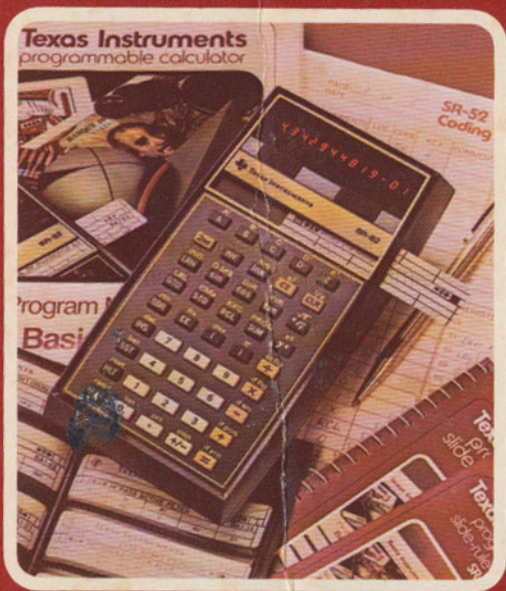


# Texas Instruments programmable slide-rule calculator SR-52



OWNER'S  
MANUAL



### **IMPORTANT**

Record the serial number from the bottom of the calculator and purchase date in the space below. The serial number is identified by the words "SERIAL NO." on the bottom case. Always reference this information in any correspondence.

#### **SR-52**

<b>Model No.</b>	<b>Serial No.</b>	<b>Purchase Date</b>
------------------	-------------------	----------------------

### **Toll-Free Telephone Assistance**

For assistance with your SR-52 calculator, call one of the following toll-free numbers:

800-527-4980 (within all contiguous United States except Texas)


800-492-4298 (within Texas)

See Appendix A and back cover for further information on service.

# TABLE OF CONTENTS

Section		Page
I.	INTRODUCTION. . . . .	1
	Features . . . . .	2
	Modes of Operation . . . . .	4
	Manual Calculations. . . . .	5
	Executing Programs. . . . .	8
	Creating Your Own Programs. . . . .	12
	Method 1 . . . . .	12
	Method 2 . . . . .	15
II.	ENTERING AND DISPLAYING NUMBERS . . .	18
	Entering Numbers. . . . .	18
	Entering $\pi$ . . . . .	19
	Scientific Notation. . . . .	20
	Advanced Effects and Uses of the <b>EE</b> Key. . . . .	21
	Clearing Incorrect Number Entries . . . . .	23
	Clearing a Calculation . . . . .	24
	Display Control . . . . .	24
	Error Conditions . . . . .	28
III.	ARITHMETIC CALCULATIONS . . . . .	29
	Basic Operations . . . . .	29
	Chained Operations . . . . .	29
	A Special Type of Operation Chain . . . . .	30
	Parentheses . . . . .	31
	The Use of <b>≡</b> in Complicated Expressions . . . . .	34
IV.	SPECIAL FUNCTIONS . . . . .	35
	Functions of a Single Variable . . . . .	35
	Functions of Two Variables. . . . .	39
	Angular Unit Conversions . . . . .	41
	Coordinate (Polar/Rectangular) Conversions . . . . .	44
V.	ALGEBRAIC NOTATION: MORE ABOUT PENDING OPERATIONS . . . . .	46
	The Algebraic Hierarchy . . . . .	46
	Keeping Track of Display Register Contents . . . . .	49
VI.	MEMORY REGISTERS. . . . .	51
	Storing Quantities in Memory. . . . .	51
	Recalling Quantities from Memory . . . . .	53

## TABLE OF CONTENTS (continued)

Section		Page
VI. (cont'd)	Clearing the Data Memory . . . . .	55
	Direct Register Arithmetic . . . . .	55
	Memory/Display Exchange. . . . .	57
	Supplying Missing Operands with Memory Functions . . . . .	58
VII.	RUNNING PRERECORDED PROGRAMS . . . . .	60
	Reading a Magnetic Card . . . . .	60
	Executing a Program . . . . .	64
VIII.	GENERAL PROGRAMMING INSTRUCTIONS. . . . .	65
	Elements of Program Execution . . . . .	66
	Mechanics of Programming . . . . .	69
	Development of Programming Style. . . . .	71
IX.	ELEMENTARY PROGRAMMING . . . . .	72
	Using Labels . . . . .	72
	Using Run and Halt . . . . .	75
	Entering Your Program . . . . .	76
	Recording a Magnetic Card . . . . .	77
	Editing Programs . . . . .	78
	Displaying the Program . . . . .	78
	Replacing an Instruction with Another. . . . .	81
	Deleting an Instruction. . . . .	81
	Inserting an Instruction . . . . .	82
	Single-Step and Backstep . . . . .	82
	Practice Problems. . . . .	84
X.	TRANSFER INSTRUCTIONS . . . . .	88
	Unconditional Transfer Instructions. . . . .	88
	Conditional Transfers (Branching Instructions) . . . . .	89
	Setting and Resetting Program Flags . . . . .	95
	Decrement and Skip on Zero (DSZ) . . . . .	98
XI.	SUBROUTINES . . . . .	101
	Calling a Subroutine. . . . .	101
	Labeling a Subroutine . . . . .	106
	Avoid Using  in Subroutines . . . . .	106
	The Return Instruction. . . . .	107
	Subroutine Practice Problems . . . . .	110

## TABLE OF CONTENTS (continued)

<i>Section</i>		<i>Page</i>
XII.	INDIRECT INSTRUCTIONS . . . . .	115
	Indirect Data-Register Instructions . . . . .	115
	Indirect Program-Transfer Instructions . . . . .	127
XIII.	PRINTER CONTROL . . . . .	132
	Listing a Program . . . . .	133
	Printing Data . . . . .	134
	Paper Advancement . . . . .	135
	Programming Implications . . . . .	136
	Trace Operation . . . . .	136
	Running Long Programs . . . . .	136
XIV.	A COMPLETE SAMPLE PROGRAM . . . . .	137
	Define the Problem . . . . .	137
	Develop a Flow Diagram . . . . .	137
	Convert Flow Diagram to Keystrokes . . . . .	140
	Enter the Program . . . . .	147
	Check and Edit Program . . . . .	148
	Document User Instructions . . . . .	150
	Running the Program . . . . .	150
	Optimizing a Program . . . . .	153
XV.	AN ADVANCED SAMPLE PROGRAM OF MATHEMATICAL MODELING . . . . .	155
APPENDIX A.	. . . . .	173
	Battery and AC Adapter/Charger Operation . . . . .	173
	Battery Pack Replacement . . . . .	174
	Caring for Magnetic Cards . . . . .	175
	Handling Cards . . . . .	175
	Cleaning Cards . . . . .	176
	Writing on Cards . . . . .	177
	Using the Head-Cleaning Card . . . . .	177
	In Case of Difficulty . . . . .	178
	If You Have Questions or Need Assistance . . . . .	180
APPENDIX B—Error Conditions	. . . . .	181
GLOSSARY . . . . .		185
INDEX . . . . .		196



## I. INTRODUCTION

You have just purchased a highly advanced pocket calculator. That much you probably already know. What you may not fully appreciate is just how much the SR-52 is going to increase your ability, speed, and pleasure in problem solving. Beyond a mere calculator, the SR-52 is a true problem-solving machine. The problem-solving capability comes from the unique combination of three elements: portability, programmability, and programming method. These elements allow you to always have at hand the means necessary for solving those complex problems you regularly encounter. The SR-52, through portability, provides ready access to the problem-solving machinery — the programs. Conveniently stored on magnetic cards, they free you from the need to remember equations, constants, numerical algorithms, and generally from the lengthy mechanical process of obtaining an answer to a well formulated problem. The method of programming allows you, with or without prior programming experience, to solve involved problems easily, creating your own programs in a manner equivalent to the mathematical sequence you use in stating the problem.

Regardless of your previous programming experience, you are in for a pleasant surprise. Even if you have no prior programming experience, you will find the simple method of programming the SR-52 is not only easy but fun. If you are an experienced user of computers, you can enjoy many features previously unavailable except on full-scale computers.

The four-function electronic calculators have already revolutionized the way people perform arithmetic; the programmable hand calculator will have at least that large an impact on the way professional people solve problems. Remember the SR-52 is a problem-solving machine. As your familiarity with this calculator increases, its usefulness to you will also increase. Perhaps these remarks will communicate what we at Texas Instruments believe to be the significance of the SR-52 to you — and our pride in offering it.

## FEATURES

The following features are indications of the versatility and performance of your SR-52.

**224 Program Storage Locations** – Simply place your calculator in the learn mode and it will remember up to 224 calculation steps and numbers which can be repeated on your command.

**72 Labels** – These labels permit quick identification or transfer to any program segment.

**23 Preprogrammed Key Functions** – Trigonometric and logarithmic functions, powers and roots, factorials, reciprocals, conversions, and pi are available directly from the keyboard.

**20 Addressable Memory Registers** – Store and recall data or perform direct register arithmetic: Addition, subtraction, multiplication or division with any memory register without affecting the calculation in progress.

**12-Digit Precision** – All registers, internal and addressable, provide 12-digit numbers with power-of-ten exponent.

**10-Digit Display** – Format controllable 10-digit display with a scientific notation range from  $10^{-99}$  to  $10^{99}$ . The 12-digit display register contents are rounded to ten digits for display.

**10 User-Definable Keys** – These allow user-definable functions to be executed simply by pressing the appropriate label key.

**10 Internal Processing Registers** – These are used to hold operands for calculations in progress.

**10 Logical Decision Functions** – Program your calculator to make repetitive decisions and branch to appropriate program segments automatically without interruption.



**5 Program Flags** – These flags may be set, reset, and tested under program control.

**3 Program Levels** – Two levels of subroutines may be defined, which when called by the main program or another subroutine will execute and then automatically return control to the calling routine.

**True Algebraic Entry** – Automatic processing of parentheses and conformity to the rules of algebraic hierarchy allow problems containing up to 10 pending operations to be programmed and entered in the same way they are normally written.

**Permanent Program Storage on Magnetic Cards** – A built-in magnetic card unit stores up to 224 program steps on a single card.

**Optional Printing Unit** – Enhance the versatility of your calculator with the desk printing unit to obtain a permanent record of your calculations or programs. Contact your dealer for more information.

To illustrate the power of the SR-52's algebraic system imagine that your program requires the evaluation of a complicated expression such as:

$$y = x_1 \left( x_2 + x_3 \sqrt{x_4}^{x_5} \right)$$

To make things worse, imagine that all the quantities  $x_1, x_2, x_3, x_4, x_5$  are defined to be expressions of the same complexity as  $y$ . The ten-level internal processing registers allow the SR-52 to evaluate all the quantities  $x_1$  through  $x_5$  and obtain  $y$  without storing any intermediate results in the user's addressable memory. (It can in fact do this and still have two levels to spare!)

## MODES OF OPERATION

The SR-52 may be operated in three different modes **calculate**, **run**, and **learn**.

When you turn your calculator on it is in the calculate mode. You will find that once you have mastered the use of the calculate mode you are well on your way to mastering the learn mode as well.

**Calculate Mode** — Here you manually operate the SR-52 as a general-purpose calculator. You command each and every step by keystrokes: entering numbers, performing mathematical operations, computing functions, and storing or recalling intermediate results or other data.

**Run Mode** — You can use a program designed by yourself or by someone else to solve a problem. Reading a small magnetic card (such as supplied in the Basic Library) into the SR-52 program memory enables you to tailor the calculator to perform the sequential steps necessary to solve a special problem automatically. You enter the data, start the calculation then typically wait just a few seconds for the answer to appear on the display. Not only does the RUN mode save you the labor of remembering and executing keystrokes; it makes problems solvable that previously required a full-scale computer.

**Learn Mode** — After defining the steps you would use in the calculate mode to solve a problem, you can key these steps directly into the SR-52 program memory for immediate use in the run mode. It is also possible to step through program memory and to display the instructions for editing purposes. This feature will soon encourage you to use the learn and run modes to solve even short problems which you might at first solve manually. Once you know there are no mistakes in a program, you have nearly eliminated the opportunity to make mistakes in

working the problem through miskeyed or misordered operations. To avoid the necessity for manually keying in your program each time it is needed, it can be recorded on a magnetic card for future use.

The SR-52 has been designed to be compatible with an optional printing unit. This option allows the user to permanently record calculator activities by performing several printing functions either in the calculate or run mode. In the calculate mode, the printer may be used to print any desired intermediate results or to provide a complete listing of the stored program. In the run mode, print statements encountered in the program cause automatic printing of the quantity just computed. Paper advance statements can be used to set off groups of data. These printing features allow you to run a program without the necessity of program halts which would be required to record multiple answers. Of particular importance in checking a program, the trace option prints all actual steps performed and the corresponding numerical results.

## MANUAL CALCULATIONS

By now you are probably eager to use your calculator. When you slide the ON/OFF switch to the right and press **CLR**, a single zero should appear in the display. If anything other than a single zero appears, the battery pack is probably discharged. Refer to Maintenance and Servicing Information in this manual.

The following examples will familiarize you with the basic operation of your calculator. It is a safe practice to press **CLR** prior to beginning any new problem to be sure all incomplete calculations are eliminated. However, when a problem is ended with **=**, a new problem may be entered without using **CLR**.

Example:  $37.84 + 122.09 - 10.369 = ?$

Enter	Press	Display
37.84	$+$	37.84
122.09	$-$	159.93
10.369	$=$	149.561

Notice that the numbers and operations are entered in the same order as they occur in the mathematical expression.

Example:  $16 \times 3.1689 \div 0.0018 = ?$

Enter	Press	Display
16	$\times$	16.
3.1689	$\div$	50.7024
.0018	$=$	28168.

Now try a slightly more complicated problem involving parentheses. Parentheses ensure correct execution of the operations used thus allowing you to enter problems in the order they are written.

Example:  $(9.1 + 11) \div 0.16 = ?$

Enter	Press	Display	Remarks
	$\text{CLR}$ $($	0.	
9.1	$+$	9.1	
11	$)$	20.1	Evaluates contents of parentheses
	$\div$	20.1	
.16	$=$	125.625	

To compute various mathematical functions of the displayed quantity, simply press the corresponding function key.

Example:  $\ln(2.718) = ?$

Enter	Press	Display
2.718	$\boxed{\ln x}$	.9998963157

To compute a trigonometric function it is first necessary to select whether you plan to measure angles in degrees or radians. This selection is made with the angular mode switch just under the display on the left side. Slide this switch to the right for degrees.

Example:  $\sin 30^\circ = ?$

**Angle:Deg**

Enter	Press	Display
30	$\boxed{\sin}$	0.5

Note that this operation was basically the same as in the previous problem except for the additional step of selecting the angular mode.

You have probably noticed that most of the keys on the SR-52 are doubly labeled: a label on the key itself and one just above it. This implies that these keys have two functions. The first function or meaning of the key is denoted by the label on the key. When the key is immediately preceded by pressing the  $\boxed{2nd}$  key, then the second function of that key is utilized. In this way the 45 keys of the SR-52 are able to represent 88 different functions. (Note: only the  $\boxed{INV}$  key and the  $\boxed{2nd}$  key itself have single meanings; the digits  $\boxed{1}$  through  $\boxed{9}$  actually have double functions, though not labeled.) Throughout this manual the  $\boxed{2nd}$  key is indicated or shown as a necessary prefix to all second functions. For example, the square root function will be shown  $\boxed{2nd}$   $\boxed{\sqrt{x}}$ , reminding the user that to access  $\boxed{\sqrt{x}}$ , it must be preceded by  $\boxed{2nd}$ . (If you press  $\boxed{2nd}$  by mistake, you may nullify its effect by pressing it again).

Example:  $(8 \times (67.8 + 11)) / \sqrt{3.0963} = ?$

Enter	Press	Display	Remarks
	CLR (	0	
8	X (	8.	
67.8	+	67.8	
11	) ) ÷	630.4	Completes numerator calculations
3.0963	2nd $\sqrt{x}$	1.759630643	Square root of 3.0963
	=	358.2570027	

The previous examples are only a sample of the mathematical capabilities of your SR-52 in the calculate mode. A complete and sequential description of all calculator functions follows this introduction.

## EXECUTING PROGRAMS

Now we come to the most interesting and useful aspect of the SR-52 – its ability to execute a program in the **run** mode. This is, after all, what the SR-52 is all about – executing programs to solve problems. To demonstrate this mode we have selected Compound Interest (BA1-08) from the Basic Library. Each program in the basic set has its own detailed instructions for use as well as a complete listing of the programming code. The present discussion will therefore be somewhat redundant with that provided in the documentation of the Basic Library.

Example: Compound Interest Computations

Given any three of the four variables as inputs, solve for the remaining variable in the compound interest equation  $FV = PV (1 + i/100)^n$ , where

PV = Present Value

FV = Future Value

i = Interest rate (percent) per compounding period

n = Number of Compounding Periods

To solve this problem, select the magnetic card for program BA1-08. Enter the program into the calculator's program memory: Press **CLR** **2nd** **read** and insert the magnetic card into the lower slot on the right side of the calculator so that Side A is read (Insert card: **◀A▶**). Do not restrict or hold the card after it is engaged by the drive motor. Remove the card from the calculator and insert it in the upper slot so that the card annotations are visible. After the program has been read into the SR-52 program memory, you are ready to solve some interest problems. The instructions for using this program are:

Step	Procedure	Press	Display
1	Initialize	<b>E</b>	0.00
2	Input 3 of 4 arguments (in any order):		
	Present value PV	<b>A</b>	PV
	Future value FV	<b>B</b>	FV
	Interest rate (%) $i$	<b>C</b>	$i$
	Number of Periods $n$	<b>D</b>	$n$
3	Compute unknown value:		
	Present Value PV	<b>2nd</b> <b>A'</b>	PV'
	Future Value FV	<b>2nd</b> <b>B'</b>	FV'
	Interest rate (%) $i$	<b>2nd</b> <b>C'</b>	$i'$
	Number of periods $n$	<b>2nd</b> <b>D'</b>	$n'$
4	To solve a new problem go to Step 2. You need to enter only those quantities that have changed.		

Observe that the key functions printed on the card make reference to the above instructions unnecessary after they are understood the first time. You may be wondering: How do I go from the calculate to the run mode? The answer to this is that everytime you press one of the labels **A** through **E** and **2nd A** through **2nd E**, the SR-52 changes from the calculate to the run mode and begins executing the instructions so labeled in the program memory. When it has completed execution, the SR-52 returns to the calculate mode.

Example: What is the value of \$500 after 24 months with interest compounded monthly if the annual interest rate is 5.75%?

Enter	Press	Display	Remarks
	<b>E</b>	0.00	Initialize
500	<b>A</b>	500.00	PV
5.75	$\div$	5.75	
12	<b>=</b>	.48*	Monthly rate i
	<b>C</b>	.48	i
24	<b>D</b>	24.00	n
	<b>2nd B</b>	560.78	FV

\*This program displays all results rounded to two decimal places.

Example: How much need be invested at an annual rate of 6.5% compounded semi-annually in order to be worth \$100,000 in 15 years?

Enter	Press	Display	Remarks
100000	<b>B</b>	100000.00	FV
6.5	$\div$	6.5	
2	<b>= C</b>	3.25	i
30	<b>D</b>	30.00	n
	<b>2nd A</b>	38308.77	PV



Example: How much will that same investment from the last example be worth in 10 years?

Enter	Press	Display	Remarks
20	<b>D</b>	20.00	n
	<b>2nd</b> <b>B</b>	72627.22	FV

Example: What is the annual interest rate of a \$500 investment that has a \$575 value after two years with the interest compounded quarterly?

Enter	Press	Display	Remarks
500	<b>A</b>	500.00	PV
575	<b>B</b>	575.00	FV
8	<b>D</b>	8.00	n
	<b>2nd</b> <b>C</b> <b>X</b>	1.76	i (Per quarter)
4	<b>=</b>	7.05	Annual rate

To solve these examples, it was not necessary for you to perform all the detailed keystrokes necessary to set up and execute the problems in the calculate mode – you don't even have to know the equations involved. Now we will give you a quick introduction to programming the SR-52 after which you should be fairly well acquainted with the calculator and ready to start learning the details.

## CREATING YOUR OWN PROGRAMS

The learn mode allows you to create your own programs and enter them directly into the program memory of the SR-52. The basic operating instructions to accomplish this are simple: Enter the learn mode by pressing **[LRN]** and define the program by keying in the proper sequence of instructions. Then, when your program is complete in program memory, transfer out of the learn mode by again pressing **[LRN]**. Your program now resides in program memory and if so desired may be recorded on a magnetic card. Of course, this is not the whole story. There is a certain amount of activity which is necessary prior to your keying in the program. This is the phase where you determine your objectives, assemble the equations you will need, and finally write down the actual program steps.

Example: In a compound-interest problem, what is the future value given the other three variables as input quantities? There is no single solution to this problem, but we will discuss two different ways it might be done.

### METHOD 1 Datamath Calculator Museum

The easiest way to write a program to solve for FV is just to copy the keystrokes which would be used in the calculate mode, except that where you would enter data, a halt instruction **[HLT]** would be used to allow entry of data from the keyboard at that point. Execution would then be resumed by pressing **[RUN]**. From the equation for future value,  $FV = PV \times (1 + (i/100))^n$ , the key sequence in the calculate mode would be as follows:

#### Enter

Present Value (PV)

1

Interest Rate (i)

100

Number of Periods (n)

#### Press

**[X]** **[ ( ]**

**[+]** **[ ( ]**

**[÷]**

**[ ) ]** **[ ) ]** **[y<sup>x</sup>]**

**[=]**

Verify that the above key sequence will correctly solve for the future value (FV) with  $PV = \$500$ ,  $i = 5.75\%$  annually, and  $n = 24$  months. The result is 560.787147. Be careful to enter the value .4792 for  $i$ , rather than attempting to interject the calculation  $5.75 \div 12 =$  in the middle of the above keystroke sequence. This is a shortcoming of the type of program we are currently designing: During halts the calculator should only be used to enter the appropriate data in this type program, unlike the more usable program, BA1-08 where you were able to interject " $5.75 \div 12 =$ " before pressing **C**. A program which is a literal automation of the foregoing calculate-mode keystrokes would be the following:

Key Sequence	Remarks
<b>2nd</b> <b>LBL</b>	(Giving the SR-52 a place to go when you press <b>B</b> to solve the problem)
<b>B</b>	
<b>X</b>	
<b>(</b>	
<b>1</b>	
<b>+</b>	
<b>(</b>	
<b>HLT</b>	(to allow entry of $i$ )
<b>÷</b>	
<b>1</b>	
<b>0</b>	
<b>0</b>	
<b>)</b>	
<b>)</b>	
<b><math>y^x</math></b>	
<b>HLT</b>	(to allow entry of $n$ )
<b>=</b>	
<b>HLT</b>	(to display the answer)

© 2010 Joerg Woerner  
Datamath Calculator Museum

Note how we have labeled this program at the beginning so the SR-52 will find its destination when **[B]** is pressed.

Now you may load this program into the SR-52 program memory. First, turn your calculator off then on again to clear the program already in the program memory. Now press the **[LRN]** key and you should see displayed 000 00; your calculator is now in the learn mode. Carefully key in the steps listed above. If you make a mistake, turn the calculator off and on and start over again. (Later you will learn how to correct mistakes without starting over.) At the conclusion of keying in the program steps you should see 018 00 displayed, designating that the next instruction would go into location 018, currently empty. To run the program, go from the learn mode to the calculate mode by pressing **[LRN]**. The instructions for using this program are as follows:

Step	Enter	Press	Remarks
1	PV	<b>[B]</b>	Begins program execution
2	i	<b>[RUN]</b>	Resumes execution
3	n	<b>[RUN]</b>	Concludes execution

Try the case already worked out, with  $PV = 500$ ,  $i = .4792$ , and  $n = 24$ , and the answer is 560.787147.

Obviously, this program is not as easy to use as BA1-08. For one thing, the data had to be entered in a definite order and the calculator could not be used for intermediate calculations during program halts. The next method will demonstrate improved programming techniques to solve the interest problem.

## METHOD 2

In this method we will design the program much more along the lines of BA1-08. We use the labels **A** , **C** , and **D** to store the present value, interest rate, and number of compounding periods into selected memory registers. After the data are entered, the calculation of Future Value (FV) at label **B** will recall those data from memory as needed. We have to decide which memory registers to use for PV, i, and n. For simplicity, select register 01 for PV, register 02 for i, and register 03 for n; although this is as arbitrary as our labeling scheme. The segments of code necessary to store the data in memory are the following:

### Key Sequence

### Remarks

2nd **LBL**  
A  
STO  
0  
1  
HLT

Stores PV into register 01.

2nd **LBL**  
C  
STO  
0  
2  
HLT

Stores i into register 02.

2nd **LBL**  
D  
STO  
0  
3  
HLT

Stores n into register 03.

We still need to define the program segment to calculate FV from the quantities in the memory registers. This is achieved like the program of Method 1, except for the memory references:

### Key Sequence

<b>2nd</b> <b>LBL</b>	$\div$
<b>B</b>	<b>1</b>
<b>RCL</b>	<b>0</b>
<b>0</b>	<b>0</b>
<b>1</b>	<b>)</b>
<b>X</b>	<b>)</b>
<b>(</b>	<b>y<sup>x</sup></b>
<b>1</b>	<b>RCL</b>
<b>+</b>	<b>0</b>
<b>(</b>	<b>3</b>
<b>RCL</b>	<b>=</b>
<b>0</b>	<b>HLT</b>
<b>2</b>	

To use the Method 2 program, first clear program memory by turning the calculator off then on again. Go to the learn mode by pressing **LRN**. Now key in all instructions beginning with **2nd** **LBL** **A** and concluding with **HLT** in the program segment labeled **B**. If you performed these steps properly, you will see the displayed number 043 00, indicating that the next program location is 043.

Now press **LRN** to return to the calculate mode, and test your program using the following procedure:

Step	Procedure	Press	Display
1	Enter PV, i, and n in any sequence:		
	Present value PV	<b>A</b>	PV
	interest rate (%) i	<b>C</b>	i
	number of periods n	<b>D</b>	n
2	Calculate future value	<b>B</b>	FV

Note that you can now calculate " $5.75 \div 12 =$ " prior to pressing **C**.

This introduction to your SR-52 should allow you to begin solving a few problems. You will find that optimizing the use of your programmable calculator is a rewarding experience. The following sections are designed to help you develop your problem solving ability.

© 2010 Joerg Woerner

Datamath Calculator Museum

## II. ENTERING AND DISPLAYING NUMBERS

Now we will return to basics and begin a more thorough description of SR-52 operation. Remember that generally any description which pertains to calculate-mode keystrokes also applies when those keystrokes are made in the learn mode and processed in the run mode.

### ENTERING NUMBERS

Any number up to ten digits may be keyed in by using the keys  $\boxed{0}$  through  $\boxed{9}$  and the decimal key  $\boxed{\cdot}$ . The procedure for entering a positive number is simply to press the keys in sequence exactly as the number appears.

Example: Enter 671.8409236

Enter	Press	Display
	$\boxed{\text{CLR}}$	0
671.8409236		671.8409236

The decimal point entry is not needed to enter an integer. The calculator automatically supplies the decimal when any function key is pressed.

Example: Enter 1019

Enter	Press	Display
	$\boxed{\text{CLR}}$	0
1019	$\boxed{+}$	1019.

Decimal point entry is required for numbers less than one, a leading zero is automatically displayed for clarity.

Example: Enter 0.04077

Enter	Press	Display
	$\boxed{\text{CLR}}$	0
.04077		0.04077



Negative numbers are entered just like positive numbers except that the change-sign key  $\boxed{+/-}$  is pressed as the final step of number entry.

Example: Enter  $-1019.04077$

Enter	Press	Display
	$\boxed{\text{CLR}}$	0
1019.04077	$\boxed{+/-}$	-1019.04077

The change-sign key may be used to change the sign of the displayed quantity at any time.

As you enter a number in the calculate mode, that number will be shown in the display, including the decimal point and sign. If you make a mistake, simply press the  $\boxed{\text{CE}}$  key and begin again. If more than ten digits are attempted in number entry, all after the tenth will be ignored.

## ENTERING $\pi$

As a convenience the SR-52 provides a means of entering  $\pi$  by pressing a single key. Use of this key to enter  $\pi$  turns out to be more than just a convenience. A 12-digit representation of  $\pi$  results from this operation. (You are restricted to only 10 digits when the manual number entry method is used.) The display always rounds to ten or fewer digits, so the extra two digits are not visible, but they are carried in all subsequent calculations.

Example: Enter pi ( $\pi$ )

Enter	Press	Display
	$\boxed{2\text{nd}}$ $\boxed{\pi}$	3.141592654

The value of pi actually used by the calculator is 3.14159265359 even though the display only shows the value rounded to ten significant digits.

The twelve digit internal representation of  $\pi$  using the **2nd**  **$\pi$**  method of entry exceeds the exact value of  $\pi$  by less than  $2.07 \times 10^{-9}$ , or less than  $6.59 \times 10^{-8}$  percent. This represents an error equivalent to about one inch in computing the equatorial circumference of a sphere the size of the earth from its radius, a precision adequate for most purposes.

## SCIENTIFIC NOTATION

To enter very large or very small numbers you may use scientific notation where the number is expressed as the product of a number and a power of ten (either positive or negative). This first factor is usually referred to as the mantissa, and the second factor, the power-of-ten, is called the exponent. The full procedure, therefore, is to key in the mantissa (including its sign) then press the enter-exponent **EE** key, and finally key in the power of ten.

Example: Enter  $6.025 \times 10^{23}$

Enter	Press	Display
	<b>CLR</b>	0
6.025	<b>EE</b>	6.025 00
23		6.025 23

Regardless of how the number is entered, the calculator will normalize the number, displaying a single-digit to the left of the decimal when any function key is pressed.

Example: Enter  $6025 \times 10^{20}$

Enter	Press	Display
	<b>CLR</b>	0
6025	<b>EE</b>	6025 00
20		6025 20
	<b>+</b>	6.025 23

The change-sign key can again be used to assign a negative sign to the mantissa and to the power-of-ten exponent.

Example: Enter  $-4.818 \times 10^{-10}$

Enter	Press	Display
	<b>CLR</b>	0
4.818	<b>+/-</b> <b>EE</b>	-4.818 00
10	<b>+/-</b>	-4.818-10

If you wish to change the sign of the mantissa after the exponent has been entered, just press **.** **+/-**. If the numerical value of the mantissa needs to be changed, reenter the complete number after pressing **CE** or **CLR**.

Example: Change the number displayed in the last example to  $4.818 \times 10^{-10}$

Enter	Press	Display
	<b>.</b> <b>+/-</b>	4.818-10

© 2010 by Voernet  
Datamath Calculator Museum

The exponent size is limited to numbers no longer than two digits. This provides a range of powers-of-ten from  $10^{-99}$  to  $10^{99}$ . Note, however, that the SR-52 will experience **underflow** whenever calculations result in numbers less than  $1. \times 10^{-99}$  and **overflow** whenever encountering calculations greater than  $9.999999999 \times 10^{99}$  in magnitude. A flashing display results from these situations.

## ADVANCED EFFECTS AND USES OF THE **EE** KEY

Pressing the **EE** key as in using scientific notation causes several things to take place:

1. The display mode is set to scientific notation.

- The two-digit registers for holding the exponent are initialized to zero unless the number in the display register already has an assigned exponent, which will not be initialized.
- Number key entries immediately following the **EE** along with the pre-existing value of the exponent (if any) are interpreted so that the last two digits entered form the exponent. The earlier-entered digits are discarded.
- The rounded value of the mantissa actually shown in the display is loaded into the display register for subsequent calculations. This is sometimes a useful feature.

The first effect is obvious. The second and third effects are illustrated in the following example.

Example: Solve the problem  $(2 \times 10^3) \times (2 \times 10^6)$  and change the exponent of the result to 12.

Enter	Press	Display
	<b>CLR</b>	0.00
2	<b>EE</b>	2.00
3	<b>X</b>	2.03
2	<b>EE</b>	2.00
6	<b>=</b>	4.09
	<b>EE</b>	4.09
1		4.91
2		4.12

Observe that the first two times **EE** was pressed following a number entry, the exponent digits were initialized to 00. When **EE** is used following a result, the exponent digits are unchanged and only the last two numbers entered are used as exponent digits.

We will use the pi key to illustrate the fourth effect of the **EE** key.

Example:  $\pi - \pi \neq 0$

Enter	Press	Display
	<b>CLR</b>	0
	<b>2nd</b> <b><math>\pi</math></b> <b>-</b>	3.141592654
	<b>2nd</b> <b><math>\pi</math></b> <b>EE</b>	3.141592654 00
	<b>=</b>	-4.1-10

As previously indicated, when **2nd**  **$\pi$**  is used, the calculator internally uses 3.14159265359 though it only displays 3.141592654. Pressing **2nd**  **$\pi$**  **EE**, however, causes the calculator to use only the displayed number — discarding all digits not displayed. The calculation which took place was actually  $3.14159265359 - 3.141592654$  which equals  $-0.00000000041$  or  $-4.1 \times 10^{-10}$ . Note that this last effect of the **EE** key affects only the number displayed at the time **EE** is pressed.

## CLEARING INCORRECT NUMBER ENTRIES

To clear an incorrect number entry from the keyboard, simply press the clear-entry key **CE** and enter the proper number. This key clears only entries from the keyboard, not results of calculations or  $\pi$ . There are other types of clearing operations for the SR-52 and other keys for accomplishing them. These will be discussed where appropriate.









## CLEARING A CALCULATION

One of these additional clearing operations is the clear key **CLR**. Pressing this key clears the display and clears all calculations in progress to insure your new problem is not affected by operations not completed from a previous problem. The clear key does not affect the contents of the memory registers or program memory.

## DISPLAY CONTROL

The SR-52 display gives you a controllable-format representation of the number in the display register. The display register like all other registers in the SR-52 is 12 digits long. For display this number is rounded to 10 or fewer digits.

When your calculator is first turned on it is in the initial display mode. In this mode all numbers are displayed without the power-of-ten present in scientific notation.

Enter	Press	Display
8		8.
2		4.
1		1.
3		.3333333333
100		100.
3		33.33333333
1		1.
8		0.125

The number of digits displayed following the decimal point is as many as required within the maximum limit of ten total digits to represent the number.

Whenever you enter a number in scientific notation or when a number resulting from a calculation is less than .0000000001 or greater than 9999999999 in magnitude, the display automatically converts to scientific notation. The exponent is shown by two digits set off to the right of the others on the display. If the exponent is negative, a negative sign appears just to the left of these two digits and a ten digit mantissa is still available. The mantissa shown in the display is always in the range  $1 \leq |M| < 10$ . A mantissa greater than this can be keyed in, but as soon as any operation, function, or storage key is pressed, the mantissa is converted to the above range.

Displaying the mantissa with as many digits as required up to the maximum of ten is known as the initial mantissa format.

Example:  $(13.5 \times 10^7) \times 9 = ?$

Enter	Press	Display
13.5	$\boxed{EE}$	13.5 00
7	$\boxed{X}$	13.5 07
	$\boxed{X}$	1.35 08
9	$\boxed{=}$	1.215 09

The display has switched from the initial display mode to scientific notation, but the initial mantissa format has been preserved.

If you don't want to see all the digits which may be present in the initial mantissa format, you may cause all displayed results to be rounded to a fixed number of places following the decimal point by pressing  $\boxed{2nd}$   $\boxed{fix}$  and then enter the desired number of digits 0 through 8. The contents of the display register will be shown with the mantissa rounded to the desired number of digits, *but all calculations will use the full unrounded value.*

Example:  $2 \div 3 = ?$  Round to two decimal places.

Enter	Press	Display
2	$\div$	2.
3	$=$	.666666667
	<b>2nd</b> <b>fix</b> 2	0.67

The result is shown rounded to hundredths. Similarly, turn the calculator off then on to restore the initial display mode and work the following example in scientific notation.

Example:  $(2 \times 10^9) \div 3 = ?$  Round to four decimal places.

Enter	Press	Display
2	<b>EE</b>	2. 00
9	$\div$	2. 09
3	$=$	6.666666667 08
	<b>2nd</b> <b>fix</b> 4	6.6667 08

To return the display to the initial mantissa format without turning the calculator off, two methods are available. Pressing **2nd** **fix** 9 or **INV** **2nd** **fix** will restore the display to the initial mantissa format without affecting scientific notation.

To convert the display from scientific notation simply use the key sequence **INV** **EE**. This does not affect the number of digits carried unless the number cannot be represented in the prevailing mantissa format with ten digits. When this occurs the initial mantissa format is used and when necessary the display remains in scientific notation.



Example:  $1 \div (2 \times 10^3) = ?$

Enter	Press	Display
1	$\div$	1.
2	EE	2. 00
3	=	5.-04
	2nd fix 2	5.00-04
	INV EE	0.00
	2nd fix 3	0.001
	2nd fix 4	0.0005
	2nd fix 5	0.00050

To convert to scientific notation there are two approaches. The first is to press  $\times$  1 EE =, which multiplies the number in the display register by  $1 \times 10^0$  and converts the display to scientific notation. The second method is to press EE =. You should be careful in using the second method. It has the consequence of loading the display register with the *rounded* quantity being displayed.

Caution! In no case should you use either of these two display commands, which use =, in the middle of a computation with pending operations. The reason is that the = completes all pending operations. Pending operations and the effect of = is discussed later in this manual. To avoid this hazard, use these conversion methods only after computations are complete, or else properly interject the multiply by  $1. \times 10^0$  without the = keystroke.

## ERROR CONDITIONS

Various error conditions during computation (either in the calculate or run modes) result in a flashing display. The quantity flashed in the display is a clue to the type of error involved. We will deal with each error condition in its appropriate place, listing only the two most common errors here. Refer to Appendix B for a complete summary of error conditions.

**Overflow** – Signifies a result whose magnitude is larger than the maximum which can be handled by the SR-52 ( $9.999999999 \times 10^{99}$ ) and results in a flashing 9.999999999 99 display. This condition also is obtained from attempted division by zero.

**Underflow** – Indicates a result which is different from zero but whose magnitude is too small to be handled by the SR-52 ( $1. \times 10^{-99}$ ). This condition results in a flashing 1.-99 display.

In the case of a flashing display resulting from any error, pressing **CE** will stop the flashing without affecting the displayed number.

### III. ARITHMETIC CALCULATIONS

#### BASIC OPERATIONS

To perform simple addition, subtraction, multiplication or division, the procedure is to key in the problem just as it is written.

Enter first number

Press  $\boxed{+}$ ,  $\boxed{-}$ ,  $\boxed{\times}$ , or  $\boxed{\div}$

Enter second number

Press  $\boxed{=}$ .

Pressing  $\boxed{\text{CLR}}$  at the beginning of this sequence clears any calculations in progress.

Example:  $(1.6 \times 10^{-19}) \times (6.025 \times 10^{23}) = ?$

Enter	Press	Display
1.6	$\boxed{\text{EE}}$	1.6 00
19	$\boxed{+/-}$ $\boxed{\times}$	1.6- 19
6.025	$\boxed{\text{EE}}$	6.025 00
23	$\boxed{=}$	9.64 04

#### CHAINED OPERATIONS

After a result is obtained in one calculation it may be directly used as the first number in a second calculation. There is no need to reenter the number from the keyboard.

Example:  $1.84 + 0.39 = ?$  then  $(1.84 + 0.39)/365 = ?$

Enter	Press	Display	Remarks
1.84	$+$	1.84	
.39	$=$	2.23	$1.84 + 0.39$
	$\div$	2.23	
365	$=$	0.006109589	$2.23 \div 365$

In this way an indefinite number of operations may be chained together, entering each operand only once.

## A SPECIAL TYPE OF OPERATION CHAIN

In the foregoing discussion chaining or using the result of one calculation as the first number of the next calculation, involved pressing  $=$  for each calculation, thereby obtaining the whole chain of intermediate results as well as the final answer. Of course you would prefer not to have to press  $=$  except at the end. There are two types of chains which allow you to do this. These are:

- 1) Chains containing only  $+$  and  $-$  operations.
- 2) Chains containing only  $\times$  and  $\div$  operations.

In order to evaluate expressions of this type simply key in the numbers and operation keys the way the problem is written and finally press  $=$  to get the answer.

Example:  $64 + 139 - 22 + 11 - 68 = ?$

Enter	Press	Display	Remarks
64	$+$	64.	
139	$-$	203.	$64 + 139$
22	$+$	181.	$64 + 139 - 22$
11	$-$	192.	$64 + 139 - 22 + 11$
68	$=$	124.	

Example:  $2.3 \times 340 \times 24 \div 16 \times 10 = ?$

Enter	Press	Display	Remarks
2.3	$\times$	2.3	
340	$\times$	782.	$2.3 \times 340$
24	$\div$	18768.	$782 \times 24$
16	$\times$	1173.	$18768 \div 16$
10	$=$	11730.	

## PARENTHESES

To introduce this subject you should try the following experiment: Press  $\text{CLR}$   $($   $5$   $\times$   $7$   $)$ , and you will see the displayed value 35. The SR-52 has evaluated  $5 \times 7$  and replaced it with 35, even though the  $=$  was not pressed. This behavior is a consequence of the following operating characteristic designed into the SR-52.

Whenever an expression is set off by parentheses, that is,  $([\text{expression}])$ , the SR-52 will evaluate that expression then use its value in any larger expressions of which it is a part. This evaluation will be properly carried out even if the expression set off by parentheses itself contains other smaller expressions which are set off by parentheses. It may also contain functions as well as the four arithmetic operations. As the keystroke sequence containing parentheses is processed, the SR-52 automatically stores away into internal processing registers those operands which cannot yet be combined with other operands.

Note that starting a problem with  $($  does not usually require using  $\text{CLR}$  since the number entry replaces the number displayed when  $($  was pressed. Example problems in this manual beginning with  $\text{CLR}$   $($  include the clear operation for the convenience of showing display contents for each step.

Example:  $(5 + (8/(9 - (2/(3 + 1)))))) = ?$

**Keystrokes**

**SR-52 Action**

- |     |  |
|-----|--|
| (   | Set up to evaluate expression  |
| 5 + | Store 5 internally, marked for pending addition  |
| (   | Set up for evaluation of 2nd-level parentheses   |
| 8 ÷ | Store 8 internally, marked for pending division  |
| (   | Set up third-level parentheses   |
| 9 - | Store 9 internally, marked for pending subtraction   |
| (   | Set up next level of parentheses   |
| 2 ÷ | Store 2 internally, marked for pending division  |
| (   | Set up next level of parentheses   |
| 3 + | Store 3 internally, marked for pending addition  |
| 1 ) | Recognize $3 + 1$ can now be performed<br>Replace $3 + 1$ with 4   |
| )   | Recognize $2 \div 4$ can now be performed<br>Replace $2 \div 4$ with .5  |
| )   | Recognize $9 - .5$ can now be performed<br>Replace $9 - .5$ with 8.5   |
| )   | Recognize $8 \div 8.5$ can now be performed<br>Replace $8 \div 8.5$ with .941176 etc.  |
| )   | Recognize $5 + .941176$ can now be performed<br>Replace $5 + .941176$ with 5.941176471<br>Recognize expression now evaluated may now be combined in any larger expression of which it is a part. |

By the time  $\square$  was encountered in the previous example, the SR-52 had stored five operands, each associated with an operation pending, plus the final operand in the sequence for a total of six operands stored in the internal processing registers. Closing the final parentheses in this particular example caused the whole expression to cascade down to a single evaluated number, the SR-52 recognizing at exactly what point each pending operation could be completed. You should key in the expression just discussed exactly as written, and observe the display at various points in the sequence. The final answer is 5.941176471. The rule which results from all this is extremely simple from the user's point of view. To evaluate an expression containing parentheses, key in the expression just as it is written. Not only does this design feature allow you to use parentheses on the SR-52 just as you do in your analytical work, it also saves your addressable memory registers for purposes other than for storing operands which have operations pending. In making optimal use of the internal processing registers, the natural keystroke sequence with parentheses is not only easy and natural, but is efficient in memory usage as well. In addition when you program the SR-52 using this same capability to deal with parenthetical expressions, you are also obtaining the most efficient operation from an execution time point of view as well as program code whose intent remains clear long after it is written.

There are limits on how many pending operations and operands can be entered in the internal processing registers. This limit, though large enough that you will probably never be aware that it exists, can accommodate as many as eleven operands with ten operations pending. If you attempt to open more than the maximum number of pending operations, the display will flash the current value in the display register.

## THE USE OF $\boxed{=}$ IN COMPLICATED EXPRESSIONS

There is one other feature related to pending operations which you will find particularly important. The effect of pressing  $\boxed{=}$  in any calculation (or encountering  $\boxed{=}$  in any SR-52 program) is to complete all pending operations. It does not matter that the right parentheses associated with existing left parentheses have not all appeared, for the  $\boxed{=}$  has the effect of immediately supplying as many right parentheses as necessary to complete the expression. (Try working the last example, substituting an  $\boxed{=}$  for the five right parentheses.)

Example:  $-10.7 + ((11.5 - 8)/(11.5 + 8)) = ?$

Enter	Press	Display
10.7	$\boxed{+/-}$ $\boxed{+}$ $\boxed{(}$ $\boxed{(}$	-10.7
11.5	$\boxed{-}$	11.5
8	$\boxed{)}$ $\boxed{\div}$ $\boxed{(}$	3.5
11.5	$\boxed{+}$	11.5
8	$\boxed{=}$	-10.52051282

By now you perhaps see that the  $\boxed{=}$  is a convenience. It is possible to use parentheses to perform all computations on the SR-52 without ever using  $\boxed{=}$ . This is an important fact, which will be explored later, in the discussion of programming subroutines.



## IV. SPECIAL FUNCTIONS

The simplest operations of all to describe and understand are probably the single-variable functions on the SR-52. We will therefore first describe these functions, then go on to the functions of two variables, and finally discuss angular unit conversions (which are really functions of a single variable).

### FUNCTIONS OF A SINGLE VARIABLE

At any point in a calculation you can replace the value in the display register with the implied operation represented by any of the following keys.

<b>2nd</b> <b>1/x</b>	<b>2nd</b> <b>x!</b>	<b>sin</b>
<b>2nd</b> <b>√x</b>	<b>2nd</b> <b>log</b>	<b>cos</b>
<b>2nd</b> <b>x<sup>2</sup></b>	<b>lnx</b>	<b>tan</b>

© 2010 Joerg Woerner

Example:  $\sqrt{10} = ?$

Enter	Press	Display
10	<b>2nd</b> <b>√x</b>	3.16227766

Example:  $\ln(8/3) = ?$

Enter	Press	Display
	<b>CLR</b> <b>(</b>	0
8	<b>÷</b>	8.
3	<b>)</b>	2.666666667
	<b>lnx</b>	0.980829253

Example:  $44! = ?$

Enter	Press	Display
44	<b>2nd</b> <b>x!</b>	2.658271575 54

In addition to the functions already listed the following inverse functions are available by use of the **INV** prefix:

**Function**

$10^x = \text{INV LOG } X$

$e^x = \text{INV LN } X$

$\text{arc sin } x \text{ or } \text{sin}^{-1}x = \text{INV SIN } X$

$\text{arc sin } x \text{ or } \text{cos}^{-1}x = \text{INV COS } X$

$\text{arc tan } x \text{ or } \text{tan}^{-1}x = \text{INV TAN } X$

**Press**

**INV** **2nd** **log**

**INV** **lnx**

**INV** **sin**

**INV** **cos**

**INV** **tan**

Example:  $e^{-3.7} = ?$

<b>Enter</b>	<b>Press</b>	<b>Display</b>
3.7	<b>+/-</b>	-3.7
	<b>INV</b> <b>lnx</b>	.0247235265

When a single-variable function is activated, its effect is immediate. The display register contents are replaced with the function value without any effect upon pending operations. (The effect is as though one were to have keyed in the special function value at that point.)

Example:  $\sqrt{(6 + 19)} - 2 = ?$

<b>Enter</b>	<b>Press</b>	<b>Display</b>
	<b>CLR</b> <b>(</b>	0
6	<b>+</b>	6.
19	<b>)</b>	25.
	<b>2nd</b> <b><math>\sqrt{x}</math></b> <b>-</b>	5.
2	<b>=</b>	3.

Now to show that pending operations are not affected by a single-variable function, perform the following key sequence:

Example:  $6 + 19 - 2 = ?$

Enter	Press	Display	Remarks
	<b>CLR</b>	0	A mere precaution
6	<b>+</b>	6.	6 is stored with + pending.
19	<b>-</b>	25.	$6 + 19 = 25$ is completed and then stored with - pending.
	<b>2nd</b> <b><math>\sqrt{x}</math></b>	5.	$\sqrt{25}$ replaces 25 in display register. Now we will verify that the pending operation will not be affected.
2		2	2 has written over the 5 in the display register.
	<b>=</b>	23.	The final pending operation (subtraction) is completed giving $25 - 2 = 23$ .

Some of the single-variable functions have restrictions on the values of their arguments, in addition to those necessary to avoid overflow or underflow. These restrictions and the error indications when they are violated are summarized in the following table:

Function	Permissible x Values	Error Indication
----------	----------------------	------------------

$\sqrt{x}$	$x \geq 0$	Flashing $\sqrt{ x }$
$x!$	$69 \geq \text{integers} \geq 0$	Flashing $ \text{INT}(x) !$ ( $\text{INT}(x) =$ integer part of $x$ )
$\log$	$x > 0$	Flashing $\log( x )$
$\ln x$	$x > 0$	Flashing $\ln( x )$
$\text{arc sin } x$	$ x  \leq 1$	Flashing $x$
$\text{arc cos } x$	$ x  \leq 1$	Flashing $x$

Although an error condition does not result, attempting to find the sin, cos, or tan of an angle  $1.001 \times 10^{14}$  degrees or larger results in that angle's being interpreted as zero. As long as the display is not using scientific notation, all displayed digits are accurate for the range  $-36000$  to  $36000$  degrees. In general, the accuracy decreases one digit for each decade outside this range.

**Angular Mode Selection** – The trigonometric functions including the inverse functions and coordinate conversions involve knowledge about units – are angles expressed in degrees or in radians? You will notice the slide switch just under the left side of the display. By sliding this switch to the right (toward the **D**) you place the calculator in the degree mode. By sliding it to the left (toward the **R**) you place it in the radian mode. The placement of this switch has absolutely no effect except when a trigonometric function or coordinate conversion is being performed. You may therefore change this switch setting at any intermediate point in a calculation. The position of this switch is indicated for each problem in this manual which depends on the switch position – **Angle:Deg** means set the switch to **D** and **Angle:Rad** means to set the switch to **R**.

Example:  $\sin 30^\circ = ?$

Angle: Deg

Enter	Press	Display
30	$\boxed{\sin}$	0.5

Example:  $\arcsin (.5) = ?$  in radians

Angle: Rad

Enter	Press	Display
.5	$\boxed{\text{INV}} \boxed{\sin}$	.5235987756

Selecting the angular mode is an easy step to perform – and to forget! Forgetting this step is responsible for a large proportion of errors in operating any calculating machine that offers this option.

## FUNCTIONS OF TWO VARIABLES

Your calculator provides two functions of two variables. The first function is powers, accessed by the  $\boxed{y^x}$  key. The second is roots, accessed through the  $\boxed{x\sqrt{y}}$  key. The rules for these two functions are essentially identical:

1. To raise  $y$  to the  $x$ th power [1] enter  $y$  then [2] press  $\boxed{y^x}$  and [3] enter the quantity  $x$ ; finally, [4] press  $\boxed{=}$  to compute the result.
2. To take the  $x$ th root of  $y$ , [1] enter  $y$  then [2] press  $\boxed{x\sqrt{y}}$  and [3] enter  $x$ ; finally, [4] press  $\boxed{=}$  to compute the result.

Example:  $2.86^{-.42} = ?$

Enter	Press	Display
2.86	$\boxed{y^x}$	2.86
.42	$\boxed{+/-}$	-0.42
	$\boxed{=}$	.6431707214

Example:  $3.12\sqrt{1460} = ?$

Enter	Press	Display
1460	$\sqrt{x}$	1460.
3.12	$=$	10.33274375

Either variable x or y or both may themselves be the results of other computations which may involve parentheses and the proper function will be computed. Furthermore, parenthetical expressions may contain these functions.

Example:  $((11.8 - 2.49)/11)^{(19/16)} = ?$

Enter	Press	Display	Remarks
	CLR ( (	0	
11.8	-	11.8	
2.49	) ÷	9.31	
11	) y <sup>x</sup> (	.8463636364	Value of y
19	÷	19.	
16	)	1.1875	Value of x
	=	.8203023062	Value of y <sup>x</sup>

Example:  $(3 \times (4^{(2^{-(\sqrt[3]{7})})})) = ?$

Enter	Press	Display	Remarks
	CLR (	0	
3	× (	3.	
4	y <sup>x</sup> (	4.	
2	y <sup>x</sup> (	2.	
7	$\sqrt[3]{x}$	7.	
4	)	1.626576562	Value of $\sqrt[3]{7}$
	+/-	-1.626576562	Value of $-(\sqrt[3]{7})$
	)	.3238557891	Value of $2^{-(\sqrt[3]{7})}$
	)	1.566681134	Value of $4^{.323...}$
	)	4.700043401	Value of $3 \times 4^{.323...}$

You will note that no  $\boxed{=}$  was used in the last example. The enclosure of the whole expression with parentheses is sufficient to cause it to be evaluated and the fact that  $y^x$  and  $\sqrt[y]{x}$  functions occur does not in any way alter the basic parentheses disciplines already described.

There is a restriction on these functions – the variable  $y$  must be non-negative. If  $y$  is negative the display will flash the value of  $|y|^x$  or  $\sqrt[y]{|x|}$ , for the respective exponentiation and root-extraction functions. The quantities  $0^0$  and  $\sqrt[0]{0}$  are mathematically indeterminate forms. Attempting  $\sqrt[0]{0}$  produces a flashing 1. display. No error condition is signalled for  $0^0$ , which is set equal to 1.

## ANGULAR UNIT CONVERSIONS

Four additional single-variable functions are provided which were not discussed in the earlier section on such functions. They are not standard mathematical functions, but rather are angular-unit conversions, as shown below.

Function	Press	Effect
Degrees-to-radians	$\boxed{2nd}$ $\boxed{D/R}$	Multiplies displayed value by $\pi/180$ .
Radians-to-degrees	$\boxed{INV}$ $\boxed{2nd}$ $\boxed{D/R}$	Multiplies displayed value by $180/\pi$ .
Degrees-minutes-seconds to decimal degrees	$\boxed{2nd}$ $\boxed{D.MS}$	Converts a number entered in the degree-minute-second format to decimal degrees.
Decimal degrees to degrees-minutes-seconds	$\boxed{INV}$ $\boxed{2nd}$ $\boxed{D.MS}$	Converts a number from decimal degrees to the degree-minute-second format.

Example: Convert 145 degrees to radians

Enter	Press	Display
145	<b>2nd</b> <b>D/R</b>	2.530727415

Example: Convert 2.530727415 radians to degrees

Enter	Press	Display
2.530727415	<b>INV</b> <b>2nd</b> <b>D/R</b>	145.

The next two functions require explanation of the degree-minute-second format. These functions allow you to enter or display an angle expressed in degrees, minutes, and seconds. The actual format is given by DDD.MMSSsss where:

DDD denotes degrees,  
• separates degrees and minutes,

MM denotes minutes,

SS denotes seconds,

and sss denotes the decimal fraction of seconds.

The number of degree digits preceding the decimal and the number of s-digits denoting the fractional part of seconds are not required to be three as shown, but are limited only by the ten total digits available. The values for MM and SS may not be larger than 97.

Example: Convert  $127^{\circ} 09' 31.2''$  to decimal degrees

Enter	Press	Display
127.09312	<b>2nd</b> <b>D.MS</b>	127.1586667



Example: Convert 38.25833333 degrees to degrees-minutes-seconds

Enter	Press	Display
38.25833333	<b>INV</b> <b>2nd</b> <b>D.MS</b>	38.153
	<b>2nd</b> <b>fix</b> 6	38.153000

The displayed answer is interpreted as  $38^{\circ} 15' 30''$ .

Notice from the last example that although the **INV** **2nd** **D.MS** sequence provides the necessary numerical value, it does not automatically change the display format so that you can observe the DD.MMSSsss representation. You are responsible for controlling the display format. The two D.MS functions may also be used to convert between decimal hours and hours-minutes-seconds.

All of these angular unit conversion functions operate independently of the position of the angular mode switch. In other respects they behave similarly to the usual single-variable functions, affecting only the displayed quantity and not any other operands or pending operations. Unlike the usual functions, however, the D.MS functions operate on the number in the display, not upon the 12-digit number in the display register. So once again, remember: You must set the display format to use the D.MS functions.

## COORDINATE (POLAR/RECTANGULAR) CONVERSIONS

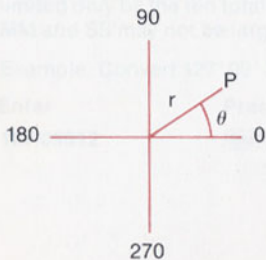
A special capability is provided on your calculator to allow you to easily convert between polar and rectangular coordinates. This coordinate conversion does involve the data memory, specifically register 00 ( $R_{00}$ ).

The action of the polar-to-rectangular conversion is summarized in the table below:

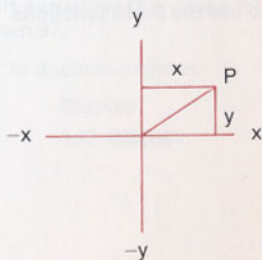
	Register Contents	
	Before	After
Display register	$\theta$	$y$
Memory register $R_{00}$	$r$	$x$

$\boxed{2nd} \boxed{P/R} \Rightarrow$

As shown in this table, with the radius  $r$  in register  $R_{00}$  and the angle  $\theta$  in the display register, the polar-to-rectangular conversion places the cartesian coordinate  $x$  in  $R_{00}$  and the cartesian coordinate  $y$  in the display register. The polar-to-rectangular conversion is activated by pressing  $\boxed{2nd} \boxed{P/R}$ . The geometrical relationships are shown below:



Polar



Rectangular

The inverse transformation, rectangular-to-polar, produces the result summarized in the next table:

		<i>Register Contents</i>	
	<u>Before</u>		<u>After</u>
Display register	y	[INV] [2nd] [P/R] ⇒	$\theta$
Memory register R <sub>00</sub>	x		r

This is seen to be just the reverse effect of the polar-to-rectangular transformation and is activated by the sequence [INV] [2nd] [P/R]. Input values x and y should be within  $10^{-50}$  for proper accuracy.

To use these transformations you must store and recall quantities using memory. The following examples show two possible key sequences.

Example: Convert to cartesian coordinates:  $r = 5, \theta = 30^\circ$

#### Angle:Deg

Enter	Press	Display	Remarks
5	[STO] 00	5.	
30	[2nd] [P/R]	2.5	Value of y
	[RCL] 00	4.330127019	Value of x

Example: Convert to polar coordinates (radians):  $x = 3, y = 4$

#### Angle:Rad

Enter	Press	Display	Remarks
3	[STO] 00	3.	
4	[INV] [2nd] [P/R]	0.927295218	Value of $\theta$
	[2nd] [EXC] 00	5.	Value of r

You will see that in the first example we used the recall instruction and just for variety we used the exchange instruction in the second example. Each has its merits: The recall is one keystroke shorter but we lose what was in the display register. The exchange method does not lose what was in the display register, swapping it with the contents of R<sub>00</sub>.

## V. ALGEBRAIC NOTATION: MORE ABOUT PENDING OPERATIONS

### THE ALGEBRAIC HIERARCHY

You have already read about how parentheses may be used on the SR-52 just as they are used in writing down algebraic expressions. You have also read about how these parentheses cause certain operations to be pending, or held up until other operations are completed. Again, this operating characteristic is like the usual algebraic practice. In normal algebraic usage, the sequence of operations appropriate for evaluating a given expression is affected (and defined) by the parentheses occurring in that expression. Specifically in algebra, with a set of nested parentheses, the expression must be evaluated from the innermost level of parentheses outward. Two or more parenthetical expressions at the same level of nesting may be evaluated in any sequence, once one has worked out to that level. But generally the sequence chosen in such cases is left-to-right. These rules of normal procedure are probably so natural to you that you hardly recognize them when they are formally stated. You don't really think of these rules consciously; you just know how to proceed through the tedium of parentheses evaluation.

Now this is important: The SR-52 executes the operations in a complicated expression in the exact sequence demanded by the foregoing rules. It does this even though the expression has been keyed in just as it was written. You need not look for the innermost level of parentheses; the SR-52 will find it (and higher levels) and retain the proper sequence.

There are certain additional rules, universally accepted, for identifying the proper sequencing of operations in a complicated algebraic expression. These rules pertain to the interpretation which should be made when the completely-defining complement of parentheses is not present. As you have probably anticipated, the SR-52 is absolutely faithful to these additional rules as well.

Suppose one has the expression  $5 \times (4 + 8) \times 3$ , the parentheses leave no doubt as to the meaning:

$5 \times 12 \times 3 = 180$ . But what would be the meaning of the expression if the parentheses were removed?

$$5 \times 4 + 8 \times 3 = ?$$

There would appear to be several possibilities of interpretation. In addition to the interpretation already given (leading to the answer 180) we could construct the following:

1.  $(5 \times 4) + (8 \times 3) = 20 + 24 = 44$

2.  $((5 \times 4) + 8) \times 3 = (20 + 8) \times 3 = 28 \times 3 = 84$

3.  $5 \times (4 + (8 \times 3)) = 5 \times (4 + 24) = 5 \times 28 = 140$

There appear to be four different interpretations and therefore four different answers. Without the parentheses the situation seems ambiguous, and the problem is aggravated further when the expressions involved are longer. Is there a convention which rules in favor of one of these four possible interpretations, or must we always resolve such ambiguities by explicitly exhibiting the parentheses? The answer to this question is that there is an accepted convention; and it is known as the **algebraic hierarchy**. The rules of algebraic hierarchy tell us that unless parentheses are present to indicate otherwise, multiplication or division should be performed prior to addition or subtraction. So the answer to the earlier questions is  $5 \times 4 + 8 \times 3 = 44$ . If you now try that keystroke sequence on your SR-52, you get 44 as the answer.

To discuss the complete rules of algebraic hierarchy consider a more complicated example:

$$4 \div 5 \times 7 + 3 \times \sin 60^{\circ} \cos 60^{\circ} = ?$$

Again, there are no parentheses to make clear the sequence intended. All the rules of algebraic hierarchy are required to resolve this case. These rules are as follows:

Except as affected by any parentheses,

1. Immediately perform function evaluations.
2. Then perform exponentiation and root extraction.
3. Then perform multiplication and division.
4. Finally perform addition and subtraction.
5. Perform operations on each level left-to-right.

According to these rules the foregoing expression should be interpreted to mean

$$(4 \div 5 \times 7) + (3 \times .8660254038^{\cdot 5})$$

The value of this expression is 8.391814577. Now see if you get that answer on the SR-52. You will if you use the natural keystrokes:

4  $\boxed{\div}$  5  $\boxed{\times}$  7  $\boxed{+}$  3  $\boxed{\times}$  6 0  $\boxed{\sin}$   $\boxed{y^x}$  6 0  $\boxed{\cos}$   $\boxed{=}$

(Did you remember to set degree mode?)

To summarize, the interpretation of an expression is affected by the presence of parentheses. In places where absence of the parentheses would leave doubt as to the proper operation sequence, that ambiguity is resolved by the rules of algebraic hierarchy. The SR-52 is designed to compute in the proper sequence as determined by these rules and by any parentheses present. If you choose, you may forget these rules and always use parentheses to completely determine the meaning of expressions. On the other hand, if you become familiar with the algebraic hierarchy it can save you from having to enter one or more sets of parentheses in most expressions.

## KEEPING TRACK OF DISPLAY REGISTER CONTENTS

You will soon learn about the use of memory to store and recall data. In order to know what quantity is being stored in response to a **[STO]** command from the keyboard (or from a program), you must be aware of what is contained in the display register. This is only possible if you become adequately familiar with the subject of pending operations as discussed up to now and practice going through calculations step by step, trying to anticipate what will be displayed with each keystroke. This will facilitate your using the learn mode much more than using the calculate mode, because in running a program you do not see what is in the display register as the SR-52 races through the program code. So practice in the calculate mode and you will greatly improve your effectiveness in the learn mode.

There is one characteristic of the display register which we have not formally explained: Whenever a number is entered into the display register, whether through direct digit-key entry, recall from memory, or as the result of a calculation, it writes over the prior contents of that register but does not affect pending operations or the contents of the other processing registers.

Example: **5** **[X]** **9** **[RCL 14]** **[=]**

This sequence produces the result  $5 \times R_{14}$ , where  $R_{14}$  is the number stored in register 14. The 9 is obliterated by the **[RCL 14]**.

Example: **[RCL 10]** **[RCL 11]** **[Y<sup>x</sup>]** **[RCL 12]** **[RCL 13]** **[=]**

This example is just to make the point again. The result of the sequence is  $R_{11}$  to the  $R_{13}$  power.

Example: 6  $\boxed{+}$  3 5  $\boxed{\ln x}$  2 1  $\boxed{=}$

In this example the answer produced is 27. The 21 entry obliterated the  $\ln 35$  present in the display register, but the pending addition and 6 stored in the internal processing registers were not affected. This example may appear silly, for there was no purpose served by the  $\ln 35$  calculation interjected where it was. However, the example shows that we could harmlessly perform the  $\ln 35$  calculation in the middle of the unrelated problem  $6 + 21 = 27$ . The fact that nothing was then done with the  $\ln 35$  result (such as storing it in memory or using it as the basis for a decision as discussed later) is not the main point. We could have done something with  $\ln 35$  before obliterating it. Hopefully this simple example gives you further insight into the workings of the internal processing registers and the display register.

Finally, a little more practical example.

Example: 9  $\boxed{+}$   $\boxed{2nd}$   $\boxed{\sqrt{x}}$   $\boxed{=}$

This may not appear to illustrate the same point; but it really does, as well as a second point. The answer obtained is  $9 + \sqrt{9} = 12$ . What occurred is this: After the 9  $\boxed{+}$ , addition was pending, with 9 stored in the processing registers. Furthermore, 9 was in the display register. Keying in 3 at this point would have had the same effect as what occurred as a result of  $\boxed{2nd}$   $\boxed{\sqrt{x}}$ : Namely  $\boxed{2nd}$   $\boxed{\sqrt{x}}$  replaced 9 in the display register with its square root just as though that number had been keyed in. With addition still pending  $\boxed{=}$  completed the calculation. This example and the ones preceding fall into the "peculiar-sequence" category. However, you will note that in the last example  $9 + \sqrt{9}$  was computed without entering 9 more than once, so there was some economy realized. You must be careful though, because if you press 9  $\boxed{+}$   $\boxed{=}$ , you do not get  $9 + 9 = 18$ , but get an error indication — you have not supplied a second operand. In the previous example, though, the square-root function supplies the operand as though it had been keyed in.



## VI. MEMORY REGISTERS

This section explains the use of the data memory which provides twenty registers for holding data. Throughout the manual this memory is referred to in several ways: Sometimes it is called the data memory (to distinguish it from the program memory), when there is little chance of confusion it is sometimes simply called the memory, and at other times it may be referred to as the data registers, the memory registers, or as the addressable registers. All of these terms mean the same thing. The twenty memory registers are identified by two digits (00 through 19). The registers themselves will be designated in text by a notation such as  $R_{14}$ , referring to data register 14. In the previous section,  $R_{14}$  was used to designate the quantity stored in that register. In the remainder of this manual, such usage would be confusing; therefore, the notation  $\bullet R_{14}$  will be used to designate the quantity stored in data register  $R_{14}$ .

It is usually arbitrary which registers you use for storing various quantities as long as you keep them straight. You should exercise discretion in using register  $R_{00}$  because the SR-52 occasionally uses this register for its own purposes, namely during polar/rectangular conversions and DSZ execution. We therefore advise you to form the habit of not using  $R_{00}$  for routine data storage.

### STORING QUANTITIES IN MEMORY

Consider evaluating an expression such as  $10.4x^5 - 30x^3 + 9x + 2$  for a value of  $x$  to be keyed in;  $x = 19.2818$ , for example. Obviously you don't want to key in this number more than once. To avoid this, simply store the number in memory the first time it is keyed in and recall it from memory whenever it is needed in the course of evaluating expressions. The following example shows another effective use of memory.

Example:

$$\frac{\sin\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right) + \cos\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right)}{\sin\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right) - \cos\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right)}$$

In addition to storing  $x$  for the evaluation of the expression  $(4x^3 - 7x + 1)/(8x^2 - 9)$ , the value of this expression should be stored also since it is the argument of both the  $\sin$  and the  $\cos$  functions. We will return to these examples soon. For now they are presented just to show typical situations for use of the memory.

In order to store the value contained in the display register into a given data register, you just key **STO** followed by the two-digit number of the memory register.

Example: Store 19.2818 into register 05.

Enter	Press	Display
19.2818	<b>STO</b> 05	19.2818

Example: Calculate  $\pi^2/3$  and store the result in  $R_{18}$ .

Enter	Press	Display
	<b>2nd</b> <b><math>\pi</math></b> <b>2nd</b> <b><math>x^2</math></b> <b><math>\div</math></b>	9.869604401
3	<b>=</b> <b>STO</b> 18	3.289868134

Example: Compute  $8 + \sin \sqrt{\pi}$  and store  $\sin \sqrt{\pi}$  into  $R_{00}$ .

Angle: Rad

Enter	Press	Display
8	<b>+</b>	8.
	<b>2nd</b> <b><math>\pi</math></b> <b>2nd</b> <b><math>\sqrt{x}</math></b>	1.772453851
	<b>sin</b> <b>STO</b> 00	.9797359325
	<b>=</b>	8.979735932

## RECALLING QUANTITIES FROM MEMORY

At any point in a calculation a value stored in memory may be introduced into the display register with the same effect as if it had been keyed in at that point. The command sequence for this is to press  $\boxed{\text{RCL}}$  followed by the two-digit number of the memory register in which the number is stored.

Example:  $A \times B^n = ?$

Where A is the value stored in  $R_{19}$

B is the value stored in  $R_{18}$

n is the value stored in  $R_{00}$ .

Press:  $\boxed{\text{RCL}} \ 19 \ \boxed{\times} \ \boxed{\text{RCL}} \ 18 \ \boxed{y^x} \ \boxed{\text{RCL}} \ 00 \ \boxed{=}$

Now we can combine the store and recall capabilities and work the examples mentioned before.

Example: Compute  $10.4x^5 - 30x^3 + 9x + 2$   
where  $x = 19.2818$ .

Enter	Press	Display	Remarks
19.2818	$\boxed{\text{STO}} \ 01 \ \boxed{y^x}$	19.2818	x in $R_{01}$
5	$\boxed{\times}$	2665249.492	$x^5$
10.4	$\boxed{-}$	27718594.71	$10.4x^5$
30	$\boxed{\times} \ \boxed{\text{RCL}} \ 01$	19.2818	
	$\boxed{y^x}$	19.2818	
3	$\boxed{+}$	27503532.57	$10.4x^5 - 30x^3$
9	$\boxed{\times} \ \boxed{\text{RCL}} \ 01$	19.2818	
	$\boxed{+}$	27503706.1	$10.4x^5 - 30x^3 + 9x$
2	$\boxed{=}$	27503708.1	Answer

Example: Compute the following function with  $x = \pi/9$  radians.

$$G(x) = \frac{\sin\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right) + \cos\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right)}{\sin\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right) - \cos\left(\frac{4x^3 - 7x + 1}{8x^2 - 9}\right)}$$

Angle:Rad

Enter	Press	Display	Remarks
	<b>2nd</b> <b>π</b> <b>÷</b>	3.141592654	
9	<b>=</b> <b>(</b> <b>STO</b> <b>01</b>	.3490658504	x in R <sub>01</sub>
	<b>y<sup>x</sup></b>	.3490658504	
3	<b>X</b>	.0425326155	x <sup>3</sup>
4	<b>-</b>	.1701304619	4x <sup>3</sup>
7	<b>X</b> <b>RCL</b> <b>01</b> <b>+</b>	-2.273330491	4x <sup>3</sup> - 7x
1	<b>)</b> <b>÷</b> <b>(</b>	-1.273330491	4x <sup>3</sup> - 7x + 1
	<b>RCL</b> <b>01</b> <b>2nd</b> <b>x<sup>2</sup></b>	.1218469679	
	<b>X</b>	.1218469679	
8	<b>-</b>	.9747757433	8x <sup>2</sup>
9	<b>)</b>	-8.025224257	8x <sup>2</sup> - 9
	<b>=</b>	.1586660323	y = $\frac{4x^3 - 7x + 1}{8x^2 - 9}$
	<b>STO</b> <b>02</b>	.1586660323	y in R <sub>02</sub>
	<b>sin</b> <b>+</b>	.1580011359	sin y
	<b>RCL</b> <b>02</b> <b>cos</b>	.9874389303	cos y
	<b>=</b> <b>÷</b> <b>(</b>	1.145440066	sin y + cos y
	<b>RCL</b> <b>02</b> <b>sin</b> <b>-</b>	.1580011359	sin y
	<b>RCL</b> <b>02</b> <b>cos</b> <b>)</b>	-.8294377944	sin y - cos y
	<b>=</b>	-1.38098369	Answer

The above example would have appeared complicated if the argument had to be evaluated anew each time it occurred. The memory has saved many keystrokes in the evaluation process.

## CLEARING THE DATA MEMORY

When you turn on the calculator, all addressable registers contain the number zero. As you proceed to use the memory, some of these registers acquire non-zero contents. It is frequently desirable to zero all of the data registers without turning the calculator off and then on again (which would also destroy the contents of the internal processing registers and program memory). This memory clearing could be done by storing zero in each one of the twenty memory registers, but it is far more convenient to use the Clear Memories instruction. To clear all twenty data registers at any time without affecting the internal processing registers, display, or program memory, simply press **2nd** **CMs** .

## DIRECT REGISTER ARITHMETIC

You can store a number at any time without affecting the display, the contents of the internal processing registers, or any pending operations. You can also perform arithmetic operations on the contents of data registers without affecting the calculation in progress. You can add the current value  $x$ , in the display register, to the contents of any memory register; you can subtract  $x$  from the register contents; you can multiply by  $x$ ; and you can divide the contents of a memory register by  $x$ . Of course until you recall the resultant quantity, you cannot see that any operation has taken place.

Again denoting the display register contents by  $x$ , and the memory register by  $nn$ , the direct register operations are performed as follows:

- To ADD  $x$  to the contents of  $R_{nn}$  press **SUM**  $nn$
- to SUBTRACT  $x$  from the contents of  $R_{nn}$   
press **INV** **SUM**  $nn$
- To MULTIPLY the contents of  $R_{nn}$  by  $x$  press **2nd** **PROD**  $nn$
- To DIVIDE the contents of  $R_{nn}$  by  $x$  press **INV** **2nd** **PROD**  $nn$

Example: Calculate  $4 \times 5.5$ ,  $3 \times 18.9$ , and  $11 \times 42.5$ ; accumulate the sum of these three products in register  $R_{01}$ .

Enter	Press	Display	Remarks
0	<b>STO</b> 01	0.	0. in $R_{01}$
4	<b>X</b>	4.	
5.5	<b>=</b>	22.	$4 \times 5.5$
	<b>SUM</b> 01	22.	Sum 22 into $R_{01}$
3	<b>X</b>	3.	
18.9	<b>=</b>	56.7	$3 \times 18.9$
	<b>SUM</b> 01	56.7	Sum 56.7 into $R_{01}$
11	<b>X</b>	11.	
42.5	<b>=</b>	467.5	$11 \times 42.5$
	<b>SUM</b> 01	467.5	Sum 467.5 into $R_{01}$
	<b>RCL</b> 01	546.2	Final sum

Example: Compute  $8\pi^2/3$  and store into  $R_{18}$ , then compute  $e^{1.08} + 4$  and divide the contents of  $R_{18}$  by this quantity, leaving the results in  $R_{18}$ .

Enter	Press	Display	Remarks
8	<b>X</b> <b>2nd</b> <b><math>\pi</math></b>	3.14159264	
	<b>2nd</b> <b><math>x^2</math></b> <b><math>\div</math></b>	78.95683521	
3	<b>=</b> <b>STO</b> 18	26.31894507	$8\pi^2/3$ in $R_{18}$
1.08	<b>INV</b> <b>lnx</b> <b>+</b>	2.944679551	$e^{1.08}$
4	<b>=</b>	6.944679551	$e^{1.08} + 4$
	<b>INV</b> <b>2nd</b> <b>PROD</b> 18	6.944679551	$\bullet R_{18}/(e^{1.08} + 4)$ in $R_{18}$
	<b>RCL</b> 18	3.789799785	

When direct register arithmetic results in overflow or underflow, a flashing display results, indicating the error.

## MEMORY/DISPLAY EXCHANGE

An additional memory instruction is available which combines the effects of a store and a recall instruction in a single step. This is the exchange instruction. You may remember it from the discussion on polar/rectangular coordinate transformations. Like all other memory operations, the exchange does not affect pending operations.

The effect of an exchange is to simply swap the contents of the display register with the contents of the memory register named in the command. The proper command sequence for an exchange is **2nd** **EXC** followed by the two-digit register number.

Example:

Enter	Press	Display	Remarks
18	<b>STO</b> 01	18.	18. in R <sub>01</sub>
42	<b>X</b>	42.	
3	<b>-</b>	126.	42 × 3
	<b>2nd</b> <b>EXC</b> 01	18.	126. in R <sub>01</sub>
	<b>=</b>	108.	
	<b>RCL</b> 01	126.	

The effect of the exchange function in this example was to store 126 (the result of  $42 \times 3$ ) into R<sub>01</sub> while recalling the earlier contents of R<sub>01</sub>, namely 18. The pending subtraction was unaffected by the exchange.

This instruction may be used for several purposes. One is to store a quantity that will be needed later while also recalling a quantity previously stored. Using exchange to accomplish this is not only efficient from an instruction (keystroke) point of view, but also from a memory usage point of view. One data register serves for two data, storing the second data value immediately upon releasing the first.

## SUPPLYING MISSING OPERANDS WITH MEMORY FUNCTIONS

There is one effect of all memory operations which has not been mentioned. After these operations have taken place, it is just as though the quantity in the display register had been keyed in. You may recall from the earlier discussion that single variable functions behave the same way. This allows us to key in  $9 \text{ [ + ] [ 2nd ] [ \sqrt{x} ] [ = ]}$  and obtain 12, whereas  $9 \text{ [ + ] [ = ]}$  gives an error indication (lacking a second operand) rather than 18 as one might guess. The square-root function replaced 9 in the display register with 3, exactly as though 3 had been keyed in, and thereby provided the second operand to go with the  $9 \text{ [ + ]}$ .

The memory functions behave the same way. You may not have noticed, but in the solution of the example on page 54, the following keystroke sequence was included (beginning in the second line):

$[ = ] [ ( ] \text{ [ STO ] 01 [ Y^* ] 3 [ X ] \dots}$  etc.

A more straightforward procedure would have been to perform the  $\text{[ STO ] 01}$  before the  $[ ( ]$ , and then to continue  $\text{[ RCL ] 01 [ Y^* ] 3 [ X ] \dots}$  etc; but this would seem a bit wasteful of keystrokes to recall a quantity into the display register which is already there. Therefore, it is tempting to eliminate the  $\text{[ RCL ] 01}$  from this sequence. However, an error condition would then occur, because we would be missing a first operand in the resulting sequence  $[ ( ] \text{ [ Y^* ] 3 [ X ]}$ . By placing the  $\text{[ STO ] 01}$  inside the parentheses, the effect just after the  $\text{[ STO ] 01}$  was as though the value in the display register had been keyed in again; and the first operand for the  $Y^X$  function was provided.

As a matter of fact, it is not even necessary to actually perform a store, recall, sum, or product to accomplish this effect. The single keys  $\text{[ STO ]}$ ,  $\text{[ RCL ]}$ ,  $\text{[ EXC ]}$ ,  $\text{[ SUM ]}$ , and  $\text{[ 2nd ] [ PROD ]}$  serve this same purpose even when not followed by a two-digit register number. Although it falls in the



category of a peculiar key sequence, a very useful construction is the following:

[...Sub-Expression...] ( ) STO OP....etc,

where OP denotes any operation. If the subexpression in brackets is needed again as the first operand in the parenthetical expression, then rather than storing it and immediately recalling it from memory, the STO alone can accomplish the same purpose. To demonstrate this let's first return to 9 + =

Example:

Enter	Press	Display	Remarks
9	+	9.	Addition pending
	STO	9.	Does not store, but has effect of reentering contents of display register
	=	18.	9 + 9

Example: Evaluate  $1.401103287^{1.401103287}$

Enter	Press	Display
1.401103287	$y^x$	1.401103287
	STO	1.401103287
	=	1.604057054

Example:  $\frac{(28.7 - 21)}{(42 - 9.8)} \log \frac{(28.7 - 21)}{(42 - 9.8)} = ?$

Enter	Press	Display	Remarks
	( (		
28.7	-	28.7	
21	) ÷ (	7.7	(28.7 - 21)
42	-	42.	
9.8	) ) X	.2391304348	$\frac{(28.7 - 21)}{(42 - 9.8)}$
	2nd log =	-.1485873176	

## VII. RUNNING PRERECORDED PROGRAMS

The most effective use of the SR-52 is realized when you run a stored program. The instructions for using each program are unique for that particular program, so we cannot attempt to properly cover every program here. Programs in the TI Program Library (such as the BASIC PROGRAM LIBRARY included with your SR-52) have detailed instructions for using each program. In the case of programs you have written yourself, we strongly recommend that you write down the detailed operating instructions as an essential part of the programming task. Otherwise, you will be surprised how easy it is to forget how to use even a well-designed program. The next sections deal more with programming your SR-52, both the mechanics and questions of technique and style. There are two basic steps necessary to run all prerecorded programs: reading the magnetic card and beginning execution.

© 2010 Joerg Woerner  
Datamath Calculator Museum

### READING A MAGNETIC CARD

A magnetic card provides the means for storing 224 prerecorded instructions. This is the exact size of the program memory. The first time a read operation is performed, **CLR** is pressed to ensure that side A of the card is read into the top half of program memory (locations 000-111). The second read operation then enters card data into the bottom half of program memory (locations 112-223). Subsequent read operations alternate between the top and bottom halves of program memory.

**CAUTION:** Prerecorded magnetic cards may be damaged or altered if exposed to dust or foreign materials, permanent magnets, or electromagnetic fields such as near electric motors or power transformers.

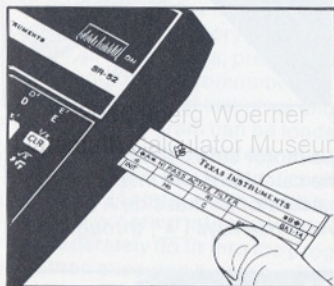
Refer to *Maintenance and Service Information* in this manual for instructions in caring for and using magnetic cards.

The following steps should be followed to read the contents of a magnetic card into the program memory.

1. Press **CLR**.
2. With the power switch on, read side A of the prerecorded card:

**2nd read** (Insert card side **◀A▶**)

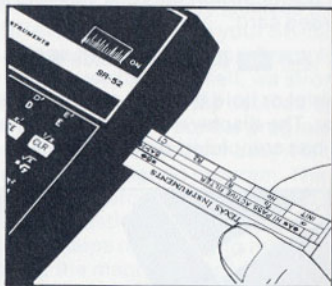
Do not restrict or hold the card after it is caught by the drive motor. The display will remain blank until the calculator has completed reading side A.



3. After the drive motor stops, remove the card from the left side of the calculator and read side B of the prerecorded card as follows:

**2nd** **read** (Insert card side **◀B▶**)

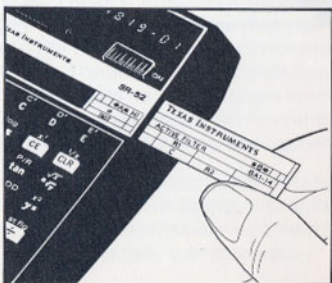
The display will remain blank until the calculator has completed reading side B.



© 2010 Joerg Woerner

Datamath Calculator Museum

4. After the drive motor stops, remove the card from the left side of the calculator and insert it into the upper slot on the calculator so that side A of the card shows in the window above keys **A** through **E**.



5. If the display flashes immediately following step 2 or 3, repeat the procedure beginning with step 1. If difficulty persists, refer to *Maintenance and Service Information* in this manual.

The program is now loaded into program memory and will remain there until the calculator is either turned off, another card is read, or new instructions are keyed in by passage to the learn mode.

If for some reason it is necessary to read a program segment into the second half of program memory without first reading a segment into the first half, the keystroke sequence is **CLR** **2nd** **read** **HLT** **2nd** **read** , followed by insertion of the card.

Of considerable significance is the fact that loading a program into the program memory does not affect the memory registers, program flags, program counter, or display format. In addition, by remembering which half of the program memory was read last (thus allowing step 1 to be omitted) a program can be loaded without affecting internal processing registers or the displayed number. This means that if you find you cannot fit an entire program into 224 steps, there is still a good chance to partition the total program into segments or load modules which separately do fit into the program memory. The program is then executed by sequentially reading and running the load modules.

## EXECUTING A PROGRAM

Execution of a well-designed program is started by pressing one of the ten user-defined label keys: **A** through **E** or **2nd A** through **2nd E**. These keys are special in that pressing one causes the calculator to position the program counter to the point in program memory so labeled and to switch into the run mode as soon as the counter is positioned.

Another method to start program execution is to position the program counter to the desired starting point while the SR-52 is in the calculate mode, perhaps enter data with the keyboard, and finally enter the run mode by pressing **RUN**. Positioning of the program counter may be accomplished with the go-to instruction (abbreviated **GTO** on the keyboard) which will be discussed in the *transfer instructions* section. More likely, if the program counter needs positioning it is to the very top of the memory; and one can accomplish this by **2nd RST**. If the program has been partitioned, the reset instruction may not be appropriate as it also clears program flags and subroutine return-pointer registers.

## VIII. GENERAL PROGRAMMING INSTRUCTIONS

From what you have learned in the preceding sections, you are already basically prepared to write useful programs for the SR-52. This comes from the fact that each keystroke in the calculate mode may be stored in a program location in the learn mode (where it is called an instruction). And when this instruction is executed in the run mode it has the very same effect as would have been obtained by that keystroke in the calculate mode.

There are still a few ingredients missing, however, if you are to make the best use of the SR-52. The first group of ingredients is essential: a handful of controls which allows you to specify when the program should halt for data entry or for you to look at the answer, the label statements which allow you to conveniently start execution at different points in the program and that sort of thing.

© 2010 Joerg Woerner

The second group of ingredients includes those instructions which substitute for your eyes and your judgment during program execution. These instructions are those which determine what shall be done next, based on the conditions which have been obtained so far. You cannot directly engage in this process in the run mode; since everything happens too fast, and the display is blanked. Thus you must make the basis of any decisions known to the calculator. For example, in a trial-and-error solution to a problem, you would stop when the answer had been "bracketed" to within the tolerance you desired. The SR-52 has a number of instructions which, when used in combination with the others, can perform this type of decision making.

The third set of programming ingredients will complete your full capabilities as an SR-52 user: They relate to a very orderly method of problem solving which the SR-52 provides. The method consists of defining the answer to a

problem from the top down: You write the answer in terms of other quantities. Rather than defining and evaluating those quantities on the spot, you just give them a name (which is a label to the SR-52) and go on with the main problem definition. When you are ready to define those quantities which have been only named until now, you do so immediately following the appropriate label.

The actual SR-52 evaluation of the problem solution in the run mode will invoke all those detailed definitions which had been delayed until you completed the statement of the whole problem at a higher level. It will invoke those definitions simply upon recognizing the names (or labels) assigned, and will automatically insert the labeled definitions and evaluations of the deferred quantities. This method of top down problem solution is made possible by the subroutine capability. The SR-52 provides for automatic performance of program execution at three levels. This means that not only can you simply give a deferred quantity a name in the main problem definition, you can also (in defining those deferred quantities) give names to quantities whose definitions are to be deferred even further!

Addition of these three ingredients to your programming skills is the primary function of the remaining sections of this manual.

## ELEMENTS OF PROGRAM EXECUTION

There are 224 locations in program memory, numbered from 000 through 223. Each location can hold one keystroke. Do not count **2nd** as a keystroke, for example **2nd** **√x** requires only one program memory location.

When the calculator is in the run mode, the sequencing of the program steps is accomplished by means of the *program counter*. This may be thought of as a marker



which moves through the program memory which indicates the next instruction to be executed. The normal operating sequence in the run mode is very straightforward:

Instruction (keystroke) executed ; program counter advanced one location.

Instruction executed ; program counter advanced.

Instruction executed ; program counter advanced.

•

•

•

(etc.)

In this "normal" sequence, the SR-52 steps through program memory, executing the instructions in exactly the order in which they appear. If the calculator attempts to go past location 223 it will flash the value in the display.

Certain types of instructions alter this simple top-to-bottom execution of the program. These are the *transfer instructions*. Detailed discussion of these is deferred until a later section; but this is an appropriate point to explain how these instructions alter the normal sequence.

There are two types of transfer instructions, unconditional transfers and conditional transfers. The names themselves convey the difference between these two types. When an unconditional transfer instruction is executed, it unconditionally repositions the program counter to the location specified in the transfer instruction. This destination therefore becomes the next instruction executed, and the program counter is then advanced a step at a time until another transfer instruction is encountered.

The conditional transfer, or branching instruction, is similar to an unconditional transfer except for one thing: A branching instruction first performs some test (for example, is a flag set or is the display register positive?); the transfer occurs only if the test is affirmative. Otherwise, the program counter is advanced (or "falls through") to the next location, as usual.

The time history of a typical program using transfers might be as follows:

Execute instruction; advance counter.	Execute instruction; advance counter.
Execute instruction; advance counter.	③ Perform test (affirmative result); Reposition counter.
•	④ Execute instruction; advance counter.
•	•
•	•
Execute instruction; advance counter.	•
① Reposition counter.	•
② Execute instruction; advance counter.	•
•	Execute instruction; advance counter.
•	⑤ Perform test (negative result) advance counter.
•	Execution instruction; advance counter.
•	•
	•
	(etc.)

In this illustrative sequence, an unconditional transfer occurred at point 1, transferring to the location of the instruction executed next at point 2. At point 3 a branching instruction was executed with an affirmative result. This caused transfer to the location of the instruction next executed at point 4. At point 5 of the process a branching instruction was executed with negative result and the instruction counter fell through to the next program location.

## MECHANICS OF PROGRAMMING

What are the steps involved in creating a program for the SR-52? From beginning to end they are essentially the following:

1. Gather equations which pertain to problem.
2. Define numerical approach (algorithms) for solving equations.
3. Determine how you would like the program to be used (input data, quantities computed, operating instructions, label assignments, etc.).
4. Conceptualize flow of program. If it is complicated (many transfers) then flow-charting is recommended. Even better, try to simplify program structure after it is flow-charted.
5. Begin making data register assignments. This task continues through the programming process. (Do not store a quantity in memory without making a written note that the register in question contains that quantity.)
6. Actual coding: Write down the instructions on a coding form which numbers the locations.
7. Make corrections to code, memory assignments, or even procedure for program use if necessary.
8. Place SR-52 in learn mode.
9. Key in program.
10. Place SR-52 in calculate mode.
11. Record program on card if desired.
12. Check out program on test problems.
13. Make any necessary corrections.
14. Document completed program thoroughly.

If these steps are performed deliberately, you are more likely to be satisfied with the result. For example, if you do not spend some time defining how you would like the program to operate (the third step in the foregoing list), then the user features of the resulting program may leave you less than satisfied. Preliminary efforts are well spent; because after you have designed and documented the program you can conveniently use it at any time.

Another way of stating this advice: The programming language of the SR-52 is so easy to use that the coding phase of programming is simple. Free from this concern, you can spend most of your effort in the definition of the problem and in enforcing the requirement that the resulting program conveniently meet your problem-solving needs. Comparison of BA1-08 with the compound interest programs developed at the conclusion of Section I should illustrate the point.

© 2010 Joerg Woerner  
Datamath Calculator Museum

## DEVELOPMENT OF PROGRAMMING STYLE

We would like to make a last point about the nature of the programming process, whether for an SR-52 or for a large-scale computer. There is no single correct programming solution to a problem. Just as no two writers use exactly the same words to describe the same thing, no two programmers use exactly the same instruction sequences to solve a given problem. As you gain experience in programming the SR-52, you will develop your own unique style. That style may become one of incredible craftiness and ingenuity which makes frequent use of all the instructions available. Or it might become one of conservative and straightforward coding, using primarily the more basic instructions, taking up more program memory space, but so clear in purpose that program operation can easily be discerned simply by inspecting the code. Each of these style extremes has advantages and shortcomings. The best style for you is the one that best meets *your* needs: If you can solve your problems without use of branching or indirect addressing, then just don't use those instructions (until your needs change).

Many SR-52 users will solve their problems very comfortably using only a portion of the SR-52 capability. If on the other hand your problems require complicated but more efficient program structures, you will find yourself developing a style closer to the crafty-but-obscure end of the spectrum.

# IX. ELEMENTARY PROGRAMMING

## USING LABELS

When you execute a program, it is necessary that the program counter be properly positioned and that the calculator be switched to the run mode. Positioning the program counter can be accomplished with the help of labels. Automatic switching to the run mode may also be accomplished by using the special labels **A** through **E** and **2nd** **A** through **2nd** **E**. These special keys are also known as the *user-defined keys*, because the effect of pressing one is to execute the program defined and so-labeled by the user. We will first discuss these special labels and then discuss the other types of label.

Using the concept of the program counter developed in the last section, the effect of pressing one of the user-defined keys in the calculate mode is as follows:

1. Positions the program counter to the first location after the corresponding label in program memory.
2. Switches the calculator into the run mode.

This means that if you wish to begin execution at (or transfer to) a given location in program memory, the simplest technique is to provide a label in the program code just prior to that desired location. This label is placed in program memory by the instruction **2nd** **LBL**, followed by the label itself:

Example: Suppose you would like to cause the following process to take place simply by pressing the key **C** in the calculate mode:

Key

**STO**

**0**

**1**

**X**

**RCL**

**0**

**2**

**sin**

**=**

(etc.)

Then, code the following sequence into program memory:

Key

**2nd** **LBL**

**C**

**STO**

**0**

**1**

**X**

**RCL**

**0**

**2**

**sin**

**=**

(etc.)

Wherever that sequence occurs in program memory, pressing **C** in the calculate mode will find it and cause execution to take place starting at that point.

Labels (of any type) may be placed anywhere in a program instruction sequence without altering the meaning of that sequence. They are simply ignored during instruction processing except for the purpose of locating a desired point in program memory and do not affect pending operations. This statement is not intended to mean that a label in a program can interrupt a sequence such as **RCL 03**, where several program locations are involved in defining a single processing action.

You do not key in labels after the rest of the code is written. You conceive the need for and define your labels as part of the program design process. They are keyed into program memory along with the rest of your code, just as though they were any other instruction steps.

Actually, any key (including second functions) may be used as a label except the following:

**2nd**, **LRN**, **INS**, **2nd del**, **SST**, **2nd bst**, and the digits **0** through **9**. Note that the second functions of the digits may be used as labels: **2nd list** and **2nd 1** through **2nd 9**.

We have explained how to use the user-defined keys as labels. The other labels are not used in an identical way. For example, just pressing **sin** in the calculate mode would not cause the calculator to search for a location so labeled and begin execution there, even if there were a **2nd LBL sin** sequence somewhere in the program. We will discuss how to use such labels in the next section. However you should know that, however they are used, the placement of those labels in the code sequence is identical to what has already been described: To establish a location labeled "sin" in the program memory, just key the instructions **2nd LBL sin** in the locations immediately preceding the one which is to acquire that name. No two program locations in program memory may have the same label.



## USING RUN AND HALT

You have learned how to start program execution, but not how to stop it for keying in data or for looking at results. Halting a program at specified points is accomplished by the halt instruction **HLT**. Upon execution of a halt instruction encountered in the program, the program counter indicates the first location after the halt and program execution stops; there is an immediate transition to the calculate mode, and hence calculator control is returned to the keyboard. By pressing **RUN**, the run mode is restored and execution is resumed from the point indicated by the program counter.

Thus sequences of halt instructions (in the program) and **RUN** commands (from the keyboard) enable control to be passed back and forth between the program and the keyboard.

The halt command entered from the keyboard when the SR-52 is in the run mode will stop execution of a program and return control to the keyboard. The program counter is left wherever it happened to be at the time of program interruption. Program execution will be resumed at that location when **RUN** is pressed. The following example shows how you might use what you've just learned to calculate the volume of a right circular cylinder.

Example: Calculate the volume of a right circular cylinder of radius  $r$  and height  $h$ ;  $V = \pi r^2 h$ . Program Operation Desired: Key in  $r$ , press **A**, then key in  $h$ , press **RUN** and see the answer.

Solution:

Key	Remarks
<b>2nd</b> <b>LBL</b>	Labels start of program
<b>A</b>	to calculate volume.
<b>2nd</b> <b>x<sup>2</sup></b>	$r^2$ (Remember $r$ was entered
<b>X</b>	before <b>A</b> was pressed).
<b>2nd</b> <b><math>\pi</math></b>	
<b>X</b>	$\pi r^2$ in display register.
<b>HLT</b>	Allows $h$ to be entered.
<b>=</b>	$V = \pi r^2 h$ .
<b>HLT</b>	Program stops; $V$ displayed.

## ENTERING YOUR PROGRAM

The sequence for keying your program into program memory is as follows:

1. With the SR-52 in the calculate mode press **2nd** **rs8t**, which positions the program counter to 000, the top of the memory.
2. Then press **LRN**, placing the SR-52 into the learn mode. You will see five digits in the display. The first three digits show the position of the program counter. The last two digits show the two-digit code of the key instruction currently stored in that location.
3. Key in your program completely: one step at a time, including all labels, not forgetting any necessary **2nd** prefixes.
4. Make sure your program did not exceed the program memory size. After filling location 223 the SR-52 switches to the calculate mode, and the five-digit display format of the learn mode will be conspicuously absent.

5. Switch from the learn mode to the calculate mode by pressing **LRN**.
6. Run a test problem and correct or edit your program as required.
7. You are now ready to record your program if desired.

## RECORDING A MAGNETIC CARD

You can permanently record any program stored in the calculator on a blank magnetic card furnished with your calculator. The magnetic card provides permanent storage for your program — the contents of the calculator memory are lost if the calculator is turned off. The magnetic card is designed to store one-half the calculator program memory (program locations 000 through 111) on side A of the card and the other half of the program (program locations 112 through 223) on side B of the card. For additional blank magnetic cards, check with your dealer or call Consumer Relations at the toll-free numbers listed on the inside front cover of this manual.

The procedure for recording a program is similar to reading a program. The write command is the inverse of a read; therefore to write press **INV** **2nd** **read** \*. To write the first 112 program steps on a card, use the sequence **CLR** **INV** **2nd** **read** \* (in the calculate mode) and feed the card into the slot. If necessary, the remaining 112 locations may be recorded by again pressing **INV** **2nd** **read** \*.

As in reading cards, if it becomes necessary to record only the second half of program memory use the key sequence **CLR** **2nd** **read** **HLT** **INV** **2nd** **read** \*. The **CLR** **2nd** **read** **HLT** portion of this sequence guarantees that the next read or write to program memory will involve the second half of program memory.

\*Magnetic cards are protected against accidental writing. Therefore, before writing, place black self-adhesive tabs (supplied with calculator) over the write-protect windows at the tip of the arrows on the card.

## EDITING PROGRAMS

In the process of keying in a program you have the following capabilities on the SR-52: 1) display the location and instruction currently present, 2) replace that instruction with another, 3) delete that instruction and close up the hole, and 4) make a space for a new instruction to be inserted. In addition, you may single-step forward or backward through the program. These capabilities permit you to make corrections or modifications to a program without reentering instructions which require no changes.

### Displaying the Program

In the learn mode, the display is designed to show you where the program counter is positioned in program memory and the instruction presently residing in that location. The first three digits are simply the location. The second two digits, set off from the first three, are indicative of the instruction present according to simple rules.

Datamath Calculator Museum

Each instruction is assigned a two-digit code based on the location of the corresponding key on the SR-52 keyboard. The first digit denotes in which of the nine rows (numbered 1 through 9 from top to bottom) the key is located. The second digit establishes which of the five columns, numbered from left to right. To distinguish second functions from first functions, the columns are labeled 1 through 5 for first functions and 6 through 0 for second. The digit keys 0 through 9 are designated by the codes 00 through 09. The null instruction is also represented by 00. Refer to the following key-code cross references:

Figure 1a. Program Key Codes

Key	Key Code	Key	Key Code	Key	Key Code	Key	Key Code	Key	Key Code
<b>A'</b>	16	<b>B'</b>	17	<b>C'</b>	18	<b>D'</b>	19	<b>E'</b>	10
<b>A</b>	11	<b>B</b>	12	<b>C</b>	13	<b>D</b>	14	<b>E</b>	15
	—		27*	<b>log</b>	28	<b>x!</b>	29	<b>1/x</b>	20
<b>2nd</b>	—	<b>INV</b>	22	<b>Inx</b>	23	<b>CE</b>	24	<b>CLR</b>	25
<b>IND</b>	36	<b>D.MS</b>	37	<b>D/R</b>	38	<b>P/R</b>	39	<b><math>\sqrt{x}</math></b>	30
<b>LRN</b>	—	<b>sin</b>	32	<b>cos</b>	33	<b>tan</b>	34	<b><math>x\sqrt{y}</math></b>	35
<b>LBL</b>	46	<b>CMs</b>	47	<b>EXC</b>	48	<b>PROD</b>	49	<b><math>x^2</math></b>	40
<b>GTO</b>	41	<b>STO</b>	42	<b>RCL</b>	43	<b>SUM</b>	44	<b><math>y^x</math></b>	45
<b>rtn</b>	56	<b>fix</b>	57	<b>dsz</b>	58	<b><math>\pi</math></b>	59	<b>st flg</b>	50
<b>SBR</b>	51	<b>EE</b>	52	<b>(</b>	53	<b>)</b>	54	<b><math>\div</math></b>	55
<b>del</b>	—		67*		68*		69*	<b>if flg</b>	60
<b>INS</b>	—	<b>7</b>	07	<b>8</b>	08	<b>9</b>	09	<b>X</b>	65
<b>bst</b>	—		77*		78*		79*	<b>if err</b>	70
<b>SST</b>	—	<b>4</b>	04	<b>5</b>	05	<b>6</b>	06	<b>-</b>	75
<b>rset</b>	86		87*		88*		89*	<b>if pos</b>	80
<b>HLT</b>	81	<b>1</b>	01	<b>2</b>	02	<b>3</b>	03	<b>+</b>	85
<b>read</b>	96	<b>list</b>	97	<b>pri</b>	98	<b>pap</b>	99	<b>if zro</b>	90
<b>RUN</b>	91	<b>0</b>	00	<b>.</b>	93	<b>+/-</b>	94	<b>=</b>	95

— Key codes omitted for functions that cannot be stored in a program.

\*Second functions of unmarked keys that may be used as labels, however, first-function numeral keys may not be used as labels.

Figure 1b. Program Key Codes

Key Code	Key	Key Code	Key	Key Code	Key
00	0	30	2nd $\sqrt{x}$	60	2nd if flg
01	1	32	sin	65	X
02	2	33	cos	67*	2nd 7
03	3	34	tan	68*	2nd 8
04	4	35	$x\sqrt{y}$	69*	2nd 9
05	5	36	2nd IND	70	2nd if err
06	6	37	2nd D.MS	75	-
07	7	38	2nd D/R	77*	2nd 4
08	8	39	2nd P/R	78*	2nd 5
09	9	40	2nd $x^2$	79*	2nd 6
10	2nd E'	41	GTO	80	2nd if pos
11	A	42	STO	81	HLT
12	B	43	RCL	85	+
13	C	44	SUM	86	2nd rset
14	D	45	$y^x$	87*	2nd 1
15	E	46	2nd LBL	88*	2nd 2
16	2nd A'	47	2nd CMs	89*	2nd 3
17	2nd B'	48	2nd EXC	90	2nd if zro
18	2nd C'	49	2nd PROD	91	RUN
19	2nd D'	50	2nd st flg	93	.
20	2nd $1/x$	51	SBR	94	+/-
22	INV	52	EE	95	=
23	Inx	53	(	96	2nd read
24	CE	54	)	97	2nd list
25	CLR	55	$\div$	98	2nd prt
27*	2nd INV	56	2nd rtn	99	2nd pap
28	2nd log	57	2nd fix		
29	2nd $x!$	58	2nd dsz		
		59	2nd $\pi$		

\*Key codes which should only appear as labels.

Through use, you will soon become familiar enough with the codes for the more common instructions such as **2nd** **LBL** , **(** , **)** , that constant reference to the keyboard will not be necessary. The others are quickly interpreted by reference to a keyboard, of course.

## Replacing an Instruction with Another

In accordance with the foregoing discussion the display shows the position of the program counter (i.e., the next instruction location) and the instruction currently there. To substitute another instruction into that location, simply key in the new instruction and it will replace the old one in that location.

## Deleting an Instruction

One may delete the displayed instruction and move all those beyond it up to fill the space. This operation is performed by pressing **2nd** **del** . As a consequence of the closing-up operation in instruction deletion, the last location in program memory (223) would be undefined. Consequently, a null instruction is placed there as a result of a delete operation.

## Inserting an Instruction

Sometimes you may wish to insert an instruction at the location indicated by the display without destroying the one already there. This involves a two step process:

Step 1. Press **INS** to push down the displayed instruction (and those beyond it in program memory) one location, leaving a null instruction in its place.

Step 2. Key in the desired instruction in place of the null instruction.

Note that if several sequential instructions are to be inserted the procedure would be: **INS** (key 1), **INS** (key 2), **INS** (key 3), etc., where key 1, key 2 and so on represent the instruction keystrokes to be inserted. Finally, note that each instruction insertion causes an instruction at location 223 to be lost, being already in the last location of program memory.

## Single-Step and Backstep

You will frequently wish to examine portions of your program as stored in memory. The single-step instruction allows you to do this. Pressing **SST** in the learn mode will increment the position of the program counter by one without affecting the stored program in any way. Pressing it repeatedly, therefore, allows you to sequentially step through the program memory, observing the codes for instructions stored there. If you attempt to single step past 223, the SR-52 will switch into the calculate mode.



Similar to the effect of the single-step instruction, the backstep instruction allows you to reposition the program counter by one location without affecting the stored program. The backstep instruction is commanded by **2nd** **bst**. It causes the program counter to decrement by one, or move upward in the program memory. A common use for this instruction is to go back and verify that an instruction just keyed in is the one desired. More generally, the backstep instruction combines with the single-step instruction to provide you with all the tools you will need for convenient program editing.

The single-step instruction is executable in the calculate mode as well as the learn mode. In the calculate mode the effect of **SST** is to cause actual execution of the stored program, one instruction at a time. Each time **SST** is pressed, the instruction located at the current position of the program counter is executed and the program counter is advanced or repositioned just as it would have been in the run mode. Conditional branches behave just as in the run mode, for example. Sometimes several single-step keystrokes are required before anything appears to happen; but this is only because the operation in process is a multistep one. For example, the sequence **RCL** **10** would require three single-step keystrokes before the recall of  $R_{10}$  would actually take place. The primary function of this single-step type of execution is to assist in program checkout and "debugging".

## PRACTICE PROBLEMS

Example: Write a single program to convert temperature from degrees–Fahrenheit to degrees–Celsius using label A, and from Fahrenheit to degrees–Kelvin using label B. The relevant equations are:

$$C = (5/9) (F - 32)$$

$$K = C + 273.15$$

Possible Solution:

LOC	CODE	KEY
000	46	*LBL
	12	B
	85	+
	04	4
	09	9
005	01	1
	93	.
	06	6
	07	7

LOC	CODE	KEY
	46	*LBL
010	11	A
	75	-
	03	3
	02	2
	95	=
015	65	×
	05	5
	55	÷
	09	9
	95	=
020	81	HLT

\*Denotes 2nd function key

The above coding is not the straightforward solution you might have expected, however, it is a prime example of how two seemingly unrelated problems can be programmed to function together.

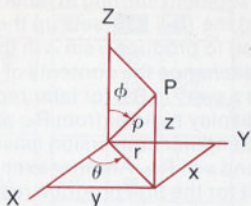
Example: Write a program to convert from spherical to rectangular coordinates. Design the program so as to operate as follows:

Enter  $\rho$ ,  $\phi$ , and  $\theta$  respectively through the A, B, and C keys (in any order). Find x, y, and z via the D key, with x given first, then press **RUN** to display y, and **RUN** again to display z.

$$z = \rho \cos \phi$$

$$x = \rho \sin \phi \cos \theta$$

$$y = \rho \sin \phi \sin \theta$$



Possible Solution:

LOC	CODE	KEY
000	46	*LBL
	11	A
	42	STO
	00	0
	01	1
005	81	HLT
	46	*LBL
	12	B
	42	STO
	00	0
010	02	2
	81	HLT
	46	*LBL
	13	C
	42	STO
015	00	0
	03	3
	81	HLT

LOC	CODE	KEY
	46	*LBL
	14	D
020	43	RCL
	00	0
	01	1
	42	STO
	00	0
025	00	0
	43	RCL
	00	0
	02	2
	39	*P/R
030	48	*EXC
	00	0
	00	0
	42	STO
	00	0
035	04	4

LOC	CODE	KEY
	43	RCL
	00	0
	03	3
	39	*P/R
040	48	*EXC
	00	0
	00	0
	81	HLT
	43	RCL
045	00	0
	00	0
	81	HLT
	43	RCL
	00	0
050	04	4
	81	HLT

\*Denotes 2nd function key

Again, the proposed solution is a bit tricky; and provided that your program gives the correct answer, ours is no better. Even so, it may be instructive to see how the program segment starting at label D works. The nine steps preceding the **2nd P/R** sets up the polar-rectangular conversion to produce  $\rho \sin \phi$  in the display and  $z$  in  $R_{00}$ . Next we exchange the contents of the display and  $R_{00}$  and store  $z$  away in  $R_{04}$  for later recall. Next we recall  $\theta$  into the display register from  $R_{03}$  and perform another polar-rectangular conversion leaving  $y$  in the display register and  $x$  in  $R_{00}$ . Another exchange places  $x$  where we want it for the first program result, the display register. Following the HLT to allow us to see the value of  $x$ ,  $y$  is recalled from  $R_{00}$  and another program halt is provided. Finally,  $z$  is recalled from  $R_{04}$  and the program is halted for the last time.

Example: Write a program to compute the average value of any number of values  $x_1, x_2, \dots, x_N$ . The desired operation is that after initializing using key **E**, the values of  $x_1, x_2$ , etc. are entered using key **A** for each entry. It should not be necessary to know the total number of values to be entered ahead of time; and at any stage in the process the average of the  $x$  values up to that time should be given by using key **B**.

## Possible Solution:

LOC	CODE	KEY
000	46	*LBL
	15	E
	47	*CMs
	81	HLT
	46	*LBL
005	11	A
	44	SUM
	00	0
	01	1
	01	1
010	44	SUM
	00	0
	02	2
	81	HLT

LOC	CODE	KEY
	46	*LBL
015	12	B
	53	(
	43	RCL
	00	0
	01	1
020	55	÷
	43	RCL
	00	0
	02	2
	54	)
025	81	HLT

\*Denotes 2nd function key

None of these problems are trivial. All are sufficiently complicated that you would be more likely to solve them correctly through programming than by attempting to perform all of the required operations directly from the keyboard in the calculate mode. This is important, and we wanted to demonstrate it to you through these examples. (Try performing these problem solutions in the calculate mode.) In the introduction we said "the SR-52 is a problem solving machine," and we hope you now see one of the reasons this is true. The type of problem you will be able to solve reliably through the several-stage process (problem definition, coding, program execution) using the SR-52 is much, much more complicated than would be possible without this calculator. The execution of some programs will produce the equivalent result of automated hundreds or even thousands of keystrokes perhaps involving decisions to be made, as well.

## X. TRANSFER INSTRUCTIONS

The background relating to transfer instructions was first presented in Section VIII. However, this section enlarges on that basic information to help you fully understand the uses of the transfer instructions. In general, transfer instructions are known as branching instructions. There are two types: *unconditional* and *conditional*. In this manual, only the *conditional* transfer instruction is referred to as a branching instruction.

### UNCONDITIONAL TRANSFER INSTRUCTIONS

There are two types of unconditional transfer instructions: the go-to ( **GTO** ) instruction and the subroutine **SBR** calls. We will not discuss the subroutine type here, deferring that discussion until the next section, which is devoted to subroutines. Unconditional transfer to a given program location occurs when **GTO**, followed by the three-digit address (program location number) of the destination, is encountered in the program. For example the sequence **GTO** 037 causes immediate repositioning of the program counter to location 037. The destination of a go-to instruction can also be specified by means of a label name following the go-to statement. For example, the sequence **GTO** **sin** would cause a transfer to the first location in the program after the sequence **LBL** **sin** .

Actually, either type of go-to command, specified by a three-digit address or by a label name, can be executed in the calculate mode. However, when executed from the keyboard the only effect is to reposition the program counter. The calculator remains in the calculate mode following the **GTO** command. Thus, the effect of pressing **GTO** **A** would be different from simply pressing **A** in this respect.

## CONDITIONAL TRANSFERS (BRANCHING INSTRUCTIONS)

There are basically four different branching instructions available (in addition to the DSZ, which is discussed later). They are the **2nd if neg**, **2nd if pos**, **2nd if err**, and **2nd if zro** instructions. Each of these has an inverse instruction as well. These four types of branching instructions all operate the same way. At the time of encountering them, a test is made to determine if a certain condition is true. If the result is affirmative, transfer is made to the location specified by a three-digit address or label name immediately following the instruction, just as in the unconditional transfers. A false result causes no such transfer to take place, and the program counter continues to the next program location immediately following the three-digit address or label.

The inverse instructions to these four are realized by prefixing the basic branching instructions with **INV**. These inverse branching instructions test to determine if a certain condition is false. If the condition is false, a transfer occurs and if true, no transfer takes place.

The eight types of tests made by these branching instructions are as shown below:

- |            |               |  |  |
|------------|---------------|--|--|
| <b>2nd</b> | <b>if err</b> | Is an error condition present (overflow, underflow, invalid argument, invalid operation)? Transfer if answer is yes. |  |
| <b>INV</b> | <b>2nd</b>    | <b>if err</b>  | Is an error condition present? Transfer if answer is no.                       |
| <b>2nd</b> | <b>if flg</b> | Is the program flag specified by the next digit (0 through 4) set? Transfer if answer is yes.                        |  |
| <b>INV</b> | <b>2nd</b>    | <b>if flg</b>  | Is the program flag specified by the next digit set? Transfer if answer is no. |
| <b>2nd</b> | <b>if pos</b> | Is the content of the display register positive (equal to or greater than zero)? Transfer if answer is yes.          |  |
| <b>INV</b> | <b>2nd</b>    | <b>if pos</b>  | Is the content of the display register positive? Transfer if the answer is no. |
| <b>2nd</b> | <b>if zro</b> | Is the content of the display register exactly zero? Transfer if the answer is yes.                                  |  |
| <b>INV</b> | <b>2nd</b>    | <b>if zro</b>  | Is the content of the display register zero? Transfer if the answer is no.     |

These branching instructions do not affect pending operations, hence they can appear anywhere desired in the program, except between multilocation operands such as **RCL** 03.



The **if err** and **if flg** branching instructions require a bit more explanation. You will note that an **if err** instruction tests for an error condition, that would cause a flashing display in the calculate mode. Such an error will not halt execution of a program unless programmed to do so by the **if err** instruction. The display will flash after the program halts unless **CE** or **CLR** were executed in the program before it halted. Using **CE** will stop a flashing display without affecting the displayed number.

The **if flg** instruction refers to the five program flags that are indicators which can be set (turned on, raised, set to 1) or reset (turned off, lowered, reset to zero) by commands in the program code or directly from the keyboard. Manipulation of these flags is discussed in the following subsection.

You may test your understanding of the branching instructions by studying the following examples. Each example is a segment of code whose effect is given under explanation.

Example: © 2010 Joerg Woerner  
Datamath Calculator Museum

LOC	CODE	KEY
000	22	INV
	70	*if err
	01	1
	00	0
	03	3
005	81	HLT

\*Denotes 2nd function key

Explanation. If no error condition is present transfer is made to location 103; otherwise the program executes the next instruction (HLT).

Example:

LOC	CODE	KEY
000	60	*if flg
	01	1
	00	0
	01	1
	01	1
005	32	sin

\*Denotes 2nd function key

Explanation. If flag 1 is set, transfer is made to location 011, otherwise the program continues by taking the sin of the display-register contents.

Example:

LOC	CODE	KEY	LOC	CODE	KEY
000	46	*LBL	010	29	*x!
	11	A		46	*LBL
	90	*if zro		87	*1'
	87	*1'		01	1
	55	÷		46	*LBL
005	42	STO	015	29	*x!
	32	sin		42	STO
	95	=		00	0
	20	*1/x		01	1
	41	GTO		81	HLT

\*Denotes 2nd function key

Explanation. This is an entire program for computing  $y = \sin x/x$ . The test ( $x = 0?$ ) causes the execution to bypass much of the calculation if  $x = 0$ , arriving at label 1' (location 012). If  $x \neq 0$ , then first  $x/\sin x$  is formed. (Refer back to the chapter on memory registers for the dummy

use of the **STO** instruction.) Then we form  $(\sin x)/x$  and skip over the 1 (following label 1') to store the answer in  $R_{01}$  and halt.

Example:

LOC	CODE	KEY
000	80	*if pos
	85	+
	94	+/-
	46	*LBL
	85	+
005	30	* $\sqrt{x}$

\*Denotes 2nd function key

Explanation. If the display-register content is positive, the program skips over the change-sign instruction and forms the square root. In the other event, the absolute value of the negative quantity is formed, then the square root is taken.

From the basic branching instructions you can synthesize other branching instructions as shown in the table below.

Transfer When:	Instruction Sequence
$a = b$	( <input type="text"/> a <input type="text"/> - <input type="text"/> b <input type="text"/> ) <input type="text"/> 2nd <input type="text"/> if zro
$a \neq b$	( <input type="text"/> a <input type="text"/> - <input type="text"/> b <input type="text"/> ) <input type="text"/> INV <input type="text"/> 2nd <input type="text"/> if zro
$a > b$	( <input type="text"/> b <input type="text"/> - <input type="text"/> a <input type="text"/> ) <input type="text"/> INV <input type="text"/> 2nd <input type="text"/> if pos
$a \geq b$	( <input type="text"/> a <input type="text"/> - <input type="text"/> b <input type="text"/> ) <input type="text"/> 2nd <input type="text"/> if pos
$a < b$	( <input type="text"/> a <input type="text"/> - <input type="text"/> b <input type="text"/> ) <input type="text"/> INV <input type="text"/> 2nd <input type="text"/> if pos
$a \leq b$	( <input type="text"/> b <input type="text"/> - <input type="text"/> a <input type="text"/> ) <input type="text"/> 2nd <input type="text"/> if pos

Example: Suppose you wish to determine if the display-register content,  $x$ , is less in magnitude than the quantity  $\epsilon$  stored in  $R_{19}$ . If that is the case, you wish to recall  $R_{01}$  and halt execution; otherwise you wish to add  $x$  to  $R_{01}$  and go to location 101. This could be accomplished as follows:

LOC	CODE	KEY
000	65	X
	53	(
	40	*x <sup>2</sup>
	30	*√x
	75	—
005	43	RCL
	01	1
	09	9
	54	)
	22	INV
010	80	*if pos
	81	HLT
	01	1
	95	=
	44	SUM
015	00	0
	01	1
	41	GTO
	01	1
	00	0
020	01	1
	46	*LBL
	81	HLT
	43	RCL
	00	0
025	01	1
	81	HLT

Begin formation of quantity for testing. **X** stores x into internal processing register for safekeeping.

Forms  $|x|$ .

$(|x| - \epsilon)$ .

Transfer on  $|x| < \epsilon$ .  
(to point labeled HLT)

Overwrite  $|x| - \epsilon$  with 1.

Complete pending multiply of x:  
 $x \times 1 = x$ .

x added to R<sub>01</sub>

Transfer

Desired action when  $|x| < \epsilon$ .

\*Denotes 2nd function key

These examples were not chosen for their simplicity, so don't be discouraged if you have to study them in order to understand them. They are very realistic in the way branching instructions are used in actual programs, so as you master these examples you are learning not just what these instructions do, but also how they are used. One sure way to develop your competence in this area is through practice and actual use of these features to accomplish desired objectives in real programs.

## SETTING AND RESETTING PROGRAM FLAGS

The five program flags are initially set to zero when the calculator is first turned on. They may each be set (or reset) from the keyboard or as the result of the appropriate instructions executed during the running of a program. Whether from the keyboard or in a program instruction sequence, the flags are set by the command **2nd st flg**, followed by the single-digit ( **0** through **4** ) identification number of the flag affected. They are reset by means of the inverse instruction **INV 2nd st flg** followed by the identification number.

There are a number of uses of the program flags; three of them are the following:

- 1) Controlling program options from the keyboard prior to execution.
- 2) Effecting a delayed branch based upon a test more conveniently made earlier.
- 3) Keeping track of execution history – which path through the program has led to the present point?

To illustrate the first of these uses, assume that you wish to run a program which normally prints out (with the accessory printing unit) four or five different quantities. However, at times some of those quantities are of little interest, and you would prefer to shorten the run time by computing and printing only one of them. This could be done by placing the appropriate **if flag** instructions in the program code. If you wanted only the main output, you might set the appropriate flag; whereas if you desired all of the answers, you would have the flag reset. Or perhaps you would desire more flexibility and set flag 0 to compute and print variable  $x_0$ , set flag 1 to compute and print variable  $x_1$ , etc. That way if you wanted to see  $x_1$ ,  $x_3$ , and  $x_4$  you would set flags 1, 3, and 4 just prior to execution.

To illustrate the second use, suppose that you have reached a certain point in the code to solve a problem and what is to be done subsequently depends on whether the display-register contents are positive. If the results are positive, you wish to execute a sequence of fifteen instructions (which we will abbreviate [S15]) and then go to location 219. If the results are not positive you wish to execute the same set of fifteen instructions then not go to location 219 but rather continue without a transfer. Thus, the point of the branching is fifteen steps beyond the point of the desired test. This problem is conveniently handled using the program flags as the following sequence demonstrates:

LOC	CODE	KEY
000	80	*if pos
	85	+
	22	INV
	50	*st flg
	00	0
005	41	GTO
	78	*5'
	46	*LBL
	85	+

LOC	CODE	KEY
	50	*st flg
010	00	0
	46	*LBL
	78	*5'
	...	[S15]
	60	*if flg
	00	0
030	02	2
	01	1
	09	9
		etc.

\*Denotes 2nd function key

The third use gives you a means of "remembering" in the program execution whether you have arrived at a given crucial point by one path (path A) or another (path B). What you wish to do at this point depends on which path your program has taken. You recall that the program counter only knows where it is and has no recollection as to how it came there. Yet such recollective ability is sometimes needed, and the program flags do this admirably. One simply places a set-flag instruction in one path, and a reset-flag instruction in the other; and the execution history is recovered through an if-flag instruction wherever desired.

Finally, the **2nd rset** instruction resets all five flags as well as clears the subroutine return- pointer registers and positions the program counter to the top of the program memory (000).

## DECREMENT AND SKIP ON ZERO (DSZ)

A powerful instruction, especially useful in programming iterative routines, is the decrement-and-skip-on-zero command, **2nd** **dsz**. It is really several instructions in one. At this point, we assume that  $R_{00}$  contains an integer. (If this is not the case, the effect is as though  $R_{00}$  contains the next larger integer.) The effect of the DSZ is the following:

- 1) First decrement the magnitude of the quantity stored in  $R_{00}$  by 1.
- 2) If the resulting content of  $R_{00}$  is zero, do not transfer to the specified address or label, but skip or fall through to the next instruction.
- 3) If the resulting content of  $R_{00}$  is not zero, transfer to the specified location or label.

The DSZ instruction also has its inverse, obtained by the sequence **INV** **2nd** **dsz**. It functions in the same way except for reversing the test: If the value is not zero the transfer is omitted; the transfer is made only if the value is zero.

If the DSZ function and polar/rectangular conversion are both used in the same program, it will be necessary to temporarily store the contents of  $R_{00}$  in another memory register, perform the polar/rectangular conversion, and return the proper data for the DSZ function to  $R_{00}$ .



To illustrate the use of the DSZ instruction, consider the following simple examples.

Example: Using the DSZ instruction, compute the sum of all integers 1 through N. (Enter the value of N, clear memories, then press **A**.)

Solution:

LOC	CODE	KEY
000	46	*LBL
	11	A
	42	STO
	00	0
	00	0
005	46	*LBL
	44	SUM
	43	RCL
	00	0
	00	0

LOC	CODE	KEY
010	44	SUM
	00	0
	01	1
	58	*dsz
	44	SUM
015	43	RCL
	00	0
	01	1
	81	HLT

\*Denotes 2nd function key

Example: Design a program to compute the following sum by merely entering the number N of terms to be summed:

$$y = \sum_{k=1}^N \ln X_k$$

Solution: Using the DSZ, the following straightforward solution results.

LOC	CODE	KEY
000	46	*LBL
	11	A
	42	STO
	00	0
	00	0
005	00	0
	42	STO
	00	0
	01	1

LOC	CODE	KEY
	46	*LBL
010	85	+
	43	RCL
	00	0
	00	0
	23	lnx
015	44	SUM
	00	0
	01	1

LOC	CODE	KEY
	58	*dsz
	85	+
020	43	RCL
	00	0
	01	1
	81	HLT

\*Denotes 2nd function key

Like the other transfer instructions, the destination used in a DSZ instruction can be a label or a three-digit absolute address. Also, the DSZ performs the test on the value in R<sub>00</sub> without recalling it from memory. Thus, pending operations are completely unaffected.

## XI. SUBROUTINES

Subroutines give you the capability to define a subprocess or function by a sequence of code, and then to invoke that code almost as though it were a keyboard function simply by "calling" it (either by its name, that is its label, or by its starting address in program memory). Furthermore, the subprocess so defined may be called more than once from anywhere in the program and, upon completion of its purpose, control will pass back to the main (or "calling") program at the next instruction past the point of the call.

Such subprocesses which are invoked and then pass control back to the calling sequence of code are known as subroutines. In the SR-52, subroutines called by the main program may themselves call subroutines which return control to them just as though the first level of subroutine were a main routine. Finally, upon invoking all required second-level subroutines and completing their assigned tasks, the first-level subroutines will pass control back to the main program. This main program can then call additional subroutines until the problem is complete. Although this may sound complicated, the coding is actually simplified by this approach.

### CALLING A SUBROUTINE

There are three methods of calling a subroutine. One is to use **SBR** followed by the label name of the subroutine being called. Thus **SBR sin** calls the subroutine labeled **sin**; or **SBR 2nd 1** calls the subroutine labeled **2nd 1**.

Example: Suppose you need to evaluate the following polynomial for three different values of  $x$  and, then sum the three results for a final answer.

$$10x^5 - 7.11x^4 + 19x^3 + 11x^2 - 6 = ?$$

Solution: Assume  $x_1$  is stored in  $R_{01}$ ,  $x_2$  is stored in  $R_{02}$ , and  $x_3$  is stored in  $R_{03}$ . This problem is solved without duplicating the code to evaluate the fifth-degree polynomial by using a subroutine, which we will name **2nd 5**. The code for the subroutine might be defined essentially as you write the polynomial:

LOC	CODE	KEY	LOC	CODE	KEY
000	46	*LBL		43	RCL
	78	*5'		01	1
	53	(	025	09	9
	42	STO		45	$y^x$
	01	1		03	3
005	09	9		65	$\times$
	45	$y^x$		01	1
	05	5	030	09	9
	65	$\times$		85	+
	01	1		43	RCL
010	00	0		01	1
	75	-		09	9
	43	RCL	035	40	* $x^2$
	01	1		65	$\times$
	09	9		01	1
015	45	$y^x$		01	1
	04	4		75	-
	65	$\times$	040	06	6
	07	7		54	)
	93	.		56	*rtn
020	01	1			
	01	1			
	85	+			

\*Denotes 2nd function key

Notice in this subroutine code, that the equals key is not used. An open parenthesis at the beginning and close parenthesis at the end performs the necessary polynomial evaluation without completing all pending operations in the main routine, as would be done by an  $\boxed{=}$ .

To obtain the final result, you do not want to write this code sequence down three times, once using  $x_1$ , once using  $x_2$ , and once using  $x_3$ . The following main-program code calls the subroutine to evaluate the polynomial for each value of  $x$  and sums the results.

LOC	CODE	KEY	LOC	CODE	KEY
050	46	*LBL		51	SBR
	11	A		78	*5'
	43	RCL		85	+
	00	0		43	RCL
	01	1	065	00	0
055	51	SBR		03	3
	78	*5'		51	SBR
	85	+		78	*5'
	43	RCL		95	=
	00	0	070	81	HLT
060	02	2			

\*Denotes 2nd function key

In this main program, the subroutine is called by the sequence  $\boxed{\text{SBR}} \boxed{2\text{nd}} \boxed{5}$ , where  $\boxed{2\text{nd}} \boxed{5}$  is the label for the subroutine. In the subroutine, the last instruction  $\boxed{\text{rtn}}$  returns control to the calling program at the point of the call. The foregoing example could also have been solved with a DSZ instruction in combination with indirect addressing, an advanced technique discussed in a later section.

Example: Construct a main program to compute an expression  $y_1^{y_2} + (\ln y_3)(y_4 + e^{y_5})$ , where each of the quantities  $y_1$  through  $y_5$  is sufficiently complicated that its computation requires a subprogram.

Solution:

LOC	CODE	KEY	LOC	CODE	KEY
000	46	*LBL		65	×
	11	A		53	(
	51	SBR		51	SBR
	87	*1'		77	*4'
	45	$y^x$	015	85	+
005	51	SBR		51	SBR
	88	*2'		78	*5'
	85	+		22	INV
	51	SBR		23	lnx
	89	*3'	020	54	)
010	23	lnx		95	=
				81	HLT

\*Denotes 2nd function key

This solution uses the subroutines labeled **2nd** **1** through **2nd** **5** just as if would use the quantities  $y_1$  through  $y_5$  themselves. In other words, the main routine is programmed as one would normally write the algebra. Everywhere that **SBR** **2nd** **3** appears, the necessary subroutine provides the actual value of  $y_3$  to the main routine.

One of the advantages of the depth of the internal processing registers of the SR-52 is that even tri-level program structures can be supported without practical danger of having too few registers to hold all pending operations.

It is irrelevant where the subroutine is stored relative to the location of the calling program. In fact, except for the obscurity of program structure which results, a subroutine can actually be a portion of the calling routine. Taking this approach to the extreme, a program segment can actually call itself as a subroutine! But you should avoid even thinking about such recursive structures for now.

The second method for calling subroutines pertains only to subroutines labeled **A** through **E** and **2nd A** through **2nd E**. Subroutines with these labels may be called by just mentioning their names. That is, the **SBR** instruction can be omitted.

Example: Construct a main program which computes  $a + b^c$  by calling subroutines A, B, and C.

LOC	CODE	KEY
000	46	*LBL
	15	E
	11	A
	85	+
	12	B
005	45	$y^x$
	13	C
	95	=
	81	HLT

\*Denotes 2nd function key

The third method even enables you to call unlabeled sequences of code as subroutines. To do this, use the subroutine followed by a three-digit address of the first program location in the desired subroutine. Thus **SBR** 007 calls the subroutine beginning at location 007.

## LABELING A SUBROUTINE

You have just seen that a subroutine need not be labeled (unless you also desire to invoke it directly from the keyboard using the user-defined keys). It is a recommended procedure, however, because it adds to the clarity of the program code—particularly if the labels are chosen well and the program documentation records the meanings of each label. In addition, by labeling subroutines you can write the code which calls the subroutines before you know where those subroutines will be stored. This also means that adding or deleting instructions after the program is stored will not require changing the location address of the subroutine calling instructions. The rules for labeling subroutines are identical in every respect to the rules for labeling any program segment.

## AVOID USING $\equiv$ IN SUBROUTINES

One should use  $\equiv$  (equals instruction) only with great discretion in subroutines, and preferably not at all. The reason is that the equals instruction completes all pending operations. Some of these pending operations may be created in the calling routine; so that an equals instruction in the subroutine would complete these operations improperly. As an illustration, in the previous example, we easily created a program to compute  $a + b^c$  with calls to subroutines A, B, and C. We did not bother to define the subroutines in the example. Now imagine that the code for subroutine B contains an equals instruction. Specifically, suppose that it has one such instruction just before the return instruction. The result would be to complete the pending addition with a; so that we would have  $(a + b)$  at that point. Whatever then occurred in subroutine C, would produce the wrong answer. [If subroutine C contained no equals instructions, we would in fact obtain  $(a + b)^c$  as the final result of the calculation.]



Avoiding the equals instruction in such cases should impose no hardship, for you learned in Section III that enclosing an expression in parentheses is sufficient to evaluate it. Accordingly, well written subroutines will often begin with a `(` and end with a `)` just before the return instruction.

Whenever the subroutine requires repeated access to `x` (the display register contents at the time of the call), the subroutine may include a store instruction prior to performing arithmetic. If `x` is only needed to begin the subroutine computation, a dummy memory operation is often convenient. This last situation often leads to subroutines beginning with sequences, such as:

`2nd` `LBL` `C` `(` `RCL` `+` `2nd` `1/x` ...etc.

## THE RETURN INSTRUCTION

The last instruction in a subroutine, the one which returns control to the calling program is always a return instruction `rtn`. We have already described what occurs when a return instruction is encountered: A transfer is effected to the first instruction after the point of the call in the calling program.

There are only two subroutine return-pointer registers. The first register holds the address for return to a main routine when a first-level subroutine is called. The second register holds the address to which a second-level subroutine should return when called by a first-level subroutine. If a third-level subroutine is called by a second-level subroutine, when `rtn` is encountered in the third-level subroutine, control will be passed back to the first level subroutine (not to the second). The section on *Indirect Instructions* shows you how to extend beyond tri-level programs.

Example: To see how this works consider the following code:

LOC	CODE	KEY	LOC	CODE	KEY
000	46	*LBL		46	*LBL
	11	A		13	C
	12	B		03	3
	01	1		42	STO
	42	STO	020	00	0
005	00	0		03	3
	01	1		56	*rtn
	81	HLT			
	46	*LBL			
	12	B			
010	13	C			
	02	2			
	42	STO			
	00	0			
	02	2			
015	56	*rtn			

\*Denotes 2nd function key

These instructions will produce the following when executed by pressing **A**, the main routine.

- 1) Subroutine B is called at location 002.
- 2) Subroutine B calls C at location 010.
- 3) Subroutine C stores 3 in  $R_{03}$ .
- 4) Subroutine C executes a return instruction, passing control back to B (first-level subroutine) at the point just beyond where B called C.
- 5) Subroutine B stores 2 into  $R_{02}$ .

- 6) Subroutine B executes a return instruction, passing control back to the main routine just past the point-of-call.
- 7) The main routine stores 1 into R<sub>01</sub> and halts.

If you execute this code and then recall the contents of R<sub>01</sub>, R<sub>02</sub>, and R<sub>03</sub>, you will find that everything worked as it was supposed to.

Now, leaving routine A and subroutine B defined as before, change the rest of the code by adding another subroutine level:

LOC	CODE	KEY	LOC	CODE	KEY
	46	*LBL		46	*LBL
	13	C	025	14	D
	14	D		04	4
	03	3		42	STO
020	42	STO		00	0
	00	0		04	4
	03	3	030	56	*rtn
	56	*rtn			

\*Denotes 2nd function key

If you execute this code and then recall the register contents (clearing the registers from the last example first) you will find that the sequence **3** **STO** **03** never took place. This is an example of what we discussed before. When C called D, there was no place left to store a return address for D to use for its return. The fourth level of routine nesting exceeded the three levels automatically available on the SR-52. Consequently, when the return instruction was executed in D, the point of return was back in subroutine B just as before; not to the **3** **STO** **03** in subroutine C.

The following rule makes it easy to use subroutines as main routines. If a return instruction is encountered when there is in fact no calling program awaiting return of control, then a halt occurs, control passes back to the keyboard, and the program counter resides at the first location following the return instruction.

Occasionally programs are designed so that completion of execution may occur inside a first-level (or even second-level) subroutine. In other words, the answer to the problem (or perhaps detection of an error in the problem) has been obtained without returning control to the calling routine(s). In such situations return-of-control remains pending, the subroutine return-pointer registers are left with the pointers back to the unsatisfied return point(s) in the calling routine(s). Unless the calculator is turned off, the very next time a return instruction is encountered in a new problem any pending return left over from the last problem will be satisfied. This would rarely be the intended effect, however; and an improper execution would result. To prevent such left-over return pointers from ruining proper execution of the next problem, the reset instruction ( **2nd** **rsl** ) should be used to reset the return-pointer registers. This may be done manually, but it is preferable, when possible, to include the reset command at the proper point in the program code, with a halt instruction at location 000.

## **SUBROUTINE PRACTICE PROBLEMS**

We conclude this section with two examples of subroutine usage which for different reasons may prove instructive and useful.

Example: Construct a subroutine to take the integer part of  $x$  and place the fractional part in  $R_{19}$ . It should be so designed that it may be used without disturbing pending operations in the calling routine.

Solution: We will construct a solution for  $x > 0$  and let you work out the more general case as an exercise.

LOC	CODE	KEY	LOC	CODE	KEY
000	46	*LBL	010	57	*fix
	15	E		00	0
	53	(		52	EE
	42	STO		22	INV
	01	1		44	sum
005	09	9	015	01	1
	75	—		09	9
	93	.		56	*rtn
	05	5			
	54	)			

\*Denotes 2nd function key

You will notice that this subroutine does leave the display in scientific fix 0 format.

Example: Write a program to solve for  $F(x) = 0$ , where  $F(x)$  is a subroutine defined function provided by the user.

Solution: We may use Newton's method of solution. (This well-known method will not necessarily converge for all problems.) This method performs the iteration  $x_{n+1} = x_n - F(x_n)/F'(x_n)$ , where  $F'(x_n)$  is an estimate of the derivative of  $F$  at  $x_n$ . The derivative will be estimated by finite differences:

$$F'(x_n) \cong \frac{F(x_n + \delta) - F(x_n - \delta)}{2\delta}$$

Preserving all data registers but  $R_{18}$  and  $R_{19}$  for you to use in the definition of  $F(x)$ , we produce the following main routine which you should find not only instructive as to the use of subroutines but useful in your problem solving as well.

LOC	CODE	KEY
000	46	*LBL
	11	A
	42	STO
	01	1
	08	8
005	81	HLT
	46	*LBL
	12	B
	42	STO
	01	1
010	09	9
	46	*LBL
	17	*B'
	53	(
	43	RCL
015	01	1
	09	9
	15	E
	55	÷
	53	(
020	53	(
	43	RCL
	01	1
	09	9
	65	×
025	53	(
	01	1
	75	-

LOC	CODE	KEY
	52	EE
	06	6
030	94	+/-
	54	)
	54	)
	15	E
	75	-
035	53	(
	43	RCL
	01	1
	09	9
	65	×
040	53	(
	01	1
	85	+
	52	EE
	06	6
045	94	+/-
	54	)
	54	)
	15	E
	54	)
050	65	×
	43	RCL
	01	1
	09	9
	65	×
055	02	2

LOC	CODE	KEY
	52	EE
	06	6
	94	+/-
	54	)
060	44	SUM
	01	1
	09	9
	80	*if pos
	85	+
065	94	+/-
	46	*LBL
	85	+
	75	-
	43	RCL
070	01	1
	08	8
	95	=
	80	*if pos
	17	*B'
075	43	RCL
	01	1
	09	9
	81	HLT
	46	*LBL
080	15	E

\*Denotes 2nd function key

The instructions for using this program are as follows (assume the foregoing code is on a magnetic card):

- 1) Load the program (side A) into program memory.
- 2) Press **GTO** **E** .
- 3) Go into learn mode (press **LRN** ).
- 4) Key in the definition of your function  $F(x)$  assuming  $x$  is in the display register to begin with and  $F(x)$  must be left in the display register at the end. You may not use **=** in this subroutine, but all data registers except  $R_{18}$  and  $R_{19}$  are available.
- 5) End your subroutine with a **rtn** instruction.
- 6) Press **LRN** to go into calculate mode.
- 7) Enter desired accuracy of  $x$  and press **A** .
- 8) Enter guess of solution and press **B** .
- 9) Answer will appear within seconds or minutes, depending upon accuracy desired, first guess, and nature of  $F(x)$ .

Because the main program only uses 79 steps, you may use 145 steps (including the label and return) to define  $F(x)$ .

Using this example, suppose you wished to solve for  $x$  such that  $\ln x = .2 x$ . You might define  $F(x) = .2 x - \ln x$ . The subroutine at E could then look like this:

LOC	CODE	KEY	LOC	CODE	KEY
	46	*LBL		65	×
080	15	E		93	.
	42	STO	090	02	2
	00	0		75	—
	01	1		43	RCL
	53	(		00	0
085	43	RCL		01	1
	00	0	095	23	lnx
	01	1		54	)
				56	*rtn

\*Denotes 2nd function key

The answer to this problem is  $x \approx 1.296$ .



## XII. INDIRECT INSTRUCTIONS

Every transfer instruction which specifies a location in program memory or addressable register operation which references data registers through the two-digit register names has a counterpart instruction, the so-called indirect form. These indirect instructions add such flexibility in programming that new applications for them will be continually found. They represent a sophisticated programming tool which many SR-52 owners can use to accomplish processing which they simply could not otherwise accomplish.

First we will discuss the indirect instructions relating to data register operations and then those which relate to program transfer instructions.

### INDIRECT DATA-REGISTER INSTRUCTIONS

There are seven basic data-register instructions: **STO**, **RCL**, **EXC**, **SUM**, **PROD**, **INV SUM**, and **INV PROD**. They all have one thing in common: In the instruction sequence which uses them, a two-digit number (00 through 19) must appear in the sequence just after each one to designate the data register affected. The sequence **RCL** 04 would recall the value in  $R_{04}$ .

An indirect instruction is formed by preceding the normal direct instruction by **2nd** **IND**. What would be the effect of **2nd** **IND** **RCL** 04? This instruction will recall the value not in data register 04 but rather the value in the data register named by the contents of  $R_{04}$ . For example if the value stored in  $R_{04}$  were 11 then **2nd** **IND** **RCL** 04 would recall the value stored in  $R_{11}$ .

Whenever a value stored in a data register is used not just as another number but as an address (in this case a data register number, hence a data register address) it is called a pointer. One says that the pointer "points to" the data register having as its address the value of the pointer. In all indirect instructions, the content of the data register directly designated in the instruction is used as a pointer.

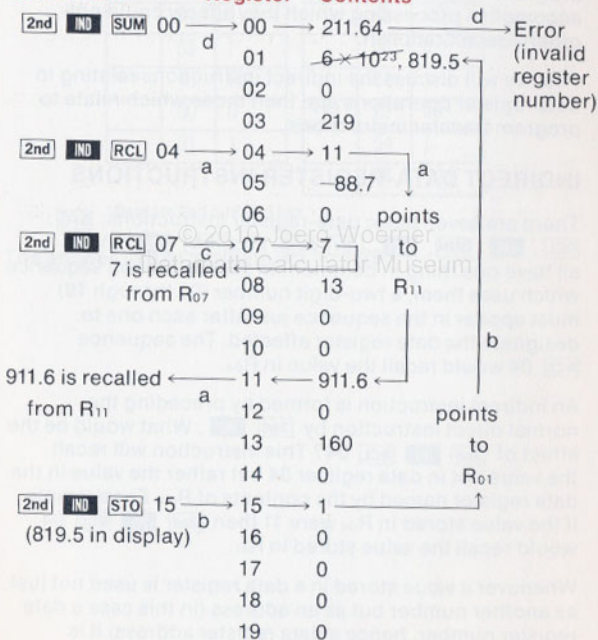
In the above example **2nd** **IND** **RCL** 04 means: "Use the contents of R<sub>04</sub> as a pointer to the data register whose contents are to be recalled." Similarly, **2nd** **IND** **STO** 15 would mean: "Use the contents of R<sub>15</sub> as a pointer to the data register into which to store the display-register value."

The diagram below illustrates these concepts graphically.

### Indirect

#### Instruction

#### Register Contents



This diagram shows the effect of **2nd** **IND** **RCL** 04 as the path of events marked a: The result is to recall the value  $R_{11} = 911.6$ . The effect of **2nd** **IND** **STO** 15 is shown as the path of events marked b: The result is to store 819.5 into  $R_{01}$ , formerly containing  $6 \times 10^{23}$ . Finally, the result of **2nd** **IND** **RCL** 07 in this example is marked c: The pointer points back to data register 07 so that the result is to recall the value  $R_{07} = 7$ .

It is implied that any pointer used as the consequence of an indirect instruction must point to a realizable address. Accordingly, the result of the **2nd** **IND** **SUM** 00 in the example diagram would be an error — there is no data register 211.64.

Now we will practice the use of these indirect instructions in some example problems:

Example: You would like to enter a varying number of data items, up to a total of 19, and have them stored successively in data registers  $R_{01}$ ,  $R_{02}$ , etc., with the total number of items entered, counted and stored in  $R_{00}$ . Design a program to allow you to enter each item just by pressing **A**.

Solution: The indirect instructions make this an easy problem. The program is given by:

LOC	CODE	KEY
000	46	*LBL
	11	A
	36	*IND
	42	STO
	00	0
005	00	0
	01	1
	44	SUM
	00	0
	00	0
010	56	*rtn

\*Denotes 2nd function key

To run the program, you initialize 1 **[STO]** 00. The program ends on a return instruction, rather than a halt, because it is just the sort of thing one might like to have as a subroutine. Now try to solve the same problem without using an indirect instruction. (No solution supplied!)

Example: Five quantities,  $X_1, X_2, X_3, X_4,$  and  $X_5$  have been computed and stored (in order) in data registers  $R_{01}$  through  $R_{05}$ . Five other quantities  $Y_1$  through  $Y_5$  have been similarly stored in  $R_{06}$  through  $R_{10}$ . You wish to create a short segment of code in your program which will compute the average value of the five quantities  $Z_k = X_k/Y_k$  ( $k = 1, 2, \dots, 5$ ).

Solution:

LOC	CODE	KEY	LOC	CODE	KEY
000	05	5		44	SUM
	42	STO		01	1
	00	0		01	1
	00	0		01	1
	01	1	030	22	INV
005	00	0		44	SUM
	42	STO		01	1
	01	1		09	9
	09	9		58	*dsz
	00	0	035	87	*1'
010	42	STO		53	(
	01	1		43	RCL
	01	1		01	1
	46	*LBL		01	1
	87	*1'	040	55	÷
015	53	(		05	5
	36	*IND		54	)
	43	RCL			
	00	0			
	00	0			
020	55	÷			
	36	*IND			
	43	RCL			
	01	1			
	09	9			
025	54	)			

\*Denotes 2nd function key



Solution: First write the main routine, assigning R<sub>19</sub> to contain the bin number into which the current value of data contributes a count:

**Main Routine:**

LOC	CODE	KEY	
000	46	*LBL	
	15	E	
	13	C	} Find BIN #
	42	STO	
	01	1	
005	09	9	} Store that # in R <sub>19</sub>
	01	1	
	36	*IND	
	44	SUM	
	01	1	} Increment count in that bin (register) by 1
010	09	9	
	81	HLT	

\*Denotes 2nd function key

The next step is to define subroutine C which finds the proper bin number.

The proper bin number for the value x is given by the equation

$$n = 1 + \text{INT} [(x - a)/(b - a) \times 10],$$

where INT (y) = "integer part of y". Allocating R<sub>18</sub> and R<sub>17</sub> to a and b respectively, one may write the following without difficulty.

## Bin Number Subroutine:

LOC	CODE	KEY
	46	*LBL
	13	C
	53	(
015	53	(
	53	(
	42	STO
	75	—
	43	RCL
020	01	1
	08	8
	54	)
	55	÷
	53	(
025	43	RCL
	01	1
	07	7

LOC	CODE	KEY
	75	—
	43	RCL
030	01	1
	08	8
	54	)
	65	×
	01	1
035	00	0
	54	)
	14	D
	85	+
	01	1
040	54	)
	56	*rtn

\*Denotes 2nd function key



By now you should be used to the dummy store instruction used to provide a first-argument following the opening of a left parenthesis. But what was done about the INT function? We simply gave it the name "D" and deferred its coding until now. The value of the argument of the integer function is greater than zero, so we may use the integer subroutine given as an example in the last section.

Fixing up the resulting display format and discarding unnecessary features from the last section in the following version produces the following code.

### Integer-Value Subroutine:

LOC	CODE	KEY	LOC	CODE	KEY
	46	*LBL	050	57	*fix
	14	D		00	0
	53	(		52	EE
045	42	STO		22	INV
	75	—		52	EE
	93	.	055	22	INV
	05	5		57	*fix
	54	)		56	*rtn

\*Denotes 2nd function key

There are only two more details to complete and the problem is solved: We should make it convenient to enter the values of a and b and to initialize  $R_{01}$  through  $R_{10}$  to zero. Because it is natural to enter a at **A** and b at **B** we write:

LOC	CODE	KEY
	46	*LBL
	11	A
060	47	*CMs
	42	STO
	01	1
	08	8
	81	HLT

LOC	CODE	KEY
065	46	*LBL
	12	B
	42	STO
	01	1
	07	7
070	81	HLT

\*Denotes 2nd function key

and the problem is solved. Furthermore, the top-down method allowed you to think of one thing at a time rather than immediately being embroiled in details.

Note that it was necessary to write the subroutines so as not to disrupt any pending operations: The required effect of the subroutines was to replace x in the display register with  $f(x)$  without affecting anything pending. But you should always write your subroutines that way anyway; in that manner your programs will be "safe."

This problem illustrates much in addition to the use of the **2nd** **IND** **SUM** used to tally the counts in the ten "bin-registers." We hope you enjoyed working through it.

Next, we cite a fairly short and simple example to show the rapid increase in the complexity of the processing logic which happens when one combines indirect instructions in an instruction sequence.

Example: Consider the code sequence given by:

LOC	CODE	KEY
000	36	*IND
	43	RCL
	00	0
	01	1
	42	STO
005	01	1
	09	9
	36	*IND
	43	RCL
	01	1
010	09	9

\*Denotes 2nd function key

What is the effect of this short instruction sequence? Imagine that the number 15 is stored in  $R_{01}$ , that the number 4 is stored in  $R_{15}$  and that the number  $4.818 \times 10^{-10}$  is stored in  $R_{04}$ . The first indirect recall instruction automatically establishes that the pointer in  $R_{01}$  points to  $R_{15}$  and recalls its content, the integer 4. This value is placed in  $R_{19}$ . The next indirect recall instruction then uses the pointer in  $R_{19}$  to recall the contents of  $R_{04}$ , namely  $4.818 \times 10^{-10}$ . The overall effect of this sequence is to produce a second-level indirect recall. That is, the effect is to find the pointer in the register first named in the sequence ( $R_{01}$ ), use this pointer to find the location of the next pointer ( $R_{15}$ ), and finally use the pointer found there to point to and bring the actual number recalled to the display ( $4.818 \times 10^{-10}$  from  $R_{04}$ ). It is invalid to write the sequence:

**2nd** **IND**

**2nd** **IND**

**RCL** 01

However, the example shows that you can concisely synthesize instruction sequences which have that intended effect.

## INDIRECT PROGRAM-TRANSFER INSTRUCTIONS

You have seen how that preceding normal memory operation with **2nd** **IND** turns that instruction into an indirect instruction. The two-digit number specified in the indirect instruction is the register containing not the needed value itself, but a pointer to where that number is to be found.

In a similar manner, all of the following instructions may be converted to the indirect form:

<b>GTO</b>				<b>SBR</b>			
<b>2nd</b>	<b>if flg</b>	<b>n</b>		<b>INV</b>	<b>2nd</b>	<b>if flg</b>	<b>n</b>
<b>2nd</b>	<b>if err</b>			<b>INV</b>	<b>2nd</b>	<b>if err</b>	
<b>2nd</b>	<b>if zro</b>			<b>INV</b>	<b>2nd</b>	<b>if zro</b>	
<b>2nd</b>	<b>if pos</b>			<b>INV</b>	<b>2nd</b>	<b>if pos</b>	
<b>2nd</b>	<b>dsz</b>			<b>INV</b>	<b>2nd</b>	<b>dsz</b>	

When any of these instructions are preceded by **2nd** **IND**, the absolute address to which the transfer should be made is to be found in the data register designated by the two-digit number in the instruction sequence.

Example: Suppose memory register  $R_{14}$  contains the integer value 168: Then the sequence **2nd** **IND** **GTO** 14 would cause unconditional transfer to location 168.

Example: Suppose memory register  $R_{01}$  contains the value 7: Then the sequence **2nd** **IND** **INV** **2nd** **if flg** 3 01 would cause transfer to location 007 if flag 3 is not set.

Example: Let memory register  $R_{10}$  contain the quantity 21: Then the sequence **2nd** **IND** **2nd** **dsz** 10 would cause a decrement of  $R_{00}$ , transfer to location 021 if the result is non-zero, and a skip (of the transfer) to the next instruction if the result is zero.

Example: The sequence **2nd** **IND** **SBR** 08 would call the subroutine beginning at the location pointed to by the content of  $R_{08}$ .

There are two things to note about all the indirect transfer instructions. They all eventually reach the destination address through the absolute location (000 through 223), and never by means of a label. One cannot store a label as such in the data registers; but one can store a pointer to the destination program memory location. A number stored in a data register for indirect addressing need not be entered with leading zeros and is the only exception to the three-digit requirement for specifying program locations. The indirect transfer specification requires one less digit in the sequence than the direct form. This results from the fact that two digits are required in the instruction sequence to specify the data register rather than the three digits necessary to specify the absolute transfer address.

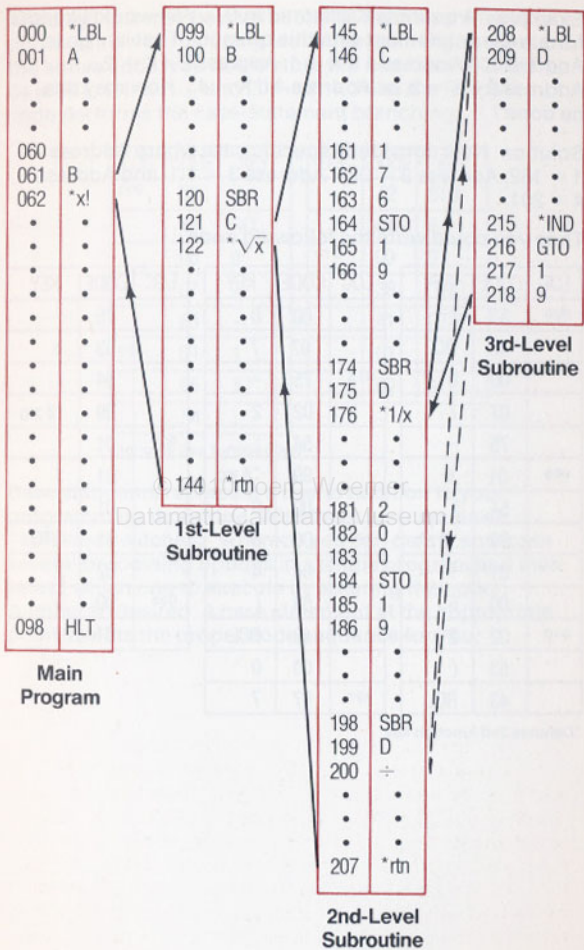
Note further that all indirect instructions whose direct counterparts can be executed from the keyboard may also be so executed.

We conclude this section with two practical examples of the use of indirect transfer instructions.

Example: Suppose you are designing a program from the top down as described before and discover that at several places you would like to define and call a certain subroutine. (We will name this subroutine D.)

Unfortunately, your problem is sufficiently "deep" that subroutine D would be a third-level subroutine (that is, a fourth level routine). You know that the SR-52 does not keep track of more than two return addresses. What do you do?

Solution: The approach is easy. You call subroutine D in the usual way with `SBR D`. However, instead of ending subroutine D with a `2nd rtn` statement, you end it with `2nd IND GTO 19`. Whereas the SR-52 provides only two dedicated registers for subroutine pointers, you simply appropriate register R<sub>19</sub> for this purpose. Oh yes, there is one thing more: Unlike the automatic processing of the first two return pointers, your code must store the return address into R<sub>19</sub> at convenient points prior to the actual subroutine calls. The result of such an approach is graphically shown in the following figure.



Example: A quantity K is stored in B<sub>07</sub>. You would like to form a case statement with this quantity: That is, "go to Address K – Address 1 if K = 1, Address 2 if K = 2, Address 3 if K = 3, or Address 4 if K = 4." How may this be done?

Solution: Now consider a specific case where Address 1 = 162, Address 2 = 064, Address 3 = 111, and Address 4 = 201.

Then you could write the following code

LOC	CODE	KEY
000	53	(
	43	RCL
	00	0
	07	7
	75	—
005	01	1
	54	)
	99	*if zro
	01	1
	06	6
010	02	2
	53	(
	43	RCL

LOC	CODE	KEY
	00	0
	07	7
015	75	—
	02	2
	54	)
	99	*if zro
	00	0
020	06	6
	04	4
	53	(
	43	RCL
	00	0
025	07	7

LOC	CODE	KEY
	75	—
	03	3
	54	)
	99	*if zro
030	01	1
	01	1
	01	1
	41	GTO
	02	2
035	00	0
	01	1

\*Denotes 2nd function key



Although this may be a solution it is not as concise as one available to you using indirect branching. Instead of the above code, you could store the numbers 162 in R<sub>16</sub>, 64 in R<sub>17</sub>, 111 in R<sub>18</sub>, and 201 in R<sub>19</sub>. Then the following code performs the case-statement branching:

LOC	CODE	KEY	LOC	CODE	KEY
000	53	(		42	STO
	43	RCL		01	1
	00	0	010	00	0
	07	7		36	*IND
	85	+		41	GTO
005	01	1		01	1
	05	5		00	0
	54	)			

\*Denotes 2nd function key

Case statements represent a useful addition to your programming repertoire. With them you can make "software switches," whereby you can define and code several processing options in a single program and then select which one to execute by entering the option number K, desired. A case statement at the appropriate point selects the proper code sequence for you.

### XIII. PRINTER CONTROL

The optional desk printing unit available for the SR-52 allows you to perform a number of different printing functions. You can:

- 1) List your program code in its entirety with a single command;
- 2) Print any results obtained while using the SR-52 in the calculate mode;
- 3) Insert print instructions in your program to print one or more results without halting program execution;
- 4) Perform paper spacing either from the keyboard or under program control in order to set off sets of results;
- 5) Place the printer in the TRACE mode to automatically keep a record of all calculations performed both manually and in the run mode. This trace includes a record of the operations performed as well as the results.
- 6) Leave your desk without having to lock your SR-52 away. The printing unit provides security as well as power for your calculator.

## LISTING A PROGRAM

To list a program (from the point currently indicated by the program counter) simply press **2nd list** while in the calculate mode. The program will be listed from that point and then automatically halt the listing after the last instruction at location 223 has been printed. The listing may also be halted manually at any time by pressing **HLT**. For a complete program listing, the initial position of the program counter may conveniently be positioned by pressing **2nd rset**. The list instruction is of course executed in the calculate mode.

The actual program listing obtained is in the following format: The first three digits show the location in program memory and the alphanumeric symbols to the right show the instruction stored there.

The chief benefits of performing a program listing are:

- 1) To verify that the instructions keyed in, correctly match those intended, as written on the coding form.
- 2) To provide quick documentation for a program either hastily constructed without benefit of careful documentation or else changed since that time.
- 3) To provide a means of later verifying that results obtained in running a program were based on a correct problem formulation.

## PRINTING DATA

From the keyboard, the contents of the display register may be printed at any time by pressing **2nd** **prt**. The same instruction encountered in the program code will cause that action to take place in the run mode. To illustrate this, consider the following program example.

Example:

LOC	CODE	KEY	LOC	CODE	KEY
000	46	*LBL		54	)
	11	A		98	*prt
	53	(		85	+
	43	RCL		43	RCL
	00	0	015	00	0
005	01	1		00	0
	55	÷		98	*prt
	43	RCL		95	=
	00	0		98	*prt
	02	2	020	81	HLT
010	98	*prt			

\*Denotes 2nd function key

In this example the following quantities are consecutively printed before the program ends:

- 1)  $\bullet R_{02}$
- 2)  $(\bullet R_{01}/\bullet R_{02})$
- 3)  $\bullet R_{00}$
- 4)  $\bullet R_{00} + (\bullet R_{01}/\bullet R_{02})$ .

To see all those quantities without a printer would require program halts and manual resumptions.

## PAPER ADVANCEMENT

The paper may be advanced in either mode by the command (or instruction) **2nd** **pap**. This feature is particularly useful for separating data. For example, you might be interested in making up a depreciation schedule with your SR-52. For this purpose you might group the data: giving year, book value, depreciated amount, depreciation to date, and reserve as a set of five consecutive values set off from other years' data by a space.

## PROGRAMMING IMPLICATIONS

You have seen that the printer can print data without halting programming execution. This gives you all the capabilities of a PAUSE instruction and more. Knowing that you are going to run a program with a printer could influence the way you design that program. In particular:

- 1) You may delete halts for observing several results;
- 2) You may print successive iterations of a repetitive calculation to see whether the result is converging or diverging and when the calculation may be halted (rather than using predefined accuracy criteria and a conditional transfer operation);
- 3) You may monitor where the program execution is currently taking place through printing cues or intermediate data. Timing information can also be obtained this way so that you can find out which portions of your program are requiring the most time.

© 2010 Joerg Woerner

## TRACE OPERATION Calculator Museum

The trace mode is the one print function which is commanded from the printer rather than from the SR-52. It is particularly useful in tracing manual computations, providing "hard copy" of the results, and in "debugging" programs. Additional details of operation are provided in the instructions supplied with the printer.

## RUNNING LONG PROGRAMS

The optional printer also serves as a security cradle. In combination with the features already discussed, this implies that you can use the SR-52 to run programs that require a long execution time without being "tied down" to your desk. The SR-52 will execute the program, the printer will print all answers, and your SR-52 will be secure even while you are absent.

## XIV. A COMPLETE SAMPLE PROGRAM

This basic program is designed to show the fundamental steps in creating, programming and running your own programs.

### Define the Problem

Suppose that the monthly service charge on your checking account is calculated as follows:

\$0.10 per check for the first five

\$0.09 per check for the next five

\$0.08 per check for the next five

\$0.07 per check for each check over fifteen

The problem is to construct a program that will compute your monthly service charge for a given number ( $n$ ) of checks.

Looking at the situation carefully, you can see that there are three conditions to test where  $n$  is the number of checks.

1. Is  $n > 5$  or  $5 - n \geq 0$ ?
2. Is  $n > 10$  or  $10 - n \geq 0$ ?
3. Is  $n > 15$  or  $15 - n \geq 0$ ?

Now the truth or falseness of these conditions will determine the action to be taken. The possibilities are summarized by the flow diagram in Figure 2.

### Develop a Flow Diagram

The flow diagram in Figure 2 provides a full visual representation of how the identified problem can be solved within the functional capabilities of the calculator. When first preparing a flow diagram, only the bare essentials necessary for clarity should be included. The

simple format selected for Figure 2 is: circles for identifying primary labels (or locations), rectangular boxes to contain mathematical functions which logically fit together, diamonds to show conditional transfer instructions, and squares to show program termination.

Assume n is  
in display

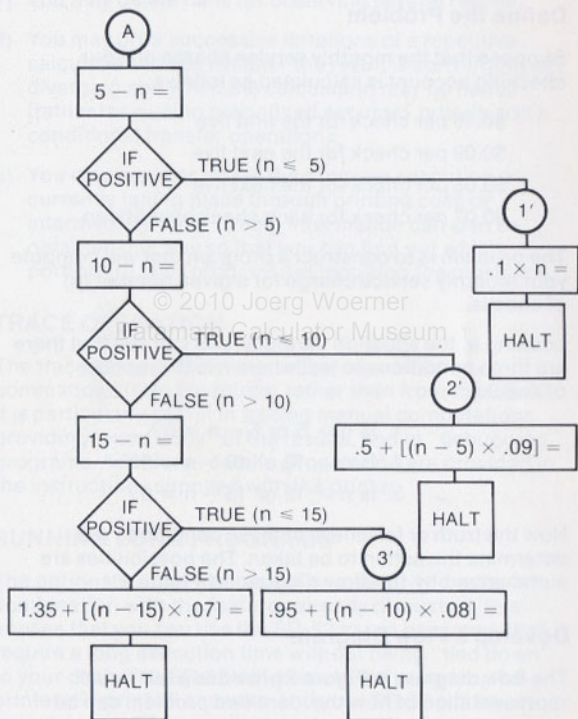


Figure 2. Sample Program Basic Flow Diagram



As indicated on the diagram, the number of checks ( $n$ ) is assumed to be in the display when starting the program. Label A is identified to show that user-defined key A is to be used to start the program.

The first operation to perform is to determine if the value of  $n$  is five or smaller. By subtracting  $n$  from five, an if-positive function makes the decision... if  $5 - n \geq 0$ . If true, then  $n$  is five or less and the program transfers to label 1'. Label 1' is the second function of the 1 key and was chosen as a label because it is convenient to remember. If the program operation reaches label 1', the value of  $n$  is five or less. Therefore, the service charge is calculated by  $\$0.10 \times n$  and the program is stopped. If the result of  $5 - n$  is negative ( $n > 5$ ), the false response of the if-positive function continues with the main program and ignores label 1'.

Now that  $n$  is known to be more than five, it must be determined if  $n$  is ten or smaller. The second if-positive function makes the decision... if  $10 - n \geq 0$ . If true,  $n$  is ten or less and the program transfers to label 2'. When the program reaches label 2', the value of  $n$  is known to be between five and ten. The cost of the first five checks is a fixed value...  $\$0.10 \times 5 = 50$  cents. The cost of checks six through ten is nine cents each...  $(n - 5) \times \$0.09$ . Adding on the 50 cents (.5) completes the calculation and the program is stopped. If the result of  $10 - n$  is negative ( $n > 10$ ), the false response of the second if-positive function continues with the main program and ignores label 2'.

The value of  $n$  is now known to be greater than ten and it must be determined if  $n$  is 15 or smaller. By subtracting  $n$  from 15, a third if-positive function makes the decision ... if  $15 - n \geq 0$ . If true, then  $n$  is 15 or less and the program transfers to label 3'. When the program reaches label 3',  $n$  is known to be between 10 and 15. The cost of the first 10 checks is fixed:  $(\$0.10 \times 5) + (\$0.09 \times 5) = 95$  cents. The cost of more than 10 checks is calculated by  $(n - 10) \times \$0.08$ . Adding on the 95 cents (.95) completes the calculation and the program is stopped. If the result of  $15 - n$  is negative ( $n > 15$ ), the false response of the third if-positive function continues with the main program and ignores label 3'.

When it is known that  $n$  is greater than 15, the cost of the first 15 checks becomes a fixed value:  $(\$0.10 \times 5) + (\$0.09 \times 5) + (\$0.08 \times 5) = \$1.35$ . The cost of more than 15 checks is calculated by  $(n - 15) \times \$0.07$  and after adding on the \$1.35, the program is stopped.

## Convert Flow Diagram to Keystrokes

Before continuing with the actual program, some help may be needed to translate the flow diagram into program steps. Most programs have what may be referred to as a main flow of program steps. The main flow in Figure 2 is obvious by the column of instructions in a column under label A. The convenience in identifying the main flow is that all program steps for it may be written in order on the coding form, with the location numbers in sequential order. For example, the main flow of the program in

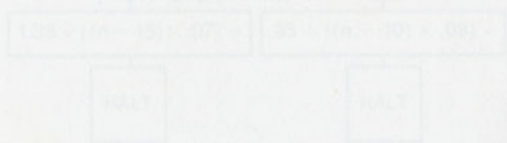


Figure 2. Sample Program Basic Flow Diagram

Figure 2 is written in locations 000 through 041 (Figure 3a through 3h). The program steps for the labels (1', 2', 3'), which may be called from the main flow program, are then written on the coding form following the main flow steps. Generally, the program steps for the labels are written in the same order they are encountered on the flow diagram. For example: locations 042 through 049 (Figure 3e) — label 1', 050 through 065 (Figure 3f) — label 2', and 066 through 083 (Figure 3g) — label 3'. This does not mean, however, that a label cannot be included in the main flow of a program.

To correlate actual program steps with the flow diagram in Figure 2, a program listing is shown in segments by Figure 3a through 3h. Each item on the flow diagram is keyed to the actual related program steps or locations. Notice that some programs steps do not relate to the flow diagram. These steps, sometimes referred to as housekeeping steps, are necessary to control data and program instructions. For example, Figure 3a shows that locations 002, 003 and 004 were not represented on the flow diagram. These three steps store the displayed number ( $n$ ) in memory register 19 ( $R_{19}$ ). It is now obvious on the flow diagram that  $n$  needs to be available for use several times during the program and that temporary storage is required.  $R_{19}$  was chosen for use in the program since small numbered memory registers are frequently used with keyboard calculations. Each time the value  $n$  is required in the program, you now could use **RCL** 19. However, in Figure 3a you see the operation  $5 - n =$  is directly written as  $5 - E =$ .

LOC	CODE	KEY
000	46	*LBL
	11	A
	42	STO
	01	1
	09	9
005	05	5
	75	—
	15	E
	95	=
	80	*if pos
010	87	*1'

} (A)

} 5 - n =

} IF POSITIVE

Figure 3(a)

LOC	CODE	KEY
	01	1
	00	0
	75	—
	15	E
015	95	=
	80	*if pos
	88	*2'

} 10 - n =

} IF POSITIVE

\*Denotes 2nd function key

Figure 3(b)

Figure 3. Program Code

LOC	CODE	KEY
	01	1
	05	5
020	75	—
	15	E
	95	=
	80	*if pos
	89	*3'

$$15 - n =$$

IF  
POSITIVE

Figure 3(c)

LOC	CODE	KEY
025	01	1
	93	•
	03	3
	05	5
	85	+
030	53	(
	15	E
	75	—
	01	1
	05	5
035	54	)
	65	×
	93	•
	00	0
	07	7
040	95	=
	81	HLT

$$1.35 + [(n - 15) \times .07] =$$

HALT

\*Denotes 2nd function key

Figure 3(d)

Figure 3. Program Code

LOC	CODE	KEY
	46	*LBL
	87	*1'
	93	•
045	01	1
	65	×
	15	E
	95	=
	81	HLT

1'

$$.1 \times n =$$

HALT

Figure 3(e)

LOC	CODE	KEY
050	46	*LBL
	88	*2'
	93	•
	05	5
	85	+
055	53	(
	15	E
	75	-
	05	5
	54	)
060	65	×
	93	•
	00	0
	09	9
	95	=
065	81	HLT

2'

$$.5 + [(n - 5) \times .09] =$$

HALT

\*Denotes 2nd function key

Figure 3(f)

Figure 3. Program Code

LOC	CODE	KEY
	46	*LBL
	89	*3'
	93	•
	09	9
070	05	5
	85	+
	53	(
	15	E
	75	—
075	01	1
	00	0
	54	)
	65	×
	93	•
080	00	0
	08	8
	95	=
	81	HLT

3'

$$.95 + [(n - 10) \times .08] =$$

HALT

Figure 3(g)

LOC	CODE	KEY
	46	*LBL
085	15	E
	43	RCL
	01	1
	09	9
	56	*rtn

\*Denotes 2nd function key

Figure 3(h)

Figure 3. Program Code

As described in Section IX, E is a user-defined key that when used alone in a program, the program will transfer to label E, execute instructions following label E, and automatically return to the location following E (location 008) when a return instruction is encountered. Figure 3h shows the program steps with label E which simply recalls  $R_{19}$  to the display. This is just one of many techniques which may be used to decrease the number of program steps in a program.

In the flow diagram of Figure 2, the value of  $n$  must be recalled to the display seven times. Since  $\boxed{RCL} 19$  requires three program steps, the total steps used to recall  $n$  is  $7 \times 3 = 21$  steps. Now consider that label E uses six locations (084 through 089) and each  $n$  in the flow diagram is replaced by E on a one-for-one basis. The total steps now required to recall  $n$  seven times is  $6 + 7 = 13$  steps, a saving of eight program locations.

Following the  $5 - E =$ , program steps (locations 005 through 008 in Figure 3a) is the if-positive function (location 009). If the result of  $5 - E =$  is zero or positive, the program transfer to label 1' as identified in location 010. Remember that a conditional transfer function (such as if-positive) must be immediately followed by a three-digit location number or a label to which the program will transfer if the result of the conditional transfer function is true. A false result of the if-positive function causes the program to skip location 010 and continue with location 011 shown in Figure 3.

In a nearly redundant fashion, the  $10 - n =$  and  $15 - n =$  operations are shown by Figures 3b and 3c respectively. Figure 3d shows the final operation in the main flow...  $1.35 + (n - 15) \times .07 = \dots$  and a halt instruction at location 041 stops program execution and displays the answer. Notice in the program steps that the algebraic hierarchy feature allows omission of the parentheses when entering the problem.

Figures 3e, 3f and 3g show the program steps for the calculations following labels 1', 2' and 3' respectively. Since this program uses labels to identify all transfer



operations, it is not necessary to keep track of program location numbers except to know which locations are used or not used. Label 1' and following steps could be moved to start with location 100 instead of location 042 without affecting the operation of the program. When a label is called by a program transfer instruction, the calculator seeks the program position where the label is identified and ignores location numbers.

Figure 3h, as previously indicated, is a simple housekeeping operation (a subroutine) which is used only to decrease the number of program steps. This sequence of instructions functions as a subroutine and therefore must be ended with a return instruction.

The CODE column on the coding form is for showing the two-digit code the calculator assigns to each key function. Cross references for converting key functions to key codes and vice versa are provided in Figures 1a and 1b.

## Enter the Program

After the program is written on a coding form, it can be entered into the calculator by resetting the program counter ( **2nd** **rset** ), switching into the learn mode ( **LRN** ) and pressing each key in the KEY column of the coding form beginning with location 000 (Figure 3a). Five digits are always displayed in the learn mode — three for the program location and two for the key code. Each keystroke (except **2nd** ) stores the function of the key pressed in the location number displayed prior to pressing the key. The program counter automatically advances to the next program location. For this reason, the two-digit key code will be displayed as zeros when first entering a program. If numbers do appear, they are from a previously entered program. When you are not sure that the last key pressed was the correct one, simply press **2nd** **bst** to decrement the program counter and view the key code at the last location. If it is not correct, press the correct key and continue. If it is correct, press **SST** one time and continue.

After the last key is pressed for the program in Figure 3, the program location number displayed should be 090. If not, too many or too few keys have been pressed and the following check and edit operations are mandatory. Press **LRN** again to take calculator out of learn mode.

## Check and Edit Program

After entering a new program into the calculator, it is a safe practice to check that the program is stored properly in the program memory. To check a program, press **2nd** **rslt** **LRN** and compare the displayed location and key code with the coding form by pressing **SST** repeatedly. When the key code displayed does not agree with the coding form for a particular location, steps must be taken to edit or correct the error. Unless the error is near the end of a program, you should step past the error a few locations, making note of the key codes, to determine one of the following:

1. Is the error an extra key entry?
2. Is the error a missing key entry?
3. Is the error simply a wrong key entry?

Figure 4 illustrates the three types of entry errors. In Figure 4a, the calculator displayed a second code 95 in location 009 and the following codes are displaced by one location. The error is corrected by backstepping ( **2nd** **bst** ) until the display shows 009 95. Pressing **2nd** **del** deletes the extra 95 code and raises the following codes to their proper positions and the display will show 009 80.

Figure 4b shows that the code 95 is missing which results in the following codes being displaced by one location. This error is corrected by backstepping to location 022 and pressing **INS** **=**. The insert key shifts the codes in locations 022 and up to the next lower location and inserts a 00 code in location 022. Pressing the equals key places the correct code in location 022, the counter advances and the display shows 023 80.

(a)

LOC	CODE	KEY	DISPLAY
	75	—	006 75
	15	E	007 15
	95	=	008 95
	80	*if pos	009 95 ← Extra key entry
010	87	*1'	010 80
	01	1	011 87
	00	0	012 01
	75	—	013 00

(b)

LOC	CODE	KEY	DISPLAY
	05	5	019 05
020	75	—	020 75
	15	E	021 15 ← Missing key entry
	95	⊖	022 80
	80	*if pos	023 89
	89	*3'	024 01
025	01	1	025 93
	93	•	026

(c)

LOC	CODE	KEY	DISPLAY
	95	=	048 95
	81	HLT	049 81
050	46	*LBL	050 46
	88	*2'	051 02 ← Wrong key entry
	93	•	052 93
	05	5	053 05
	85	+	054 85

\*Denotes 2nd function key

Figure 4. Key Entry Errors

A wrong key entry is evident in Figure 4c because all locations except 051 have the correct code displayed. In this case, the 2nd prefix was not originally entered, thus a numerical two was entered. This error is corrected by backstepping to location 051 and pressing the correct keys ( **2nd** **2** ).

If the location displayed is not near the location you desire to change, switch out of the learn mode, press **GTO** yyy (where yyy is the desired location number), and switch back into the learn mode to make the change.

## Document User Instructions

After the program is verified to have been correctly entered into the calculator program memory, the user instructions form should be used to document how to operate the calculator to run the special program. Of course, the basic information for the user instructions should have been developed along with the flow diagram prior to coding the program. Well documented user instructions will permit you or someone else to later run the program without considering the details of the flow diagram or the coding form. Figure 5 illustrates a user instructions form filled out for the sample program entered into the calculator.

The user instructions include: an appropriate title, information to be placed on the magnetic card if the program is recorded, informational steps of how to run the program, and notes which are pertinent to the user. Step one may seem basic but it serves as a reminder to check that the proper program is in the calculator, whether entered from the keyboard or a magnetic card.

## Running the Program

To use this program, suppose the number of checks shown by the last four bank statements are 4, 13, 10 and 22, and the individual service charges on the statements and the total service charge are required.



TITLE Checking Account Service Charge PAGE 1 OF 1

<b>A</b> Check Service Charge LD-1				
n → Ser Chg				RCL n

STEP	PROCEDURE	ENTER	PRESS	DISPLAY
1.	Enter Program			
2.	Enter number of checks	n	<b>A</b>	Service Chg.
3.	To recall value of n for displayed service charge:		<b>E</b>	n
	NOTE:			
	Charge rate per check is:			
	No. Checks (n)	Cost per Check		
	1-5	\$0.10		
	6-10	\$0.09		
	11-15	\$0.08		
	16 or more	\$0.07		

Figure 5. Sample Program User Instructions

Enter	Press	Display	Comments
4	<b>A</b>	.4	$.1 \times n$
	<b>STO</b> 01	.4	Store in R <sub>01</sub>
13	<b>A</b>	1.19	$.95 + (n - 10) \times .08$
	<b>SUM</b> 01	1.19	Sum to R <sub>01</sub>
10	<b>A</b>	.95	$.5 + (n - 5) \times .09$
	<b>SUM</b> 01	.95	Sum to R <sub>01</sub>
22	<b>A</b>	1.84	$1.35 + (n - 15) \times .07$
	<b>SUM</b> 01	1.84	Sum to R <sub>01</sub>
	<b>E</b>	22.	Check last entry
	<b>RCL</b> 01	4.38	Total service charge

An important item illustrated by the above example is that the individual results must be summed into memory to accumulate a total service charge. Only specially written programs have the effect of replacing the input variable with its associated output variable without affecting pending operations. You will recall this issue arose in connection with avoiding **≡** in subroutines. Thus, because the present program has **≡** in several places, you cannot expect to obtain the correct total service charge for four checks in March plus 13 checks in April by the sequence 4 **A** **+** 13 **A** **≡**. This is illustrated below:

Example: 4 checks + 13 checks  $\neq$  \$1.59

Enter	Press	Display	Comments
4	<b>A</b>	.4	Cost of 4 checks
	<b>+</b>	.4	Add is pending
13	<b>A</b>	1.19	Cost of 13 checks
	<b>≡</b>	1.19	Does not total

The pending add instruction was completed at the very first **≡** in the program. In this case, the correct value was obtained for 13 checks at the completion of routine **A**, although that would not always occur. Since the pending

add has been already completed, the final  $\boxed{=}$  in the example key sequence has no further effect.

Two options may be considered to allow keyboard arithmetic with program-produced numbers. The first is to avoid leaving operations pending when the program is invoked – using register arithmetic instead. This was the option illustrated in the example before, where  $\boxed{\text{SUM}} \boxed{01}$  was used to accumulate charges. The second is to write the program in the style of a subroutine, avoiding  $\boxed{=}$  entirely through use of parentheses, so that the effect of the program is to replace the input with the output without affecting pending operations.

You will probably want to permanently record your personal programs on blank magnetic cards. To record a program on a magnetic card, please refer to the detailed procedure, Recording a Magnetic Card in Section IX.

## Optimizing a Program

Of the many reasons to optimize a program, only two are significant. One reason is to simplify program interface requirements, and the second is to condense a program to fit in the 224 program-step limit.

Simplifying the program interface requirements means to make the program easier for you to use. Although the sample program in Figure 3 is already quite simple, one simple change which could be made is to have the final answer displayed with the decimal fixed at two places. Figure 6 illustrates the minor program changes required to make this improvement.

Condensing a program to a smaller number of steps is a time consuming exercise. If a program is less than 224 steps and operates properly, any time spent to condense the program is virtually wasted except for the personal satisfaction of doing it. If a program exceeds 224 steps by only a small number of steps, program optimization is warranted. However, if a program is excessively long, it should be split into two load modules... using the memory registers to hold intermediate data while changing load modules.

LOC	CODE	KEY		
000	46	*LBL	Existing program steps	
	11	A		
	42	STO		
	01	1		
	09	9		
Added steps	005	*fix	Sets display for fixed-point 2	
		02		2
		05	5	Existing program steps shifted to lower locations
		75	—	
		15	E	

\*Denotes 2nd function key

Figure 6. Program Interface Simplification

To help you become familiar with the problems of optimizing a program, we suggest that you optimize the program in Figures 2 and 3 to require fewer program steps.

Datamath Calculator Museum



## XV. AN ADVANCED SAMPLE PROGRAM OF MATHEMATICAL MODELING

The chart below shows census data for the United States for the years 1890-1970.

Year:	1890	1900	1910
U.S. Population:	62,947,714	75,994,575	91,972,266
	1920	1930	1940
	105,710,620	122,775,046	131,669,275
	1950	1960	1970
	150,697,361	179,323,175	203,235,298

What is the expected population, based on this data for the years 1980, 1984, and 2000? This type of problem is very common in many areas. What we attempt to do is to model the behavior of one variable (population) versus another (year) by means of a formula. The formula generally contains unknown constants, which we attempt to determine by a best-fit criterion using known data. Having selected the formula and determined the best-fit values for the constants in the formula, we may then use the resulting mathematical model to predict the values which will result for some unmeasured situation (a future year, for example).

Two models are of particular importance, inasmuch as they describe, between them, a large fraction of the relationships observed in the real world. They are:

Model 1      Linear model, where  $y = a + bx$ ;

Model 2      Exponential model, where  $y = \alpha e^{\beta x}$ .

In these equations,  $x$  is the independent variable (year) upon which the dependent variable  $y$  (population) depends. The quantity  $y$  is the model-value predicted; and  $(a, b)$  or  $(\alpha, \beta)$  are the unknown constants to be determined by means of a best-fit analysis of known data. Frequently, one doesn't know which of these two models achieves a better fit to the data, and must simply try both.

A statistical measure for determining which model is more successful is the **coefficient of determination**, which is the correlation coefficient between the observed data and the candidate model, squared. The model having the larger value for the coefficient of determination is best for the given situation. This coefficient varies between zero and one, the better the model – the higher the coefficient. This introduction sets up our problem.

Problem: Design a program to best predict the value of a variable at some future point in time based on a quantity of historical data.

We need a program to decide whether a linear or an exponential model best describes given empirical data (such as the census data shown). Then, based on this decision, extrapolate the curve to some chosen point in time.

At the highest level, the required program must perform four functions:

1. Assimilate the known data, saving whatever is necessary to perform steps 2 and 3.
2. Evaluate the coefficients of determination for each of the two models and select the superior alternative.
3. Determine the unknown constants for the superior model.
4. Use the model determined above to predict the value of the as yet unmeasured quantity versus the independent variable.

The initial step in the solution to this problem is the thought organization and equation collection phase. The first result of that process is the observation that the two models are almost identical: With the linear model we fit  $y = a + bx$ ; whereas with the exponential model we fit  $\ln y = \ln \alpha + \beta x$ . Thus in both cases, provided one first takes the natural logarithm of  $y$  or not as appropriate to the model, the fitting problem is of the same form: Fit  $Z$  ( $y$  or  $\ln y$ ) to a model of the form  $A$  ( $a$  or  $\ln \alpha$ ) plus  $B$

(b or  $\beta$ ) times x. From this we see that the working core of the program should be something which assumes a linear model  $Z = A + Bx$ , where the interpretation of Z (y or  $\ln y$ ) will depend upon which of the two original models is being examined.

Recognizing this, the equation collection phase produces the following equations\* for the best-fit values of A and B and the resulting coefficient of determination  $r^2$ .

$$B = \frac{\sum_{k=1}^n x_k Z_k - \frac{1}{n} \left( \sum_{k=1}^n x_k \right) \left( \sum_{k=1}^n Z_k \right)}{\sum_{k=1}^n x_k^2 - \frac{1}{n} \left( \sum_{k=1}^n x_k \right)^2}$$

$$A = \frac{1}{n} \left\{ \sum_{k=1}^n Z_k - B \sum_{k=1}^n x_k \right\}$$

$$r^2 = \frac{\left[ \sum_{k=1}^n x_k Z_k - \frac{1}{n} \left( \sum_{k=1}^n x_k \right) \left( \sum_{k=1}^n Z_k \right) \right]^2}{\left[ \sum_{k=1}^n x_k^2 - \frac{1}{n} \left( \sum_{k=1}^n x_k \right)^2 \right] \left[ \sum_{k=1}^n Z_k^2 - \frac{1}{n} \left( \sum_{k=1}^n Z_k \right)^2 \right]}$$

In these equations, n is the number of data pairs available, dependent variable versus independent variable. The quantity  $Z_k$  is defined equal to  $y_k$  for the case of the linear model and is equal to  $\ln y_k$  for the exponential model. The model 1 constants a and b are equal to A and B respectively when  $Z_k = y_k$ . The model 2 constants  $\alpha$  and  $\beta$  are related to A and B derived from setting  $Z_k = \ln y_k$  through the equations  $\beta = B$ ,  $\alpha = e^A$ .

\*These equations would be found from a study of a text on the relevant statistical subject – linear regression.

Observe that all the quantities  $A$ ,  $B$ , and  $r^2$  depend on the data through various sums ( $\Sigma$ ) which can be accumulated as we input the data. Our preliminary design is summarized by the flow chart in Figure 7.

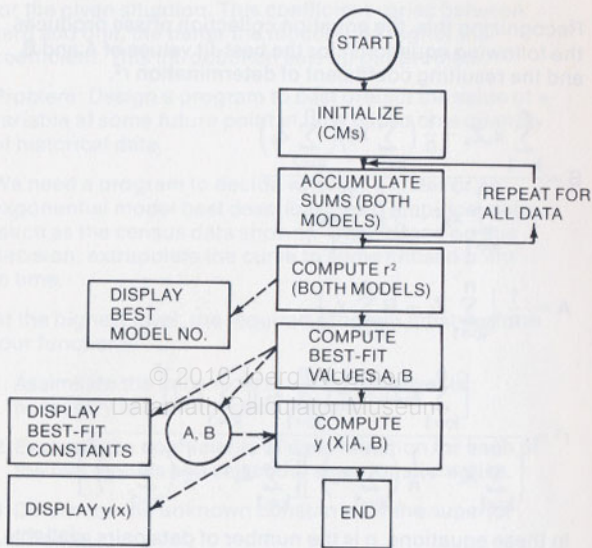


Figure 7. Mathematical Model Flow Chart

A preliminary specification for how the program should work is provided by our initial-version program user instructions (conceived before any code is written) in Figure 8.

STEP	PROCEDURE	ENTER	PRESS	DISPLAY
1	Load program			
2	Initialize		<input type="button" value="E"/>	
3	Independent variable	$x_k$	<input type="button" value="A"/>	$x_k$
4	Dependent variable	$y_k$	<input type="button" value="B"/>	$y_k$
	(Repeat steps 3 and 4 for all data pairs)			
5	Compute models		<input type="button" value="C"/>	Best mod. no.
			<input type="button" value="RUN"/>	$\alpha$ or $a$
			<input type="button" value="RUN"/>	$\beta$ or $b$
			<input type="button" value="RUN"/>	$r^2$
6	Predict $y(x)$	$x$	<input type="button" value="D"/>	$y(x)$
	(For new $x$ values repeat step 6)			

Figure 8. Preliminary User Instructions

The preliminary memory assignments may be made by examining the sums to be formed:

### Preliminary Memory Assignments

R <sub>00</sub>	unassigned	R <sub>10</sub>	$\Sigma (\ln y)^2$
R <sub>01</sub>	$x_k$	R <sub>11</sub>	$\Sigma x \ln y$
R <sub>02</sub>	$y_k$	R <sub>12</sub>	unassigned
R <sub>03</sub>	$\Sigma x$	R <sub>13</sub>	unassigned
R <sub>04</sub>	$\Sigma x^2$	R <sub>14</sub>	unassigned
R <sub>05</sub>	$n$	R <sub>15</sub>	unassigned
R <sub>06</sub>	$\Sigma y$	R <sub>16</sub>	unassigned
R <sub>07</sub>	$\Sigma y^2$	R <sub>17</sub>	unassigned
R <sub>08</sub>	$\Sigma xy$	R <sub>18</sub>	unassigned
R <sub>09</sub>	$\Sigma \ln y$	R <sub>19</sub>	unassigned

As we begin coding (figure 9), we expect to save various intermediate quantities in the unassigned memory registers. The user instructions and memory assignments represent a considerable fraction of the total programming design task.

As you look over the preliminary user instructions you may next wish to make an improvement. Recognizing that miskeying a number and pressing **B** could prove awkward as things stand, you choose to add a function (called B') which nullifies the effect of the wrong value of  $y_k$ . We therefore have a modification to the user instructions: "To correct an error after pressing **B**, press **2nd** **B'** and key in the correct value." All that is necessary at B' is to subtract from memory those quantities which were added as a result of pressing **B**. Whereas the code at B' is going to look almost like that at B (except for some inverse instructions) you seek a way to economize. See the Data Entry code in figure 9.

LOC	CODE	KEY	LOC	CODE	KEY	LOC	CODE	KEY
000	81	HLT		88	*2'		43	RCL
	46	*LBL		22	INV		00	0
	11	A	030	46	*LBL		01	1
	42	STO		88	*2'		60	*if flg
	00	0		44	SUM	060	00	0
005	01	1		00	0		78	*5'
	81	HLT		07	7		22	INV
	46	*LBL	035	30	* $\sqrt{x}$		46	*LBL
	12	B		23	ln x		78	*5'
	42	STO		60	*if flg	065	44	SUM
010	00	0		00	0		00	0
	02	2		89	*3'		03	3
	50	*st flg	040	22	INV		40	*x <sup>2</sup>
	00	0		46	*LBL		60	*if flg
	46	*LBL		89	*3'	070	00	0
015	17	*B'		44	SUM		79	*6'
	60	*if flg		00	0		22	INV
	00	0	045	09	9		46	*LBL
	87	*1'		40	*x <sup>2</sup>		79	*6'
	22	INV		60	*if flg	075	44	SUM
020	46	*LBL		00	0		00	0
	87	*1'		77	*4'		04	4
	44	SUM	050	22	INV		30	* $\sqrt{x}$
	00	0		46	*LBL		65	×
	06	6		77	*4'	080	43	RCL
025	40	*x <sup>2</sup>		44	SUM		00	0
	60	*if flg		01	1		02	2
	00	0	055	00	0		95	=

\*Denotes 2nd function key

Figure 9(a). Model Fit/Data Entry Module

LOC	CODE	KEY	LABELS	REGISTERS	
	60	*if flg	A $X_k$	00	
085	00	0	B $y_k$	01 last x	
	67	*7'	C	02 last y	
	22	INV	D	03 $\Sigma x$	
	46	*LBL	E Initialize	04 $\Sigma x^2$	
	67	*7'	A'	05 n	
090	44	SUM	B' Correction	06 $\Sigma y$	
	00	0	C'	07 $\Sigma y^2$	
	08	8	D'	08 $\Sigma xy$	
	43	RCL	E'	09 $\Sigma \ln y$	
	00	0		10 $\Sigma (\ln y)^2$	
095	01	1		11 $\Sigma x \ln y$	
	65	$\times$		12	
	43	RCL		13	
	00	0		14	
	02	2		15	
100	23	$\ln x$		16	
	95	=		17	
	60	*if flg		18	
	00	0		19	
	68	*8'		FLAGS	
105	22	INV		0 Used	
	46	*LBL		1	
	68	*8'		2	
	44	SUM		3	
	01	1		4	
110	01	1			
	01	1			

\*Denotes 2nd function key

Figure 9(a). Model Fit/Data Entry Module



LOC	CODE	KEY
112	60	*if flg
	00	0
	69	*9'
	22	INV
	46	*LBL
117	69	*9'
	44	SUM
	00	0
	05	5
	43	RCL
122	00	0
	02	2
	86	*rset
	46	*LBL
	15	E
127	47	*CMs
	81	HLT

\*Denotes 2nd function key

Figure 9(b). Model Fit/Data Entry Module

This code performs all of the initialization, data summing, and error correction in less space than would have been required had you not thought to use a flag to skip around all the inverse instructions for B and hit upon all of them for B'. You now count and find that 129 program steps have been used. It does not take you long to discover that there is much left to be done and that fitting it all into 95 program steps is impossible. You might consider removing the B' error-correction feature; however, as you assess the future computations, you recognize that the remaining tasks will require almost all of a 224-step load

module anyway. Your decision is therefore to split the problem into two load modules. The first, which you already have completed, is the data entry module. The second is the analysis module which must perform all the remaining computations. Furthermore as you recognize the amount of computation left to be done you note that the preliminary user instructions present a (or  $\alpha$ ) and b (or  $\beta$ ) in the opposite sequence from the one naturally following from the equations. These considerations result in another modification to the user instructions: "After performing all empirical data entry, load program card number 2." Also, "Following determination of the best-fit model type at A" (rather than C as before) "the quantities  $\beta$  (or b) and  $\alpha$  (or a) are found in sequence by pressing RUN." Finally, " $r^2$  is shown upon pressing RUN again."

Now examine the remaining problem. The similarity in what must take place to compute A, B, and  $r^2$  for the two models is evident. The only difference appears to lie in whether the quantities stored in  $R_{06}$ - $R_{08}$  or those in  $R_{09}$ - $R_{11}$  are used. The technique which comes to mind is to call a subroutine and let it use some indirect memory references. In order to compute  $r^2$  this subroutine will need to compute the quantities

$$\sum_{k=1}^n x_k z_k - \frac{1}{n} \left( \sum_{k=1}^n x_k \right) \left( \sum_{k=1}^n z_k \right)$$

and

$$\sum_{k=1}^n z_k^2 - \frac{1}{n} \left( \sum_{k=1}^n z_k \right)^2$$

Because the first of these will be needed to find A and B, we choose to store it in  $R_{18}$  in the subroutine where it is computed. But there are two models to be evaluated and hence two such values to be saved. We select  $R_{01}$  to hold the value produced by the subroutine for model-1, as  $R_{01}$  is now free to use again. To distinguish between  $R_{06}$ - $R_{08}$

and R<sub>09</sub>-R<sub>11</sub> by indirect addressing, reserve R<sub>15</sub>-R<sub>17</sub> as pointer registers. It is apparent that we still need a place to store the quantity

$$\sum_{k=1}^n x_k^2 - \frac{1}{n} \left( \sum_{k=1}^n x_k \right)^2$$

for this quantity is used several times in the calculations. Select R<sub>14</sub> for this purpose. Also, the r<sup>2</sup> value for model-2 must be compared with the value for model-1, so we must save the model-1 value somewhere and choose R<sub>12</sub>. This results in the following preliminary memory assignments for the Analysis load module.

#### Preliminary Memory Assignments (Second Load Module)

R <sub>00</sub> unassigned	R <sub>10</sub> $\sum (\ln y)^2$
R <sub>01</sub> $\sum xy - n^{-1} (\sum x) (\sum y)$	R <sub>11</sub> $\sum x \ln y$
R <sub>02</sub> unassigned	R <sub>12</sub> r <sup>2</sup> (linear model)
R <sub>03</sub> $\sum x$	R <sub>13</sub> unassigned
R <sub>04</sub> $\sum x^2$	R <sub>14</sub> $\sum x^2 - n^{-1} (\sum x)^2$
R <sub>05</sub> n	R <sub>15</sub> Pointer 1 (6 or 9)
R <sub>06</sub> $\sum y$	R <sub>16</sub> Pointer 2 (7 or 10)
R <sub>07</sub> $\sum y^2$	R <sub>17</sub> Pointer 3 (8 or 11)
R <sub>08</sub> $\sum xy$	R <sub>18</sub> $\sum xZ - n^{-1} (\sum x) (\sum Z)$
R <sub>09</sub> $\sum \ln y$	R <sub>19</sub> unassigned

You may now proceed with much of the coding of the second load module (figure 10). Name the subroutine C' which computes r<sup>2</sup>, and name two as yet undefined subroutines (for finding A and B) A' and B'. Although A' and B' are not defined yet they doubtless will require a pointer (either to R<sub>01</sub> or to R<sub>18</sub>). We store this pointer in R<sub>13</sub>. The main portion of the analysis module can now be coded — see instructions 000-110 of the Analysis Module at the back of this section.

LOC	CODE	KEY
000	46	*LBL
	11	A
	43	RCL
	00	0
	04	4
005	75	—
	43	RCL
	00	0
	03	3
	40	*x <sup>2</sup>
010	55	÷
	43	RCL
	00	0
	05	5
	95	=
015	42	STO
	01	1
	04	4
	06	6
	42	STO
020	01	1
	05	5
	07	7
	42	STO
	01	1
025	06	6
	08	8
	42	STO

LOC	CODE	KEY
	01	1
	07	7
030	18	*C'
	42	STO
	01	1
	02	2
	43	RCL
035	01	1
	08	8
	42	STO
	00	0
	01	1
040	03	3
	44	SUM
	01	1
	05	5
	44	SUM
045	01	1
	06	6
	44	SUM
	01	1
	07	7
050	18	*C'
	75	—
	43	RCL
	01	1
	02	2
055	95	=

LOC	CODE	KEY
	22	INV
	80	*if pos
	75	—
	44	SUM
060	01	1
	02	2
	50	*st flg
	01	1
	02	2
065	81	HLT
	01	1
	08	8
	42	STO
	01	1
070	03	3
	17	*B'
	81	HLT
	16	*A'
	22	INV
075	23	ln x
	81	HLT
	43	RCL
	01	1
	02	2
080	81	HLT
	46	*LBL
	75	—
	22	INV

\*Denotes 2nd function key

Figure 10(a). Model Fit/Analysis Module

LOC	CODE	KEY	LABELS	REGISTERS
	50	*st flg	A Compare	00
085	01	1	B	01 $\Sigma xy - n^{-1} \Sigma x \Sigma y$
	01	1	C	02
	81	HLT	D	03 $\Sigma x$
	03	3	E Estimate	04 $\Sigma x^2$
	94	+/-	A' A-value	05 n
090	44	SUM	B' B-value	06 $\Sigma y$
	01	1	C' $r^2$	07 $\Sigma y^2, A$
	05	5	D'	08 $\Sigma xy, B$
	44	SUM	E'	09 $\Sigma \ln y$
	01	1		10 $\Sigma (\ln y)^2$
095	06	6		11 $\Sigma x \ln y$
	44	SUM		12 $r^2$
	01	1		13 Pointer 4
	07	7		14 $\Sigma x^2 - n^{-1} (\Sigma x)^2$
	01	1		15 Pointer 1
100	42	STO		16 Pointer 2
	01	1		17 Pointer 3
	03	3		18 $\Sigma xZ - n^{-1} \Sigma x \Sigma Z$
	17	*B'		19
	81	HLT		FLAGS
105	16	*A'		0
	81	HLT		1 Used
	43	RCL		2
	01	1		3
	02	2		4
110	81	HLT		
	46	*LBL		

\*Denotes 2nd function key

Figure 10(a). Model Fit/Analysis Module

LOC	CODE	KEY
112	15	E
	65	×
	43	RCL
	00	0
	08	8
117	85	+
	43	RCL
	00	0
	07	7
	95	=
122	60	*if flg
	01	1
	22	INV
	81	HLT
	46	*LBL
127	22	INV
	22	INV
	23	ln x
	81	HLT
	46	*LBL
132	17	*B'
	53	(
	36	*IND
	43	RCL
	01	1
137	03	3
	55	÷
	43	RCL

LOC	CODE	KEY
	01	1
	04	4
142	54	)
	42	STO
	00	0
	08	8
	56	*rtn
147	46	*LBL
	16	*A'
	53	(
	53	(
	36	*IND
152	43	RCL
	01	1
	05	5
	75	-
	43	RCL
157	00	0
	03	3
	65	×
	43	RCL
	00	0
162	08	8
	54	)
	55	÷
	43	RCL
	00	0
167	05	5

LOC	CODE	KEY
	54	)
	42	STO
	00	0
	07	7
172	56	*rtn
	46	*LBL
	18	*C'
	53	(
	53	(
177	36	*IND
	43	RCL
	01	1
	07	7
	75	-
182	43	RCL
	00	0
	03	3
	65	×
	36	*IND
187	43	RCL
	01	1
	05	5
	55	÷
	43	RCL
192	00	0
	05	5
	54	)
	42	STO

\*Denotes 2nd function key

Figure 10(b). Model Fit/Analysis Module

LOC	CODE	KEY
	01	1
197	08	8
	40	*x <sup>2</sup>
	55	÷
	43	RCL
	01	1
202	04	4
	55	÷
	53	(
	36	*IND
	43	RCL
207	01	1
	06	6
	75	-
	36	*IND
	43	RCL
212	01	1
	05	5
	40	*x <sup>2</sup>
	55	÷
	43	RCL
217	00	0
	05	5
	54	)
	54	)
	56	*rtn
222		

\*Denotes 2nd function key

Figure 10(b). Model Fit/Analysis Module

The work is not complete, for you must still write the portion of the main routine which evaluates  $y(x)$  and the subroutines  $A'$ ,  $B'$ , and  $C'$ . The foregoing code has thoughtfully included a flag for remembering which model was best. Flag 1 is set if the exponential model is best and reset otherwise. Allocating  $R_{07}$  and  $R_{08}$  to the constants  $A$  and  $B$ , the evaluation portion of the main routine can now be written. Since  $E$  has not been used in the second load module, modify the user instructions so as to name this program portion  $E$  (rather than  $D$  as it was earlier). You may now add the code instructions 111-130 to the Analysis Module.

It now only remains to write the three subroutines to solve for  $B$ ,  $A$ , or  $r^2$ . These subroutines follow in a straightforward way from the equations.

This completes the programming. The final code, memory and label assignments, and user instructions would be documented and appear as in figure 11 after checking the program with several problems having known answers.

		<b>←A←</b> Model Fit/Data Entry		Card 1
		Delete		
$x_k$		$y_k$		INIT

CARD 1

		<b>←A←</b> Model Fit/Analysis		Card 2
Compare				Estimate

CARD 2

Figure 11a. Model Fit User Instructions



STEP	PROCEDURE	ENTER	PRESS	DISPLAY
1	Load program Card 1			
2	Initialize		<b>E</b>	
	Perform 3-4 for $k=1, \dots, n$			
3	Independent variable	$x_k$	<b>A</b>	$x_k$
4	Dependent variable	$y_k$	<b>B</b>	$y_k$
	If you make a mistake entering $x_k$ it may be corrected before step 4 by reentering the proper value. To delete an incorrect entry.		<b>2nd</b> <b>B'</b>	
	Reenter proper value of $y_k$ after B'			
5	Load program Card 2			
6	Find best model*		<b>A</b>	Best mod. no.
	Find b or $\beta$		<b>RUN</b>	b or $\beta$
	Find a or $\alpha$		<b>RUN</b>	a or $\alpha$
	Find $r^2$		<b>RUN</b>	$r^2$
7	Find model estimate for any value of x	x	<b>E</b>	y (x)
	(Repeat as desired)			
	*Model 1: $y = a + bx$			
	Model 2: $y = \alpha e^{\beta x}$			

Figure 11b. Model Fit User Instructions

Notice that instructions 131-146 in the Analysis Module compute A, instructions 147-172 compute B, and 173-221 computes  $r^2$ .

The program may now be used to analyze the census data with the following results. The superior model is model no. 2, the exponential, with a coefficient of determination given by  $r^2 = .9914660242$ , a very good fit. The resulting model is

$$y = \alpha e^{\beta x}$$

where  $\alpha = .0001716447$

and  $\beta = .0141183065$ .

(This last value, for  $\beta$ , implies a growth rate of about 1.41 percent per year.)

The predicted populations for the years 1980, 1984, and 2000 are given below rounded to the nearest integer.

1980: 237,140,403

1984: 250,917,854

2000: 314,510,831

Of course the same program can now be used to analyze all kinds of data – vehicular traffic accidents, business growth, cost-of-living, etc. Whether you really wish to combine two statistical models into one program as was done here is debatable perhaps. However, that was the premise we started with; and it was sufficiently stressing to involve almost all of the instructions you have learned about.

# APPENDIX A

## MAINTENANCE AND SERVICE INFORMATION

### BATTERY AND AC ADAPTER/CHARGER OPERATION

#### Normal Operations

To ensure maximum portable operating time, connect the AC9130 Adapter/Charger to a standard 115 Vac/60 Hz outlet, plug into calculator, and charge battery pack 4 hours with the calculator off or 10 hours with the calculator on. If during portable operation the display appears dim or erratic, connect the adapter/charger and continue calculations. **CAUTION: Calculator can be damaged if the adapter/charger is connected without the battery pack installed.**

#### Periodic Recharging

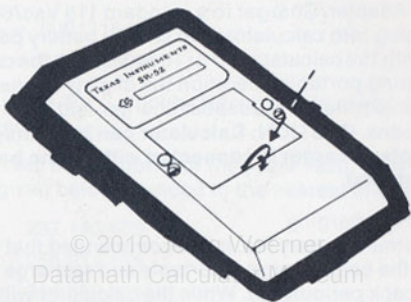
For maximum battery life, it is recommended that you operate the calculator as a portable and recharge the battery pack periodically. While the calculator will operate indefinitely with the adapter/charger, the rechargeable battery pack can lose its storage capability if it is not allowed to discharge occasionally.

#### Excessive Battery Discharging

If the calculator is left on for an extended period of time after the battery pack is discharged (for example, accidentally left on over night), connect the adapter/charger for at least 16 hours with the calculator off. Repeated occurrences of excessive battery discharging will permanently damage the battery pack. Spare and replacement BP-1 battery packs can be purchased from your local TI Retailer or directly from Texas Instruments.

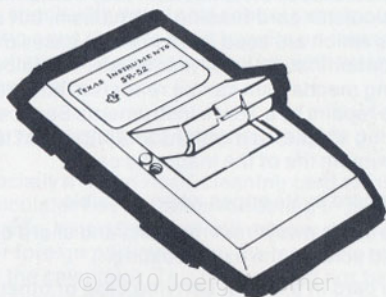
## BATTERY PACK REPLACEMENT

The battery pack can be quickly and simply removed from the calculator. Hold the calculator with the keys facing down. Place a small coin (penny, dime) in the slot in the bottom of the calculator. A slight prying motion with the coin will pop the slotted end of the pack out of the calculator. The pack can then be removed entirely from the calculator.



The exposed metal contacts on the battery pack are the battery terminals. Care should always be taken to prevent any metal object from coming into contact with the terminals thereby shorting the batteries.

To reinsert the battery pack, place the rounded part of the pack into the pack opening so that the small step on the end of the pack fits under the edge of the calculator bottom. The slotted end of the pack will then be opposite the instruction label. A small amount of pressure on the battery pack will snap it properly into position.



© 2010 Joerg Jander  
Datamath Calculator Museum

## CARING FOR MAGNETIC CARDS

The magnetic cards have the ability to retain information placed on them for an indefinite amount of time. The recorded information does not tend to fade or weaken with age and will remain unchanged until actually altered by an external magnetic field. While the magnetic signal will not deteriorate, the physical characteristics of the card and the card drive unit in the calculator are susceptible to damage.

### Handling Cards

Developing good habits in handling magnetic cards is important. A card which is physically marred, creased or dented may be useless for its intended purpose. However, physical degradation of a card generally results from an accumulation of mishaps or poor handling techniques.

There are numerous contaminants to consider. Ashes, food particles, drinks, dust and oil-based liquids are the most common contaminants to guard against. A card can be contaminated by placing it directly on a contaminated surface; or indirectly, by transferring the contaminant to the card with your fingers. Even the natural oils on your fingers will transfer to the cards and cause accumulation of dust and foreign particles. Note that using one contaminated card in the calculator may contaminate not only the calculator card reading mechanism, but also other cards which are used later. In some cases of extreme contamination by oily materials, the calculator card reading mechanism can be rendered inoperative and will require repairs by a Texas Instruments Service Facility. The following simple instructions are important to assure maximum life of the magnetic cards.

1. Handle a card by its edges when possible.
2. Keep the cards away from magnets and sharp objects that could scratch the oxide coating.
3. Keep the card in the vinyl carrying case or other protective container while the card is not in use.
4. If a card is contaminated, clean it immediately.

### **Cleaning Cards**

Contaminated card may be cleaned easily without using special cleaners or solvents. Petroleum based fluids should not be used under any circumstances to clean cards. Dust and foreign particles should be removed from a card with a soft brush or a dry soft cloth. Other forms of contamination may be washed from the card with warm water and a small amount of mild liquid detergent. Rinse the card and dry with a soft cloth.

## Writing on Cards

The blank magnetic cards furnished with your calculator have areas designated for you to write numbers, symbols and abbreviated titles for your personal programs. You may write information temporarily on a card with a soft, fine-lead pencil or a fine-point, felt-tip pen with washable ink. Of course, a felt-tip pen with non-washable or permanent ink will permanently mark your card. For best results, check with your local school supply outlet and ask for felt-tip pens that are used to write on transparencies. Most outlets carry a variety of colors with washable or permanent inks.

## USING THE HEAD-CLEANING CARD

The specially marked head-cleaning card furnished with your calculator has an abrasive coating in place of the usual oxide. Using this card will remove any buildup of oxide or foreign particles from the magnetic read/write head in the calculator. This card should not be used as an all-purpose remedy for any difficulty experienced, as excessive use could change the characteristics of the read/write head. The In Case of Difficulty instructions should normally be used as the guide for when the head cleaning card may be used to remedy a difficulty. To use the card; press **2nd read**, insert the card into the lower slot of the calculator as you would a regular card, and let the drive motor pull the card through the calculator. Press **CLR** if the display flashes after using the card. The head cleaning card should be used sparingly and no more than one time per difficulty.

## IN CASE OF DIFFICULTY

In the event that you have difficulty with your calculator, the following instructions will help you analyze the problem and you may be able to fix your calculator without returning it to a service center. If the suggested remedies are not successful, contact the Consumer Relations Department by mail or telephone (refer to If You Have Questions or Need Assistance). Please describe in detail the symptoms of your calculator.

If one of the following symptoms appears while operating with the optional printing unit, remove the calculator and reinstall the battery pack before using the following procedures. If the symptom disappears when the calculator is removed from the printing unit, refer to the printing unit manual.

1. While performing keyboard operations, the calculator display flashes erratic numbers, grows dim or even goes blank. Or, card reader turns on automatically.

The battery pack is discharged, refer to Battery and AC Adapter/Charger Operation.

2. Display is blank for no obvious reason.

Press and hold **[HLT]** momentarily. If display turns on, the calculator was in the run mode executing a loop program with no means to stop or waiting for a card to be inserted. Check program for improper code. If display does not turn on, the battery pack is not properly installed or it is discharged.

3. Display flashes each time one of the user-defined keys is pressed.

The key pressed has not been assigned as a label in the program or an illegal operation, overflow or underflow occurred while the program was running.



4. Display flashes after reading a magnetic card.

The calculator has detected a reading error. Repeat the card reading procedure. If difficulty continues, try reading other cards. If other cards read properly, check the first card for physical defects or contamination and clean or replace card as necessary. If other cards do not read properly, use the head cleaning card one time — refer to **Using the Head-Cleaning Card**.

5. While reading or recording a magnetic card, the card stops before it should or even stops inside the calculator.

Press and hold **HLT** momentarily. If display does not turn on, the battery pack is discharged. If display turns on, press **2nd read**. If card passes on through the calculator, check card for dust, smudges on top side, or physical defects and clean or replace card as necessary. If the card does not move, but the distinctive whine of the drive motor can be heard, push the card on through with your finger or another card. Check card for contamination or physical defects.

6. A program which has been read from a prerecorded card does not run properly.

Check contents of program memory against program listing for that program. If an incorrect instruction is found in program memory, perform Diagnostic 3 (BA1-20) in the Basic Library. If an incorrect instruction is not found in program memory, perform Diagnostic 1 (BA1-18) and Diagnostic 2 (BA1-19). If an error code is displayed when running either diagnostic, reread the diagnostic card and check for same error code.

7. Calculator displays incorrect results.

Perform Diagnostic 1 (BA1-18) and Diagnostic 2 (BA1-19) in the Basic Library. If an error code is not displayed when running either diagnostic, check for invalid key sequence. If an error code appears, repeat diagnostic and confirm error code.

When returning your calculator for repair, return the calculator, adapter/charger, and any magnetic cards which were involved when the difficulty occurred. For your protection, the calculator must be sent insured; Texas Instruments cannot assume any responsibility for loss of or damage to uninsured shipments. Include information on the difficulty experienced with the calculator as well as return address information including name, address, city, state and zip code. The shipment should be carefully packaged, adequately protected against shock and rough handling, and sent to one of the Texas Instruments Service Facilities listed on the back cover.

NOTE: The P.O. box number listed for the Lubbock Service Facility is for United States parcel post shipments only. If you desire to use another carrier, please call the Consumer Relations Department for the proper shipping address.

## **IF YOU HAVE QUESTIONS OR NEED ASSISTANCE**

© 2010 Joerg Woerner

If you have questions or need assistance with your calculator, write the Consumer Relations Department at:

Texas Instruments Incorporated  
P.O. Box 22283  
Dallas, Texas 75222

or call Consumer Relations at 800-527-4980 (toll-free within all contiguous United States except Texas) or 800-492-4298 (toll-free within Texas). If outside the contiguous United States, call 214-238-5461. (We regret that we cannot accept collect calls at this number.)

For repair-related inquiries only, you may also call our Service Facility toll-free at 800-858-1802 (800-692-1353 within Texas).

## APPENDIX B ERROR CONDITIONS

A number of different situations result in a flashing display, signaling an error condition. These conditions and the quantity flashed when in (or upon returning to) the calculate mode are summarized here.

### Underflow and Overflow

When a number entry or calculation results in a non-zero quantity whose magnitude is less than  $1. \times 10^{-99}$  an underflow condition exists and the display flashes 1.—99. Similarly, if a magnitude greater than  $9.999999999 \times 10^{99}$  should occur, the display will flash 9.999999999 99 to indicate overflow.

### Division by Zero

Attempting to divide by zero or to take the reciprocal of zero results in an error with the same indication as for overflow, a flashing 9.999999999 99 display.

### Function Argument Outside of Range

The mathematical functions have certain restrictions placed on their arguments in addition to those imposed by the overflow/underflow criterion. The functions,

invalid arguments, and the error indications are summarized below.

Function	Invalid Argument	Quantity Flashed
$x!$	Negative or non-integer	$\text{Int}( x )!$ , where $\text{Int}(x) =$ integer part of $x$ .
$\sqrt{x}$	Negative number	$\sqrt{ x }$
$\ln x$	Negative number	$\ln( x )$
$\log x$	Negative number	$\log( x )$
$\sin^{-1}x$	$ x  > 1$	$x$
$\cos^{-1}x$	$ x  > 1$	$x$
$y^x$	$y < 0$	$ y ^x$
$\sqrt[y]{y}$	$y < 0$	$\sqrt[ y ]{ y }$
$\sqrt[y]{y}$	$x = 0, y = 0$	1.

## Undefined Transfers

The display will flash the value currently in the display register whenever an attempt is made to transfer to an undefined location. Examples of such errors are:

1. Attempting to transfer to an address not in the range 000 through 223.
2. Attempting to transfer to a label address which has not been defined by means of a label instruction.
3. Indirect transfers where the pointer is not one of the integers 0 through 223.

## Attempt to Execute Past Location 223

Whenever the program counter reaches location 223, if there is no halt, return, reset, or other transfer instruction there the program counter will attempt to increment to location 224. However 224 is an invalid program address and this gives rise to an error similar to those discussed above. The display flashes the current value in the display register.

## Exceeding Capacity of Internal Registers

The internal processing registers can accommodate up to ten pending operations, and hence up to nine open left parentheses. Any calculation which attempts to exceed these maxima results in an error indication. The display flashes the current display value.

## Illegal Operation Sequences

Various sequences of keystrokes are meaningless and result in an error condition. Particularly, these include sequences with missing operands such as the following:

RCL 01 + )

8.1 ÷ =

RCL 11 + X

142 y<sup>x</sup> +

Such sequences with ) or = immediately following an arithmetic operator or with two consecutive operators (or two-variable functions) not separated by an operand are illegal. The display flashes the current display-register value.

## Clearing an Error Condition

The display can be stopped from flashing by pressing CE. In most cases this also removes all internal conditions indicating that an error is present. Whether the calculation can proceed after an error has occurred depends on the type of error, the problem itself, and quantities saved in memory. However, there is one error condition which remains latent even after the flashing has been stopped with the CE key. When direct register arithmetic results in underflow or overflow of that register, the error condition remains until the contents of that register are changed.

Example:

Enter	Press	Remarks
1	<b>EE</b>	
51	<b>STO</b> 05	$1 \times 10^{51}$ stored in R <sub>05</sub>
	<b>2nd</b> <b>PROD</b> 05	Overflow in R <sub>05</sub>
	<b>CE</b>	Flashing halted
9	<b>2nd</b> <b><math>\sqrt{x}</math></b>	Calculation $\sqrt{9} = 3$ performed properly
	<b>RCL</b> 05	Error condition in R <sub>05</sub> still present

## Errors Encountered in the Run Mode

When any of the foregoing errors occur in the run mode, what happens next depends upon the programmer. Program halts are not an automatic consequence of an error condition. The program will continue, using the value which would have flashed in the calculate mode for subsequent calculations, and the presence of an error will be signalled by flashing the "answer" obtained when the program halts. This may or may not be the correct answer, depending upon the problem and the type of error condition. However, it is the best selection which can be made in the absence of specific programming directives to take other action. These directives, which should be supplied in the program at points where error conditions might arise, utilize the if-error branching instructions discussed in Section X.

## Errors In Reading or Writing Magnetic Cards

When the display flashes immediately following a card read operation, the calculator has detected a read error. Repeat the read operation, if display continues to flash, refer to Appendix A. If the display flashes after recording a card, the black tab over the write-protect window of the card is missing or improperly positioned.

## GLOSSARY

**Address** – A location in program memory – designated by either an absolute address (a number from 0 through 223) or a label assigned in a program.

**Addressable Register** – One of the 20 storage areas of data memory.

**Algebraic Heirarchy** – The rules providing a unique interpretation of an expression which lacks a completely definitive set of parentheses. The SR-52 interprets expressions using these rules when parentheses have not all been explicitly included.

**Angular Modes** – The two options, degrees or radians, in which angles are to be expressed. Selected by slide switch on upper left of keyboard.

**Backstep** – To decrement the program counter in the learn mode for editing purposes. Effected by **2nd** **bst** .

**Basic Program Library** – The set of recorded programs and manual included with your SR-52.

**Branching Instruction (conditional transfer instruction)** – A decision making program statement offering a choice of ways to continue processing.

**Calculate Mode** – The type of SR-52 operation in which calculations are performed under step-by-step control from the keyboard.

**Card (Magnetic Card, Program Card)** – The magnetic strip for permanently recording a program.

**Case Statement** – A type of programming element wherein transfer is to be made to one of n prescribed locations, depending upon the value (1 through n) of a control variable K.

**Chain Operations** – A sequence of mathematical operations where the result of one calculation is used as the starting point of the next, all the way to the end of the sequence.

**Clear** – A generic term meaning to reset to the original zero starting condition. One may clear entries only (see **CE** in the key index), clear display and pending operations ( **CLR** ), clear data memory only ( **2nd** **CMs** ), or reset program flags and clear subroutine return-pointer registers ( **2nd** **rsl** ).

**Code** – See Program Coding or Key Code.

**Commands** – Program instructions.

**Conditional Transfer Instruction (Branching Instruction)** – A decision making program statement offering a choice of ways to continue processing. Effected by **2nd** – **if pos** , **if zero** , **if err** , **if flg** or **dsz** .

**Control Flags** – See Flags.

**Coordinate Transformation** – Conversion from polar coordinates to rectangular coordinates or vice versa, an operation provided by the **2nd** **P/R** keys or their inverse.

**Data Memory** – Twenty addressable registers numbered 00-19. See Memory.

**Decisions** – The results obtained through use of the branching instructions.

**Degree Mode** – See Angular Modes.

**Delete** – The act of eliminating one or more program steps as a part of the editing process. Subsequent steps are automatically moved up to fill the gap. Effected by the **2nd** **del** keys.



**Direct Register Arithmetic (Direct Register Operations)** – Performing addition, subtraction, multiplication, or division upon the contents of an addressable register (leaving the answer in that register) without recalling the register contents from memory. Effected by **SUM** ,

**INV SUM** , **2nd PROD** , and **INV 2nd PROD** .

**Display** – The 10 digit (maximum) representation of the display register, plus 2-digit scientific notation.

**Display Format** – The manner in which numbers are being displayed. There are two independent issues: scientific notation usage and mantissa format.

**Display Register** – The register which contains the quantity most recently computed, recalled from memory, or entered from the keyboard. Can carry 12 digits plus scientific notation.

**Drive Motor** – The mechanical part of the calculator that transports the card through the machine.

**Dummy Operations** – Those program steps which serve solely to supply the current display-register value as an operand. Example: In the sequence **RCL 01 [Inx] + ( STO ÷ RCL 02)**, the **STO** is an economical way of introducing the quantity  $\ln(\cdot R01)$  as the first operand of the open parentheses.

**Editing** – The processes of altering, adding, and deleting instructions as a final process of creating a working and satisfactory program. Effected by the **2nd del** , **INS** , **SST** , and **2nd bst** keys.

**Error Conditions** – A variety of situations which arise when the calculation encounters ill-defined quantities, illegal operations, or numbers beyond the capacity of the SR-52. (See Appendix B.)

**Exchange** – The operation in which the content of the display register is exchanged with that of a specified addressable register. Effected by the **2nd EXC** keys.

**Execution** – The phase during which the SR-52 is running under program control (the run mode). The controlling program is said to be in the process of execution.

**Exponent** – As used here, the power of ten associated with a scientific notation number representation.

**Exponentiation** – A built-in two-variable function for raising  $y$  to the  $x$ th power. Effected by the  $y^x$  key.

**Expressions** – Instruction sequences which acquire a value depending upon register contents and which, when taken alone, leave no operations pending or operators unsatisfied. Expressions set off by parentheses may be combined with other such expressions to form larger expressions.

**Extraction of Roots** – See Root Extraction.

**Fixed-Point** – The content of the display register can be rounded to the number of places specified by the  $\boxed{2nd}$   $\boxed{FIX}$  key.

**Flags (Program Flags)** – On/off program switches numbered 0-4 used as markers for various events in a program. Two-state devices which can be changed from the keyboard or in a program. Certain branching instruction  $\boxed{IF}$   $\boxed{flg}$  tests the state of a flag as the basis for a transfer decision. Flags are "set" by  $\boxed{2nd}$   $\boxed{st}$   $\boxed{flg}$  and "reset" by  $\boxed{2nd}$   $\boxed{INV}$   $\boxed{st}$   $\boxed{flg}$ .

**Flowchart** – A programming design device which graphically charts the paths of processing through a program.

**Format** – See Display Format, Mantissa Format.

**Functions** – The keyboard mathematical operations of one and two variables (such as  $x!$  or  $y^x$ ).

**Function Key**— The meaning attached to each key: The first function is accessed directly; the second, through prefixing the key with **2nd** .

**Head-Cleaning**— An occasional maintenance process to ensure proper operation of the magnetic card read/write mechanism.

**Illegal Instructions**— Sequences which are not proper in the SR-52 discipline. See Appendix B.

**Indirect Instruction**— Any instruction which uses the contents of a specified register as a pointer to the actual data register or program address.

**Initial Display Mode**— The display mode invoked when the SR-52 is first turned on. It uses the initial mantissa format, but does not indicate scientific notation.

**Initial Mantissa Format**— That type of display wherein up to ten digits are used to represent a number. A decimal point can be present anywhere in the number and trailing zeros are suppressed.

**Insert**— A program editing procedure whereby one pushes down all instructions from the current location leaving a null instruction. A new program step is then inserted to take its place. Effected by **INS** .

**Instruction**— One or more key steps which define an action to be taken in a program.

**Instruction Code**— See Key Code.

**Internal Processing Registers**— The ten registers which, along with the display register, are used by the SR-52 to evaluate expressions with pending operations without affecting the user's data registers.

**Interruption**— The act of halting program execution from the keyboard without precise knowledge of the state of processing at the time of intervention.

**Inverse Operations** – Those operations which result when an operation is prefixed by **INV** , which reverses the effect of the operation.

**Key** – Sometimes used to mean one of the 45 physical keyboard elements, but more generally used to denote one of the 88 key functions, including both first and second functions.

**Key Code** – The two-digit representation of each key based upon its column (1-5 for primary meanings or 6-0 for second meanings starting from the left) and row (1-9 from top to bottom) location on the keyboard.

**Key, User-Defined** – One of the ten functions, **A** through **E** and **2nd A** through **2nd E** which provide a starting point for program execution by merely pressing that key.

**Label** – A name assigned to a particular point in a program which can be referenced by a transfer instruction or by program initialization. Effected by **2nd LBL** .

**Learn Mode** – The type of SR-52 operation in which keystrokes alter the contents of program memory. This mode is used for keyboard construction and editing of a program. Effected by **LRN** .

**Levels of Parentheses** – The number of operations made pending by means of open (left) parentheses.

**Levels of Routines** – The number of program segments, either active or suspended, and awaiting return of control in a program.

**List** – To effect a step-by-step printed record of the instructions of a program. (Can be used only with the optional printing unit.) Effected by **2nd list** .

**Load Module** – Any portion of a partitioned program which resides (in its entirety) in program memory at some given time.

**Locations** – Positions in program memory (000-223).

Logical Tests – Those tests performed as a part of conditional transfer instructions.

Loops – Program structures in which an instruction sequence repeats a number of times before exiting to other portions of the program.

Magnetic Card – See Card.

Magnitude – The numerical size of a number regardless of its sign.

Mantissa – The number in scientific notation which is to be multiplied by a given power of ten to equal the quantity desired.

Mantissa Format – Any given convention for selecting the number of significant digits to be displayed for the mantissa part of a number in scientific notation or for the number displayed when scientific notation is absent.

Memory – This generic term refers both to program memory and data memory. *Program memory* contains the program steps to solve a given problem and is addressed 000 through 223. *Data memory* consists of the twenty addressable registers, numbered 00 through 19. A register is usually designated by  $R_{nn}$ , where  $nn$  is the two-digit number of the addressable register. Its contents are denoted by a notation such as  $\bullet R_{14}$  (the contents of  $R_{14}$ ).

Memory Register – One of the twenty addressable storage areas in the data memory.

Modes (Display) – See Display Modes.

Modes (of Operation) – The calculate mode, run mode, and learn mode of using the SR-52.

NO-OP – See Null Instruction.

Notation – See Scientific Notation.

**Null Instruction** – An instruction which does nothing. This is used during step insertion in program editing. When the SR-52 is first turned on, program memory is filled with null instructions, which are zeros.



**Operand** – A number or numerical expression in a mathematical operation.

**Operation** – One of the four arithmetic functions (+, −, ×, ÷), or sometimes one of the two-variable functions  $y^x$  and  $\sqrt[x]{y}$ .

**Operator** – Any function that mathematically alters a number.

**Overflow** – The situation which results when the magnitude of a calculated or entered number exceeds  $9.999999999 \times 10^{99}$ . See Appendix B.

**Overwriting** – To replace the contents of a register with another value, obliterating the first quantity.

**Parentheses**   – Devices used to set off expressions as in algebra, to ensure they will be evaluated properly before being combined with other expressions.

**Pending Operations** – Those operations which cannot immediately be completed – pending evaluation of expressions opened by parentheses, or because of the algebraic hierarchy.

**Pointer** – A number which resides in data memory but which is used to specify a program address or another addressable register.

**Polar/Rectangular Conversions** – See Coordinate Transformations.

**Powers** – See Exponentiation.

**Printer** – An optional peripheral device for the SR-52 providing automated printing capability under program or keyboard control.

**Processing Registers** – See Internal Processing Registers.

**Program** – The logical sequence of keystrokes, which when stored and executed from program memory in the run mode, effects the solution of a problem.

**Program Coding** – To write down the step-by-step instructions of a program. Program code results from that program design process.

**Program Counter** – The internal device which keeps track of where the SR-52 currently resides in the instruction sequence.

**Program Flag** – See Flag.

**Program Instruction** – One or more key strokes which define an action to be taken in a program.

**Program Location** – Any of the 224 available positions in program memory.

**Program Memory** – 224 locations where a program can be stored. See Memory.

**Radian Mode** – See Angular Modes.

**Reading** – The process of loading a program from a magnetic card into the SR-52 program memory. Effected by the **2nd read** key.

**Recall** – To bring the value in a specified data register into the display register. Effected by the **RCL** key.

**Rectangular/Polar Conversions** – See Coordinate Transformations.

**Registers** – A generic term for any calculator storage unit which may be used to hold a numerical value. See Internal Processing Registers, Addressable Registers, Return-Pointer Registers.

**Reset**— To restore to zero; especially to restore a flag to zero. Also, the instruction which resets all flags, resets the return-pointer registers, and positions the program counter to 000. Effected by **2nd** **rset** .

**Return**— A transfer of control back to a calling program segment. Effected by **2nd** **rtn** .

**Return Pointer**— An indicator showing where to return control in a program after the processing sequence has been diverted to a subroutine.

**Return Pointer Registers**— The two registers which internally provide the return pointers for up to two subroutines.

**Root Extraction**— A built-in two-variable function for obtaining the xth root of y. Effected by  **$\sqrt[x]{y}$** .

**Rounded (Roundoff)**— To eliminate the least significant digits of a number and adjust the remaining digits to be as close as possible to the original number. e.g., the display register "rounds" from 12 to 10 digits for display.

**Run Mode**— The SR-52 type of operation in which execution is under program control.

**Scientific Notation**— The method for representing a number by a "mantissa" m (in the range  $1 \leq m < 10$ ) times a power of ten.

**Second Function**— The function or operation listed just above each key. To effect these "secondary" key meanings they are preceded by the **2nd** key.

**Single-Step**— The process of executing or observing a program one step at a time. Effected by the **SST** key.

**Store**— To place a replica of the contents of the display register into a specified addressable register.



**Subroutine** – An isolated program segment used primarily for repetitive calculations. It returns to the calling routine (either the main or another subroutine) upon completion of its task.

**Subroutine Return Pointer** – See Return Pointer.

**Top-Down** – The approach whereby a problem is solved in the large before details are filled in.

**Trace** – A printer capability for automatically recording each step executed and its result.

**Transfer Instructions** – Those instructions which can cause the program counter to be repositioned to a point other than that which would be reached by normal incrementing. There are two types of transfers:

**Unconditional transfers** *always* reposition the program counter to some out-of-sequence location. **Conditional transfers**, or branching instructions, make a test and either transfer or not (fall through) depending upon the outcome of the test.

**Unconditional Transfer** – Program instruction which unquestioningly repositions the program counter to some out-of-sequence location. See Transfer Instructions.

**Underflow** – The situation obtained when a number is keyed in or is produced through computation whose magnitude is greater than zero but less than  $1 \times 10^{-99}$ . See Appendix B.

**User Defined Labels** – See Key, User-Defined.

**Writing** – The process of recording a program on a magnetic card.

**Writing over** – See Overwriting.

# INDEX

## A

AC adapter/battery charger . . . . .	173
A/C operation . . . . .	173
Accumulation in addressable registers . . . . .	56
Accuracy . . . . .	2, 19
Addition . . . . .	29, 30
Address . . . . .	88, 89
Addressable registers . . . . .	2, 33
Advantages of SR-52 . . . . .	2
Algebraic hierarchy . . . . .	46
Ambiguous expressions . . . . .	48
Angular mode . . . . .	7, 38
Antilogarithms . . . . .	36
Arccosine . . . . .	36
Arcsine . . . . .	36
Arctangent . . . . .	36
Arithmetic operations . . . . .	29, 55
Automatic display mode switching . . . . .	25
Automatic printing . . . . .	5, 136

## B

Backstep . . . . .	82
Basic Program Library . . . . .	8, 60
Battery charger/AC adapter . . . . .	173
Battery operation . . . . .	173
Battery pack replacement . . . . .	174
Blanking of display . . . . .	61
Branching . . . . .	67, 89
BST key . . . . .	82

## C

Calculate mode . . . . .	4, 10
Card, magnetic	
blank . . . . .	177
care of . . . . .	175
cleaning . . . . .	176
handling . . . . .	175
prerecorded . . . . .	8, 60
reading . . . . .	61
recording . . . . .	77, 177
writing on . . . . .	177
Cartesian coordinates . . . . .	44
Case statement . . . . .	131

## INDEX (continued)

CE key . . . . .	19, 23
Chain operations . . . . .	29
Change-sign operation . . . . .	19, 21
Changing instructions . . . . .	78
Charging, battery . . . . .	173
Clearing	
calculations. . . . .	5, 24
data memory . . . . .	55
entries . . . . .	19, 23
error conditions. . . . .	28, 183
program memory . . . . .	14
CLR key . . . . .	5, 24
CMs key . . . . .	55
Codes, instruction. . . . .	55, 78
Coding form . . . . .	141
Common logarithms. . . . .	35
Conditional transfers . . . . .	67, 89
Control flags . . . . .	95
Controlling display . . . . .	24
Controlling printer. . . . .	132
Conversions,	
degrees-minutes-seconds . . . . .	42
degree/radian. . . . .	41
polar/rectangular . . . . .	44
Coordinate transformations . . . . .	44
Correcting programs . . . . .	78

### D

Data memory . . . . .	2, 51
Data registers . . . . .	2, 51
Decisions. . . . .	88
Decrement and skip on zero . . . . .	98
Defective battery pack . . . . .	174
Degree-minute-second format . . . . .	42
Degree mode . . . . .	7, 38
Degree/radian conversion . . . . .	41
DEL key . . . . .	81
Deleting instructions . . . . .	81
Direct register arithmetic. . . . .	55
Discharged battery . . . . .	173
Display	
erratic . . . . .	173
format . . . . .	25
register. . . . .	24, 49

## INDEX (continued)

Division . . . . .	29, 30
Division to memory . . . . .	55
D.MS key . . . . .	41
D/R key. . . . .	41
DSZ key . . . . .	98
Dummy memory operations . . . . .	58


### E

Editing programs . . . . .	78
EE key . . . . .	20
EE key, additional effects . . . . .	21
Equals key . . . . .	5, 34
Equals, when to avoid . . . . .	27
Error conditions. . . . .	28, 33, 181
Errors, tests for . . . . .	89
$e^x$ function . . . . .	36
EXC key . . . . .	57
Exchange instruction . . . . .	57
Execution, order of . . . . .	46
Executing a program . . . . .	64
Exponent. . . . .	20
Exponential functions . . . . .	36, 39
Expressions . . . . .	
ambiguous . . . . .	48
parenthetical . . . . .	31
Extracting roots. . . . .	39

### F

Factorial functions . . . . .	35
Failure to clear return pointers . . . . .	110
Fix key . . . . .	25
Fixed point display . . . . .	25
Flags. . . . .	3, 95
Flashing display. . . . .	21, 28, 33, 38, 178, 181
Flowchart. . . . .	69, 137, 158
Format, degree-minute-second. . . . .	43
Functions, effect of . . . . .	36
Functions, key . . . . .	2
Functions: one variable . . . . .	35
Functions: two variables . . . . .	39
Functions: trigonometric. . . . .	35

## INDEX (continued)

<b>G</b>		
GTO key . . . . .		88
<b>H</b>		
Halt instruction . . . . .		75
Hazard of the  key . . . . .	27, 106	
Head cleaning card . . . . .		177
Hierarchy, algebraic . . . . .		46
HLT key. . . . .		75
Hours-minutes-seconds conversion . . . . .		43
<b>I</b>		
If err key . . . . .		89
If flg key . . . . .		89
If pos key . . . . .		89
If zro key . . . . .		89
Improper arguments. . . . .		37
IND key. . . . .		115
Indirect addressing . . . . .		115
Initial display mode . . . . .		24
Initial mantissa format . . . . .		25
INS key. . . . .		82
Inserting instructions . . . . .		82
Instruction codes . . . . .		78
Instructions, program		
changing . . . . .		78
displaying . . . . .		82
keying in . . . . .		76
recording. . . . .		77
Internal processing registers . . . . .		31
Interrupting a program. . . . .		75
INV key . . . . .	26, 36	
Inverse functions . . . . .	26, 36	
<b>K</b>		
Key codes . . . . .		78
Keying in exponents of ten . . . . .		20
Keying in numbers. . . . .		18
Keys . . . . .		2, 7
<b>L</b>		
Labels . . . . .		2, 72

## INDEX (continued)

LBL key . . . . .	72
Learn mode . . . . .	4
Levels of parentheses . . . . .	32
Levels of routines . . . . .	3, 66
LIST key . . . . .	133
Listing a program . . . . .	133
Inx key . . . . .	35
Load module, program. . . . .	153, 163
LOG key . . . . .	35
Logarithms . . . . .	35
LRN key . . . . .	12

### M

Magnetic card (See Card, magnetic)	
Maintenance . . . . .	173
Mantissa . . . . .	20
Mantissa format. . . . .	25
Manual problem solving . . . . .	5
Memory, data . . . . .	4, 51
Memory, program . . . . .	4, 61
Mistakes, correcting . . . . .	78
Mode, display . . . . .	24
Modes, calculate, learn, run . . . . .	4
Multiplication . . . . .	29, 30
Multiplication to memory. . . . .	55

### N

Natural logarithms. . . . .	35
Negative exponents . . . . .	21
Negative numbers . . . . .	19
Nested parentheses . . . . .	46
Null instruction . . . . .	78, 81
Number entry . . . . .	18
Numeral keys . . . . .	7, 18, 79

### O

OFF/ON switch . . . . .	5
Operands. . . . .	33
Operations, pending. . . . .	33
Order of operations . . . . .	48
Overflow . . . . .	21, 28, 181

## INDEX (continued)

### P

PAP key . . . . .	135
Parentheses . . . . .	31
Pending operations . . . . .	33
Pi ( $\pi$ ) . . . . .	19
Pointer . . . . .	109
Polar/rectangular conversions . . . . .	44, 86
Powers, raising numbers to . . . . .	39
Printing . . . . .	3, 132
P/R key. . . . .	44
Processing registers. . . . .	2
PROD key. . . . .	55
Program card (See Card, magnetic)	
Program, corrections to . . . . .	78
Program counter . . . . .	60, 66
Program flags. . . . .	95
Program, keying in . . . . .	12, 76
Program memory . . . . .	4, 12, 76
Program, recording . . . . .	77
Program steps . . . . .	12
Program, writing . . . . .	65
Programming . . . . .	65
Programming style . . . . .	71
PRT key . . . . .	134

### R

Radian/degree conversion . . . . .	42
Radian mode . . . . .	38
Raising numbers to powers . . . . .	39
Range of display . . . . .	21
Range of function arguments. . . . .	38, 181
RCL key . . . . .	53
Reading a magnetic card. . . . .	61
Recalling from memory . . . . .	53
Reciprocal function . . . . .	35
Recording program on a card . . . . .	77
Rectangular (Cartesian) coordinates . . . . .	44
Rectangular/polar conversions. . . . .	45
Register arithmetic . . . . .	55
Registers, addressable. . . . .	51
Replacement of display register value . . . . .	49, 57
Return-pointer . . . . .	109
Return-pointer registers . . . . .	107

## INDEX (continued)

Root extraction . . . . .	35, 39
Rounding of display . . . . .	19, 25
RSET key . . . . .	97
RTN key . . . . .	107
RUN key . . . . .	12, 75
Run mode . . . . .	4, 10, 60
Running a program . . . . .	60

<b>S</b>	
SBR key . . . . .	101
Scientific notation . . . . .	20
Scientific notation removal . . . . .	26
Second function . . . . .	7
Service . . . . .	173
Sine . . . . .	35
Single-step . . . . .	82
Single-step execution . . . . .	82
Shipping instructions . . . . .	180
Spherical coordinates . . . . .	84
Square roots . . . . .	35
Squaring . . . . .	35
SST key . . . . .	82
Steps, displaying . . . . .	78
STO key . . . . .	51
Storing to memory . . . . .	51
Subroutines . . . . .	101
Subroutine return-pointer . . . . .	109
Subtracting from memory . . . . .	55
Subtraction . . . . .	29
SUM key . . . . .	55
Summing to memory . . . . .	55

<b>T</b>	
Tangent . . . . .	35
Ten, powers of . . . . .	20
Tests . . . . .	89
Top-down problem solving . . . . .	66
Trace . . . . .	136
Transfer instructions . . . . .	67, 88
Trigonometric functions . . . . .	35
Two-variable functions . . . . .	39



## INDEX (continued)

<b>U</b>	
Unconditional transfer . . . . .	67, 88
Underflow . . . . .	21, 28, 181
User-definable labels . . . . .	72
User-defined keys . . . . .	72

<b>W</b>	
Warranty . . . . .	back cover
Writing a program . . . . .	12, 65, 137, 155
Writing onto magnetic card . . . . .	77
Writing over display register contents . . . . .	49

<b>X</b>	
$x^2$ key . . . . .	35
$\sqrt{x}$ key . . . . .	35
$x!$ key. . . . .	35

<b>Z</b>	
Zero, division by. . . . .	28, 181
Zero instruction code . . . . .	78

© 1980 Joerg Woerner  
Datamath Calculator Museum

NOTES

© 2010 Joerg Woerner  
Datamath Calculator Museum

Texas Instruments reserves the right to make changes in materials and specifications without notice.

© 2010 Joerg Woerner  
Datamath Calculator Museum

## ONE-YEAR LIMITED WARRANTY

### WARRANTEE

This Texas Instruments electronic calculator warranty extends to the original purchaser of the calculator.

### WARRANTY DURATION

This Texas Instruments electronic calculator is warranted to the original purchaser for a period of one (1) year from the original purchase date.

### WARRANTY COVERAGE

This Texas Instruments electronic calculator is warranted against defective materials or workmanship. **THIS WARRANTY IS VOID IF: (i) THE CALCULATOR HAS BEEN DAMAGED BY ACCIDENT OR UNREASONABLE USE, NEGLIGENCE, IMPROPER SERVICE OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIAL OR WORKMANSHIP, (ii) THE SERIAL NUMBER HAS BEEN ALTERED OR DEFACED.**

### WARRANTY PERFORMANCE

During the above one (1) year warranty period your calculator will either be repaired or replaced with a reconditioned model of an equivalent quality (at TI's option) when the calculator is returned, postage prepaid and insured, to a Texas Instruments Service facility listed below. In the event of replacement with a reconditioned model, the replacement unit will continue the warranty of the original calculator or 90 days, whichever is longer. Other than the postage and insurance requirement, no charge will be made for such repair, adjustment, and/or replacement.

### WARRANTY DISCLAIMERS

**ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE ONE (1) YEAR PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR LOSS OF USE OF THE CALCULATOR OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE PURCHASER.**

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

### LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

## TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

Texas Instruments Service Facility  
P.O. Box 2500  
Lubbock, Texas 79408

Texas Instruments Service Facility  
41 Shelley Road  
Richmond Hill, Ontario, Canada

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information:

Texas Instruments Consumer Service  
3185 Airway Drive, Bldg J  
Costa Mesa, California 92626  
(714) 540-7190

Texas Instruments Consumer Service  
10700 Southwest Beaverton Highway  
Park Plaza West, Suite 111  
Beaverton, Oregon 97005  
(503) 643-6758

## CONSUMER RELATIONS DEPARTMENT

If you have questions or need assistance with your calculator, write the Consumer Relations Department at: **Texas Instruments Incorporated, P.O. Box 22283, Dallas, Texas 75222.** Or, call Consumer Relations at 800-527-4980 (toll-free within all contiguous United States except Texas) or 800-492-4298 (toll-free within Texas). If outside contiguous United States call 214-238-5461. (We regret that we cannot accept collect calls at this number.)

**TEXAS INSTRUMENTS**  
INCORPORATED  
DALLAS, TEXAS