LEN' STO                 ; get the number of jagged matrix rows and store then in LEN
      IF 'ROW > 0 AND ROW ≤ LEN'        ; is the value of the accessed row within the matrix limit?
      THEN                              ; Yes! Then proceed
            JM ROW GET                  ; get the accessed row
            DUP SIZE                    ; get the size of the row
            1 GET
            'LEN' STO                   ; store the array size in variable LEN
            IF                          ; is the column value within the jagged matrix row range?
                'COL > 0 AND COL ≤ LEN'
            THEN                        ; Yes! Access the sought element
                  COL GET
            ELSE                        ; No! Push 0 in the stack as the result
                  DROP
                  0
            END
      ELSE                              ; No! Push zero in the stack as the result
            0
      END
      »
»
```

Examples of recalling jagged matrix elements are presented next. The first case deals with recalling an element from row 2 and column 3 of a jagged matrix. Since the specified values of the row and column are within the jagged matrix limits, the non-zero value of 66 is returned:

INPUT STACK	OUTPUT STACK
3: { [11 22 33] [44 55 66 77] }	
2: 2	
1: 3	1: 66

The second case shows an attempt to access the matrix element at row 3 and column 3. Since

the jagged matrix has only two rows, the coordinates of the matrix element are out of bound and a zero is returned instead:

INPUT STACK	OUTPUT STACK
3: { [11 22 33] [44 55 66 77] }	
2: 3	
1: 3	1: 0

Storing Rows

A quicker way to store an entire row is accomplished by inserting a new row or overwriting an existing one. This assumes that your application can build or update a jagged matrix in chunks of rows. The speed of storing rows over storing single elements is due to two reasons:

1) A single call to the row-storage program is made. This contrast the multiple calls that must be issued to program STOIJ.
2) No row expansion occurs, since the new row is inserted "as is".

The process of expanding the jagged matrix rows must still be carried out as with the element-storage scheme. The program STOR is listed below:

```
; Listing 9.20.  Program STOR to store a jagged matrix row.
;
; Version 1
; Date 12/14/90
;
; INPUT:
;           Jagged matrix
;           Insertion row
;           Inserted array
;
; OUTPUT:
;           Updated jagged matrix
;
«
0 → JM ROW ARR LEN                    ; assign the parameters to local variables
                                      ; jagged matrix to variable JM
                                      ; insertion row to variable ROW
                                      ; inserted row to variable ARR
                                      ; 0 (dummy) to variable LEN
      «
      IF 'ROW < 1'                    ; assign one to ROW if its value is non-positive
      THEN
            1 'ROW' STO
      END
      JM SIZE 'LEN' STO               ; get the size of the jagged matrix and store it in LEN
      JM                              ; push the jagged matrix in the stack
      IF 'ROW > LEN'                  ; does the insertion row exceed the size of the matrix?
      THEN                            ; Yes! Add new rows to the jagged matrix
            1 ROW LEN -               ; calculate the number of addition rows needed and use
                                      ; that value as the upper limit of a START-NEXT loop
            START
```

```
                [ 0 ] +              ; append an empty row ([ 0 ]) to the jagged matrix
        NEXT
    END
    ROW ARR PUT                      ; insert the new row into the jagged matrix
    »
»
```

Examples of storing rows in a jagged matrix are presented next. The first example overwrite an existing row with a new one --- the second row, array [3 4], is replaced by a bigger one, array [11 22 33]:

INPUT STACK	OUTPUT STACK
3: { [1 2 3] [3 4] [5 6 7 8 9] }	
2: 2	
1: [11 22 33]	1: { [1 2 3] [11 22 33] [5 6 7 8 9] }

The second example shows how the STOR program expands a jagged matrix by adding a new row. An array is inserted as the new fifth row in a three-row jagged matrix. The result is the expansion of the jagged matrix by two rows --- row 4 is [0] and row 5 is the inserted row:

INPUT STACK	OUTPUT STACK
3: { [1 2 3] [3] [5 6] }	
2: 5	
1: [11 22]	1: { [1 2 3] [3] [5 6] [0] [11 22] }

Recalling Rows

The counterpart of storing rows in a jagged matrix is recalling them. The steps involved are very simple. The access row must be valid, otherwise a [0] array is returned. The RCLR program is shown below:

```
; Listing 9.21.  Program RCLR to recall a jagged matrix row.
;
; Version 1
; Date 12/14/90
;
; INPUT:
;           Jagged matrix
;           Accessed row
;
; OUTPUT:
;           Array representing accessed jagged matrix row,
;     OR    [ 0 ] if the value of the accessed row is out of range
;
«
0 → JM ROW LEN                       ; assign parameters to local variables
                                     ; jagged matrix to variable JM
                                     ; insertion row to variable ROW
                                     ; 0 (dummy) to variable LEN
```

```
    «
    JM SIZE 'LEN' STO                  ; get the jagged matrix size and store it in variable LEN
    IF 'ROW > 0 AND ROW ≤ LEN'         ; is the value of ROW valid?
    THEN                               ; Yes! Extract the sought row
          JM ROW GET
    ELSE                               ; No! push an empty array in the stack
          [ 0 ]
    END
    »
»
```

Here are a couple of examples for recalling jagged matrix rows. The first example shows the typically successful call to program RCLR --- the second row of a three-row jagged matrix is recalled:

INPUT STACK	**OUTPUT STACK**
2: { [1 2 3] [33 33] [5 6] }	
1: 2	1: [33 33]

The second example illustrates the effect of a negative row number:

INPUT STACK	**OUTPUT STACK**
2: { [1 2 3] [33 33] [5 6] }	
1: -2	1: [0]

The same result is obtained if the row number was zero or greater than 3.

10

Strings

Strings are special object types that store and retrieve readable text. Since the HP48SX is a number crunching machine and not a text processing one, the built-in string functions are kept to a minimum. This chapter presents a library of string processing routines. There are two reasons for including this chapter in this book: educational and practical. The educational aspect enables you to study more examples of RPL programs and how they are documented. The practical part provides you with sets of routines that offer the lacking string functions. However, these routines are NOT the fastest in the world. If speed is critical, you might want to consider string manipulation routines offered by independent vendors. The *HP48 Programmers Toolkit*, by James Donnelly, is an example.

The string routines included in this chapter perform a variety of tasks. They are:

Routine	Purpose
DELSTR	Deletes part of a string
INSTR	Inserts a substring in a string
IPOS	Offers a version of POS with an offset
ITRNSL	Performs case-insensitive string translation
LEFT	Extracts the left side of a string
LOCASE	Converts the characters of a string into lowercase
LTRIM	Trims the left part of a string
PADLF	Pads the left side of a string
PADEND	Pads both ends of a string
PARDRT	Pads the right side of a string
REPSTR	Creates a string by multiple concatenation of a substring
REVSTR	Reverses the characters of a string

RIGHT	Extracts the right side of a string
RTRIM	Trims the right side of a string
TRIMEND	Trim both ends of a string
TRNSL	Performs case sensitive string translation
UPCASE	Converts the characters of a string into uppercase

DELSTR

Purpose

Deletes a substring by specifying the first and last characters. If the last character exceeds the actual string size, the trailing part of the string is deleted.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
3: *old string*	3:
2: *first character*	2:
1: *last character*	1: *new string*

Example

The following example shows the stack before invoking DELSTR. It contains the string "Hello World!" and the indices 3 and 9 for the first and last deleted characters, respectively. The output stack shows the resulting string "Held!".

STACK INPUT	STACK OUTPUT
3: "Hello World!"	3:
2: 3	2:
1: 9	1: "Held!"

The second example presents a variation over the first one. The index of the last deleted character is 100, a value exceeding the actual size of the string. The output stack shows how the trailing part of the input string was clipped.

STACK INPUT	STACK OUTPUT
3: "Hello World!"	3:
2: 3	2:
1: 100	1: "He"

Program

The pseudo-code for DELSTR is as follows:

Input: A string S with the indices, F and L, for the first and last deleted characters, respectively.

1. Extract the leading F-1 characters.

2. Extract the substring from character L+1 to the end of the string.
3. Concatenate the leading and trailing substrings.

The listing for DELSTR is shown below:

```
; Listing 10.1. DELSTR version 1.0
«
0
→ FIRST LAST LEN                    ; store the indices to first and last characters in the local
                                    ; variables FIRST and LAST.  Zero a dummy 0 in variable
                                    ; LEN
      «
      DUP                           ; duplicate string to obtain its size
      SIZE 'LEN' STO                ; assign the string size to local variable LEN
      LASTARG                       ; recall string
      1                             ; extract the leading FIRST-1 characters
      FIRST 1 -
      SUB
      SWAP                          ; swap with duplicate of original string
      LAST 1 +                      ; extract from character LAST+1 to the end
      LEN
      SUB
      +                             ; concatenate the leading and trailing string fragments

      »
»
```

Local Variables

FIRST The index of the first deleted character. Must be greater than zero.

LAST The index of the last deleted character. Must be greater than zero.

LEN The length of the original string.

INSTR

Purpose

Inserts a string into another after a specified character index.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
3: *inserted string*	3:
2: *main string*	2:
1: *insertion index*	1: *new string*

Example

The next example shows the insertion of the string "joyful " after the sixth character of string "Hello World!":

STACK INPUT	STACK OUTPUT
3: "joyful "	3:
2: "Hello World!"	2:
1: 6	1: "Hello joyful World!"

Program

The pseudo-code for INSTR is as follows:

Input: the main string S, the inserted substring U, and the insertion index I.

1. LEN = size of the main string.
2. S1 = extracted leading I characters of S.
3. S2 = extracted trailing LEN - I characters of S
4. S = S1 + U + S2

The listing for INSTR is shown below:

```
; Listing 10.2.  INSTR version 1.0
«
OVER                                    ; copy main string into level 1
SIZE                                    ; obtain the size of the main string
→ INDEX LEN                             ; store the insertion index and the string size into the
                                        ; local variables INDEX and LEN, respectively
      «
      IF 'INDEX>0'                      ; is INDEX positive?
      THEN
            DUP                         ; duplicate the main string
            1 INDEX SUB                 ; extract the leading N characters
            SWAP                        ; swap the other copy of the main string into level 1
            INDEX 1 + LEN SUB           ; extract the characters from INDEX+1 to the end of the string
            SWAP                        ; swap substrings such that the leading portion
                                        ; of the original string is in level 1 and the
                                        ; tail portion is in level 2
            3 PICK                      ; pick the inserted substring from level 3
            +                           ; concatenate the leading portion of the main string with the
                                        ; inserted string
            SWAP                        ; swap the tail end of the original string into level 1
            +                           ; concatenate the rest
            SWAP                        ; swap copy of inserted string into level 1
            DROP                        ; drop copy of inserted string from the stack
      ELSE                              ; just concatenate strings
            +                           ; concatenate inserted and main strings (in that order)
      END
      »
»
```

Local Variables

INDEX The insertion index.

LEN The size of the main string.

IPOS

Purpose

Returns the position of a substring in a string, given the index of the first character to be searched. This is a version of the built-in POS function with an offset.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
3: *string*	3:
2: *substring*	2:
1: *first character index*	1: *substring position*

Example

The following example scans the string "Haris Paris" for the occurrence of the substring "aris" starting at the 3rd character. The result is 8, the location of the second occurrence of "aris", since the first occurrence is bypassed by the offset of 3.

STACK INPUT	STACK OUTPUT
3: "Haris Paris"	3:
2: "aris"	2:
1: 3	1: 8

Program

The pseudo-code for IPOS is as follows:

Input: the string S, the substring U, and the index of the first search character J.

1. IF J is greater than 1 then delete the leading J-1 characters of string S.
2. Locate substring U in string S. Assign result to I.
3. If I is positive (that is, a match is found) and J is greater than 1 (that is, the leading characters of the original string have been clipped) add J-1 to the result of step 2.

The listing for DELSTR is shown next:

```
; Listing 10.3.  IPOS version 1.0
«
0
→ J I                                   ; store the index of the first search character in
                                        ; the local variable J.  Store a dummy 0 in variable I
      «
      IF
            'J>1'                       ; is J greater than 1?
      THEN
            SWAP                        ; swap string to level 1
            1 J 1 - DELSTR              ; delete first J-1 characters
            SWAP                        ; swap string to level 2
      END
      POS                               ; find position of substring in string
      'I' STO                           ; assign result to local variable I
      I                                 ; push I onto the stack
      IF
            'I>0 AND J>1'               ; need to adjust result of POS?
      THEN
            J + 1 -                     ; add J-1
      END
      »
»
```

Subprograms Used

DELSTR

Local Variables

I The result of calling POS.

J The index of the first search character.

ITRNSL

Purpose

Translates a string by replacing parts of it with other strings. The translation is case insensitive.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
5: *client string*	5:
4: *search-string*	4:
3: *replace-string*	3:
2: *first search character*	2:
1: *max. # of translations*	1: *translated string*

Example

Given the string "X=3.14+3.14*Y*SQR(3.14)", the search-string "3.14", and the replace-string "PI". The translation of the client string can occur in several ways. In the first example, all of the occurrences of the search-string are replaced, leading to a full string translation. The index of the first search character is set to 1 to translate the entire string. Moreover, the value for the maximum number of translations is assigned a high number (if you want to be even more sure, use a value in the thousands or even millions!). The result is string "X=PI+PI*Y*SQR(PI)":

STACK INPUT	STACK OUTPUT
5: "X=3.14+3.14*Y*SQR(3.14)"	5:
4: "3.14"	4:
3: "PI"	3:
2: 1	2:
1: 9	1: "X=PI+PI*Y*SQR(PI)"

In the second example, the first occurrence of "3.14" is not translated. This is done by supplying a value for the first search character that in the range of 4 to 8. The maximum number of translation is still a large number. The result is string "X=3.14+PI*Y*SQR(PI)":

STACK INPUT	STACK OUTPUT
5: "X=3.14+3.14*Y*SQR(3.14)"	5:
4: "3.14"	4:
3: "PI"	3:
2: 4	2:
1: 9	1: "X=3.14+PI*Y*SQR(PI)"

To translate only the second occurrence of "3.14", the value for the first search character is also set between 4 and 8, while the maximum number of translation is set to 1.

STACK INPUT	STACK OUTPUT
5: "X=3.14+3.14*Y*SQR(3.14)"	5:
4: "3.14"	4:
3: "PI"	3:
2: 4	2:
1: 1	1: "X=3.14+PI*Y*SQR(3.14)"

Program

The pseudo-code for ITRNSL is as follows:

Input: the string to be translated, S, the sought substring, F, the replacement substring, R, the index of the first search character, Ofs, and the maximum number of translations, N.

1. Let L = size of the search-string F
2. Make the characters of F uppercase
3. Let SCOPY = uppercase copy of S
3. Let I = the position of string F in string SCOPY, such that the search starts at character Ofs
4. Loop while I and N are positive:
    A. Decrement N
    B. Delete the characters of S between I and I+L-1
    C. Insert the string R, in string S, after character I-1
    D. Delete the characters of SCOPY between I and I+L-1
    E. Insert the string R in SCOPY, after character I-1
    F. Let I = the position of string F in string SCOPY, such that the search starts at character Ofs

The listing for ITRNSL is shown next:

```
; Listing 10.4.  ITRNSL version 1.0
«
" "                                   ; insert dummy values
0 0
→ S F R OFS N S2 I LEN                ; assign information to the following local variables
                                      ; client string to S
                                      ; search-string to F
                                      ; replace-string to R
                                      ; index of the first search character to OFS
                                      ; maximum number of translations to N
                                      ; the space character to S2, the upper-case copy of S
                                      ; 0 (dummy value) to search index I
                                      ; 0 (dummy value) to size of search-string to LEN
      «
      F                               ; obtain the size of string F
      SIZE
      'LEN' STO                       ; store result in LEN
      F                               ; make F into uppercase string
      UPCASE
      'F' STO                         ; update string F
      S                               ; make S into uppercase string
      UPCASE
      'S2' STO                        ; store upper-case S in S2
      S2 F OFS                        ; find position of F in S2, starting at character OFS
      IPOS
      'I' STO                         ; store result of IPOS in I
      WHILE                           ; loop while an F string was found in S, AND
                                      ; while the number of translations has not exceeded the
                                      ; maximum specified
            'I>0 AND N>0'
      REPEAT
            -1                        ; decrement the number of translations
            'N' STO+
            S                         ; delete F string at character I
            I
            DUP
            LEN
            +
            1
            -
            DELSTR
            R                         ; insert R string after character I-1
            SWAP
            I
            1
            -
            INSTR                     ; store updated string in S
            'S' STO
                                      ; perform the same alteration on string S2
            S2                        ; delete F string at character I
            I
            DUP
            LEN
            +
            1
            -
            DELSTR
            R                         ; insert R string after character I-1
            SWAP
```

```
            I
            1
            -
            INSTR                       ; store updated string in S2
            'S2' STO
            S2 F OFS                    ; find position of F in S2, starting at character OFS
            IPOS
            'I' STO                     ; store result of IPOS in I
        END
        S                               ; push S in the stack
        »
»
```

Subprograms Used

DELSTR Deletes the search string.

INSTR Inserts the replacement string.

IPOS Finds the position of a substring in a string.

UPCASE Converts to upper-case string.

Local Variables

F The search-string.

I The result of the IPOS subprogram.

LEN The size of the search-string.

N The maximum number of translations.

OFS The index of the first search character.

R The replace-string.

S The client string.

S2 An upper-case copy of the client string

LEFT

Purpose

Extracts a number of leftmost characters of a string.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
2: *old string*	2:
1: *num*	1: *new string*

Example

The next example shows the extraction of the five leftmost characters of string "Hello World!":

STACK INPUT	STACK OUTPUT
2: "Hello World!"	2:
1: 5	1: "Hello"

Program

The pseudo-code for LEFT is as follows:

Input: the string S and the number of leading characters to extract, N.

1. Extract the N leading number of characters.

The listing for LEFT is shown below:

```
; Listing 10.5.  LEFT version 1.0
«
1                                       ; insert 1 into the stack
SWAP                                    ; swap 1 and the number of characters to extract
SUB                                     ; extract the sought substring
»
```

LOCASE

Purpose

Converts the uppercase characters of a string into lowercase.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
1: *original string*	1: *lowercase string*

Example

The following example shows how a call to LOCASE converts the string "Hello World!" into "hello world!".

STACK INPUT	STACK OUTPUT
1: "Hello World!"	1: "hello world!"

Program

The pseudo-code for LOCASE is as follows:

Input: the string S.

1. Let LEN = the size of string S.
2. Let U be the resulting string, initialized as an empty string.
3. Loop FOR I = 1 TO LEN perform the following steps:
    A. Extract the character at index I.
    B. Obtain the ASCII code for the extracted character.
    C. If ASCII code is in range 65 to 90 then ASCII = ASCII + 32.
    D. Convert ASCII code to character.
    E. Concatenate character with string U.
4. String U contains the result.

The listing for LOCASE is shown below:

```
; Listing 10.6.  LOCASE version 1.0
«
DUP                                     ; duplicate string
SIZE                                    ; obtain string size and assign it to the local variable LEN
→ LEN
      «
      ""                                ; push the resulting string (initially empty) onto the stack
      1 LEN                             ; push FOR loop limits into the stack
      FOR I                             ; I is the loop control variable
            OVER                        ; get a copy of the string from level 2
            I I SUB                     ; extract character at index I

            NUM                         ; convert character to ASCII code
            → C                         ; store ASCII code in local variable C
                  «
                  C                     ; push ASCII code onto the stack
                  IF
                    'C>64 AND C<91'     ; is the ASCII code that of a uppercase character?
                  THEN
                        32              ; add 32 to convert code to that of the corresponding
                        +               ; lowercase character
                  END
                  CHR                   ; convert ASCII code to character
                  »
            +                           ; concatenate character with resulting string
      NEXT
      SWAP                              ; swap original string into level 1
      DROP                              ; drop original string from stack
      »
»
```

Local Variables

C The ASCII code of the extracted characters.

LEN The size of the string.

I The FOR-NEXT loop control variable.

LTRIM

Purpose

Trims characters from the left side of a string. The trimmed characters are specified in a separate string.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
2: *client string*	2:
1: *trim-characters*	1: *left-trimmed string*

Example

The following example shows how string " +-+-Hello" is stripped from leading spaces, plus, and minus characters. The trim-character string " +-" is specified. The same result is achieved if the trim-character string is any combination of the three deleted characters:

STACK INPUT	STACK OUTPUT
2: " +-+-Hello"	2:
1: "+- "	1: "Hello"

Program

The pseudo-code for LTRIM is as follows:

Input: the client string S and the trim-character string T.

1. Let L = the size of string S
2. Let I = 0, the character index
3. Repeat the following steps
   - A. Increment I
   - B. Let C = the I'th character of S
   - C. Let J = the position of C in string T

   Until I = L, or J = 0
4. Let S = characters of original S between I and L

The listing for LTRIM is shown below:

```
; Listing 10.7.  LTRIM version 1.0
«
OVER                                    ; get a copy of the client string into level 1
SIZE                                    ; obtain the size of the client string
0                                       ; push a dummy value for character index in the stack
0                                       ; push a dummy value for the character position in the stack
→ S TRM LEN I J                         ; assign stack data to the following local variables:
                                        ;       the client string to S
                                        ;       the trim-character string to TRM
                                        ;       the size of S to LEN
                                        ;       0 to I, the character index
                                        ;       0 to J, the character position
      «
      DO                                ; start a DO-UNTIL loop
            TRM                         ; push TRM in the stack
            'I' INCR                    ; increment the character index
            S                           ; extract the I'th character of string S
            SWAP                        ; swap the value of I in level 1
            DUP                         ; duplicate the value of I
            SUB
            POS                         ; obtain position of character in trim-character string
            'J' STO                     ; store position in J
      UNTIL 'I==LEN OR J==0'            ; test the following condition:
                                        ; has I reached the end of the client string?
                                        ; OR
                                        ; was the last extracted character NOT in the
                                        ; trim-character string?
      END
      S                                 ; push client string in level 1
      I LEN SUB                         ; extract characters from index I to the end
      »
»
```

Local Variables

I The character index.

J The character position.

LEN The size of the client string.

S The client string.

TRM The trim-character string.

PADLF

Purpose

Pads the left side of a string with multiple copies of a padding string (or character).

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
3: *client string*	3:
2: *padding string*	2:
1: *count*	1: *left-padded string*

Example

In this example the string "+-" is padded twice to the left side of string "HELLO", yielding string "+-+-HELLO":

STACK INPUT	STACK OUTPUT
3: "HELLO"	3:
2: "+-"	2:
1: 2	1: "+-+-HELLO"

Program

The pseudo-code for PADLF is as follows:

Input: the client string S, the building-unit for the padding string, B, and N be the number of times string B is concatenated to the left side of string S.

1. Let P = N concatenation of string B
2. Let S = P + S

The listing for PADLF is shown below:

```
; Listing 10.8.  PADLF version 1.0
«
REPSTR                                  ; call REPSTR to created the complete padding string
SWAP                                    ; swap the client string into level 1
+                                       ; concatenate strings
»
```

Subprograms Used

REPSTR Used to create the complete padding string.

PADEND

Purpose

Pads the both sides of a string with multiple copies of a padding string (or character).

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
3: *client string*	3:
2: *padding string*	2:
1: *count*	1: *padded string*

Example

In this example the string "+-" is padded twice on both side of string "HELLO", yielding string "+-+-HELLO+-+-":

STACK INPUT	STACK OUTPUT
3: "HELLO"	3:
2: "+-"	2:
1: 2	1: "+-+-HELLO+-+-"

Program

The pseudo-code for PADEND is as follows:

Input: the client string S, the building-unit for the padding string, B, and N be the number of times string B is concatenated to the left side of string S.

1. Let P = N concatenation of string B
2. Let S = P + S + P

The listing for PADEND is shown below:

```
; Listing 10.9.  PADEND version 1.0
«
REPSTR                                  ; call REPSTR to created the complete padding string
SWAP                                    ; swap the client string into level 1
OVER                                    ; copy padding string into level 1
+                                       ; concatenate the right padding string and the client string
+                                       ; concatenate the left padding string and the client strings
»
```

Subprograms Used

REPSTR	Used to create the complete padding string.

PADRT

Purpose

Pads the right side of a string with multiple copies of a padding string (or character).

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
3: *client string*	3:
2: *padding string*	2:
1: *count*	1: *right-padded string*

Example

In this example the string "+-" is padded twice to the right side of string "HELLO", yielding string "HELLO+-+-":

STACK INPUT	STACK OUTPUT
3: "HELLO"	3:
2: "+-"	2:
1: 2	1: "HELLO+-+-"

Program

The pseudo-code for PADRT is as follows:

Input: the client string S, the building-unit for the padding string, B, and N be the number of times string B is concatenated to the right side of string S.

1. Let P = N concatenation of string B
2. Let S = S + P

The listing for PADRT is shown below:

```
; Listing 10.10.  PADRT version 1.0
«
REPSTR                                  ; call REPSTR to created the complete padding string
+                                       ; concatenate strings
»
```

Subprograms Used

REPSTR Used to create the complete padding string.

REPSTR

Purpose

Creates a string by concatenating a specified number of duplicates of smaller strings.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
2: *building string*	2:
1: *count*	1: *resulting string*

Example

The following example illustrates how the string "BABABABABA" is created by chaining 5 "BA" strings:

STACK INPUT	STACK OUTPUT
2: "BA"	2:
1: 5	1: "BABABABABA"

Program

The pseudo-code for REPSTR is as follows:

Input: the building string S and the number of times, N, to concatenate it.

1. Let the resulting string R = S
2. Loop N-1 times
   Let R = R + S (concatenate resulting and building strings)
3. The string R contains the result.

The listing for REPSTR is shown next:

```
; Listing 10.11.  REPSTR version 1.0
«
SWAP                                    ; swap building string to level 1
→ C                                     ; store building string in the local variable C
      «
      C                                 ; push C in the stack.  This becomes the resulting string
      SWAP                              ; swap count into level 1
      1 -                               ; subtract 1 from count
      1                                 ; push 1 in the stack
      SWAP                              ; swap 1 and count-1
      START                             ; begin fixed iteration loop
            C                           ; push a copy of the building string in the stack
            +                           ; concatenate with the resulting string
      NEXT                              ; end of fixed loop
      »
»
```

Local Variables

C The building string.

REVSTR

Purpose

Reverses the characters of a string.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
1: *old string*	1: *new string*

Example

The next example shows the reversal of the string "Hello World!":

STACK INPUT	STACK OUTPUT
1: "Hello World!"	1: "!dlroW olleH"

Program

The pseudo-code for REVSTR is as follows:

Input: the string S.

1. Let LEN = the size of string S.
2. Let U be the resulting string, initialized as an empty string.
3. Loop FOR I = LEN TO 1 perform the following steps:
    A. Extract the character at index I.
    B. Concatenate character with string U.
4. String U contains the result.

The listing for REVSTR is shown below:

```
; Listing 10.12.  REVSTR version 1.0
«
DUP                                     ; duplicate string
SIZE                                    ; obtain string size and assign it to the local variable LEN
→ LEN
      «
      ""                                ; push the resulting string (initially empty) onto the stack
      LEN 1                             ; push FOR loop limits onto the stack
      FOR I                             ; I is the loop control variable
            OVER                        ; get a copy of the string from level 2
            I I SUB                     ; extract character at index I
            +                           ; concatenate character with resulting string
            -1                          ; specify loop step size
      STEP
      SWAP                              ; swap original string into level 1
      DROP                              ; drop original string from stack
      »
»
```

Local Variables

LEN The size of the string.

I The FOR-NEXT loop control variable.

RIGHT

Purpose

Extracts the rightmost characters of a string.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
2: *old string*	2:
1: *num*	1: *new string*

Example

The next example shows the extraction of the six rightmost characters of string "Hello World!""

STACK INPUT	STACK OUTPUT
2: "Hello World!"	2:
1: 6	1: "World!"

Program

The pseudo-code for RIGHT is as follows:

Input: the string S and the number of trailing characters to extract, N.

1. Let LEN = the size of string S.
2. Extract the substring between the character indices (LEN - N + 1) and LEN.

The listing for RIGHT is shown below:

```
; Listing 10.13.  RIGHT version 1.0
«
OVER                                    ; get a copy of the source string
SIZE                                    ; obtain string size
SWAP                                    ; swap count to level 1
- 1 +                                   ; calculate size - count + 1
OVER                                    ; get a copy of the source string
SIZE                                    ; obtain the string size
SUB                                     ; extract the sought substring
»
```

RTRIM

Purpose

Trims characters from the right side of a string. The trimmed characters are specified in a separate string.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
2: *client string*	2:
1: *trim-characters*	1: *right-trimmed string*

Example

The following example shows how string "Hello+-+- " is stripped from trailing spaces, plus, and minus characters. The trim-character string " +-" is specified. The same result is achieved if the trim-character string is any combination of the three specified characters:

STACK INPUT	STACK OUTPUT
2: "Hello+-+- "	2:
1: "+- "	1: "Hello"

Program

The pseudo-code for RTRIM is as follows:

Input: the client string S and the trim-character string T.

1. Let L = the size of string S
2. Let I = L, the character index
3. Repeat the following steps
   - A. Decrement I
   - B. Let C = the I'th character of S
   - C. Let J = the position of C in string T

   Until I = 1, or J = 0
4. Let S = characters of original S between 1 and I

The listing for RTRIM is shown below:

```
; Listing 10.14.  RTRIM version 1.0
«
OVER                                    ; get a copy of the client string into level 1
SIZE                                    ; obtain the size of the client string
DUP                                     ; duplicate the size
1 +                                     ; increment the size
0                                       ; push a dummy value for the character position in the stack
→ S TRM LEN I J                         ; assign stack data to the following local variables
                                        ; the client string to S
                                        ; the trim-character string to TRM
                                        ; the size of S to LEN
                                        ; LEN+1 to I, the character index
                                        ; 0 to J, the character position
      «
      DO                                ; start a DO-UNTIL loop
            TRM                         ; push TRM in the stack
            'I'                         ; decrement the character index
            DECR
            S                           ; extract the I'th character of string S
            SWAP                        ; swap the value of I into level 1
            DUP                         ; duplicate the value of I
            SUB
            POS                         ; obtain position of character in trim-character string
            'J' STO                     ; store position in J
      UNTIL 'I==1 OR J==0'              ; test the following condition
                                        ; Has I reached the beginning of the client string?
                                        ; OR
                                        ; Was the last extracted character NOT in the trim-character
                                        ; string?
      END
      S                                 ; push client string in level 1
      1 I SUB                           ; extract the first I characters
      »
»
```

Local Variables

I The character index.

J The character position.

LEN The size of the client string.

S The client string.

TRM The trim-character string.

TRIMEND

Purpose

Trims characters from the both sides of a string. The trimmed characters are specified in a separate string.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
2: *client string*	2:
1: *trim-characters*	1: *trimmed string*

Example

The following example shows how string " +Hello+-+- " is stripped from leading and trailing spaces, plus, and minus characters. The trim-character string " +-" is specified. The same result is achieved if the trim-character string is any combination of the three specified characters:

STACK INPUT	STACK OUTPUT
2: " +Hello+-+- "	2:
1: "+- "	1: "Hello"

Program

The pseudo-code for TRIMEND is as follows:

Input: the client string S and the trim-character string T.

1. Trim leading characters
2. Trim trailing characters

The listing for TRIMEND is shown next:

```
; Listing 10.15.  TRIMEND version 1.0
«
DUP2                                    ; duplicate objects in levels 1 and 2
LTRIM                                   ; invoke the LTRIM routine
SWAP                                    ; swap result of LTRIM with trim-character string
RTRIM                                   ; invoke the RTRIM routine
SWAP                                    ; bring the copy of original client string into level 1
DROP                                    ; drop it, leaving result in level 1
»
```

Suprograms Used

LTRIM trim the leading characters.

RTRIM trim the trailing characters.

TRNSL

Purpose

Translates a string by replacing parts of it with other strings. The translation is case sensitive.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
5: *client string*	5:
4: *search-string*	4:
3: *replace-string*	3:
2: *first search character*	2:
1: *max. # of translations*	1: *translated string*

Example

Given the string "X=3.14+3.14*Y*SQR(3.14)", the search-string "3.14", and the replace-string "PI". The translation of the client string can occur in several ways. In the first example, all of the occurrences of the search-string are replaced, leading to a full string translation. The index of the first search character is set to 1 to translate the entire string. Moreover, the value for the maximum number of translations is set to a high value (if you want to be even more sure, use a value in the thousands or even millions!). The result is string "X=PI+PI*Y*SQR(PI)":

STACK INPUT	STACK OUTPUT
5: "X=3.14+3.14*Y*SQR(3.14)"	5:
4: "3.14"	4:
3: "PI"	3:
2: 1	2:
1: 9	1: "X=PI+PI*Y*SQR(PI)"

In the second example, the first occurrence of "3.14" is not translated. This is done by supplying a value for the first search character that in the range of 4 to 8. The maximum number of translation is still a large number. The result is string "X=3.14+PI*Y*SQR(PI)":

STACK INPUT	STACK OUTPUT
5: "X=3.14+3.14*Y*SQR(3.14)"	5:
4: "3.14"	4:
3: "PI"	3:

2: 4	2:
1: 9	1: "X=3.14+PI*Y*SQR(PI)"

To translate only the second occurrence of "3.14", the value for the first search character is also set between 4 and 8, while the maximum number of translation is set to 1.

STACK INPUT	STACK OUTPUT
5: "X=3.14+3.14*Y*SQR(3.14)"	5:
4: "3.14"	4:
3: "PI"	3:
2: 4	2:
1: 1	1: "X=3.14+PI*Y*SQR(3.14)"

Program

The pseudo-code for TRNSL is as follows:

Input: the string to be translated, S, the sought substring, F, the replacement substring, R, the index of the first search character, Ofs, and the maximum number of translations, N.

1. Let L = size of the search-string F
2. Let I = the position of string F in string S, such that the search starts at character Ofs
3. Loop while I and N are positive:
   A. Decrement N
   B. Delete the characters of S between I and I+L-1
   C. Insert the string R after character I-1
   D. Let I = the position of string F in string S, such that the search starts at character Ofs

The listing for TRNSL is shown below:

```
; Listing 10.16.  TRNSL version 1.0
«
0 0                                   ; insert dummy values
→ S F R OFS N I LEN                   ; assign information to the following local variables
                                      ; client string to S
                                      ; search-string to F
                                      ; replace-string to R
                                      ; index of the first search character to OFS
                                      ; maximum number of translations to N
                                      ; 0 (dummy value) to search index I
                                      ; 0 (dummy value) to size of search-string to LEN
      «
      F                               ; obtain the size of string F
      SIZE
      'LEN' STO                       ; store result in LEN
      S F OFS                         ; find position of F in S, starting at character OFS
      IPOS
      'I' STO                         ; store result of IPOS in I
      WHILE                           ; loop while an F string was found in S, AND
                                      ; while the number of translations has not exceeded the
```

```
                                        ; maximum specified
        'I>0 AND N>0'
    REPEAT
        -1                              ; decrement the number of translations
        'N' STO+
        S                               ; delete F string at character I
        I
        DUP
        LEN + 1 -
        DELSTR
        R                               ; insert R string after character I-1
        SWAP
        I 1 -
        INSTR                           ; store updated string in S
        'S' STO
        S F OFS IPOS                    ; find position of F in S, starting at character OFS
        'I' STO                         ; store result of IPOS in I
    END
    S                                   ; push S in the stack
    »
»
```

Subprograms Used

DELSTR	Deletes the search string.
INSTR	Inserts the replacement string.
IPOS	Finds the position of a substring in a string.

Local Variables

F	The search-string.
I	The result of the IPOS subprogram.
LEN	The size of the search-string.
N	The maximum number of translations.
OFS	The index of the first search character.
R	The replace-string.
S	The client string.

UPCASE

Purpose

Converts the lowercase characters of a string into uppercase.

Stack I/O

The general stack usage is shown below:

STACK INPUT	STACK OUTPUT
1: *original string*	1: *uppercase string*

Example

The following example shows how a call to UPCASE converts the string "Hello World!" into "HELLO WORLD!".

STACK INPUT	STACK OUTPUT
1: "Hello World!"	1: "HELLO WORLD!"

Program

The pseudo-code for UPCASE is as follows:

Input: the string S.

1. Let LEN = the size of string S.
2. Let U be the resulting string, initialized as an empty string.
3. Loop FOR I = 1 TO LEN
    A. Extract the character at index I.
    B. Obtain the ASCII code for the extracted character
    C. If ASCII code is in range 97 to 122 then ASCII = ASCII - 32.
    D. Convert ASCII code to character
    E. Concatenate character with string U

The listing for UPCASE is shown below:

```
; Listing 10.17.  UPCASE version 1.0
«
DUP                                     ; duplicate string
SIZE                                    ; obtain string size and assign it to the local variable LEN
→ LEN
      «
      ""                                ; push the resulting string (initially empty) into the
                                        ; stack
      1 LEN                             ; push FOR loop limits into the stack
      FOR I                             ; I is the loop control variable
            OVER                        ; get a copy of the string from level 2
            I I SUB                     ; extract character at index I
            SUB
            NUM                         ; convert character to ASCII cod
            → C                         ; store ASCII code in local variable C
                  «
                  C                     ; push ASCII code onto the stack
                  IF
                   'C>96 AND C<123'     ; is the ASCII code that of a lowercase
                                        ; character?
                  THEN
                    32 -                ; subtract 32 to convert code to that of the corresponding
                                        ; uppercase character
                  END
                  CHR                   ; convert ASCII code to character
                  »
            +                           ; concatenate character with resulting string
      NEXT
      SWAP                              ; swap original string into level 1
      DROP                              ; drop original string from stack
      »
»
```

Local Variables

C The ASCII code of the extracted characters.

LEN The size of the string.

I The FOR-NEXT loop control variable.

Notes

Index