ASTROCAL

ASTROCAL is a highly advanced calculator. Beyond a mere calculator, ASTROCAL is a true problem-solving machine. The problem-solving capability comes from the unique combination of programmability and programming method.

ASTROCAL provides ready access to the problem-solving machinery —— the programs. Conveniently stored on disks, 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 enables 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 ASTROCAL is not only easy but also fun. If you are experienced users of computers, you can enjoy many features previously available only on full-scale computers.

Enjoy and have fun,

Vernon B. Hester

Third printing January 1999
Fourth printing April 2001
Fifth printing August 2010

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

FEATURES

10,000 Program Storage Locations — Simply place ASTROCAL in the program mode and it remembers up to 10,000 calculation instructions and numbers that can be repeated on your command.

85 Labels — These labels permit quick identification or transfer to any program segment.

25 Preprogrammed Key Functions — Trigonometric and logarithmic functions, powers and roots, factorials, reciprocals, conversions, and π are available directly from the keyboard.

100 Addressable Memory Registers — Store and recall data or perform direct register arithmetic: addition, subtraction, multiplication, or division with any addressable memory register without affecting the calculation in progress.

User Selectable Precision — All registers, internal and addressable, provide the selected digit numbers with power-of-ten exponent. The selected digits are 119, 79, 39, 21, 19, and 17. Model I and Model III are 119, 63, 39, 19, and 17.

26 User Definable Keys — These enable user-definable functions to be executed simply by pressing the appropriate upper case letter key.

85 Internal Processing Registers — These are used to hold operands for calculations in progress. MAX-80 has 69 Internal Processing Registers, and the number of Internal Processing Registers for the Model I and Model III:

High Memory Usage	Diskette Configuration	Processing Registers
Nothing	Single-sided diskettes	39
Nothing	Double-sided diskettes	23
Printer Filter	Single-sided diskettes	35
Printer Filter	Double-sided diskettes	19

10 Logical Decision Functions — Program ASTROCAL to make repetitive decisions and transfer to appropriate program segments without interruption.

10 Program Flags — These flags can be set, reset, and tested under program control.

73 Program Levels — 72 levels of subroutines can 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 enable programs containing up to 85 (MAX-80 = 69, and see above for the Model I/III) pending operations and up to 36 open parentheses to be programmed and entered in the same way they are normally written.

Permanent Program Storage on Disk — Save your programs on disk for quick and easy retrieval.

Printing — Print the results of your calculations for permanent records. Or print a listing of your program for verification of the instructions keyed in.

MODES OF OPERATION

ASTROCAL can be operated in three different modes: calculate, program, and execute.

When you initialize ASTROCAL it is in the calculate mode. You will find that once you have mastered the use of the calculate mode you are well on you way to mastering the program mode as well.

Calculate Mode — Here you manually operate ASTROCAL as a general-purpose calculator. You command each and every instruction by keystrokes: entering numbers, performing mathematical operations, computing functions, and storing or recalling intermediate results or other data. In the calculate mode you can also single-step each instruction to cause actual execution of the stored program, one instruction at a time. The single-step execution is primarily used to assist you in program checkout and debugging.

Program Mode — After defining the instructions you would use in the calculate mode to solve a problem, you can key these instructions directly into ASTROCAL's program memory for immediate use in the execute mode. It is also possible to step through program memory, and display the instructions for editing purposes. This feature will soon encourage you to use the program and execute modes to solve even short problems that 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 of manually keying in your program each time it is needed, you can save your program on a disk for future use.

Execute Mode — You can use a program designed by yourself or by someone else to solve a problem. Loading a saved program into program memory enables you to tailor ASTROCAL to perform the instructions 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 execute mode save you the labor of remembering and executing keystrokes; it makes problems solvable to a precision that previously required a full-scale computer. As a subset of the execute mode, you can trace the program with the instructions and the display-register contents available for review.

You can permanently record calculator activity by performing printing functions in the calculate, program, or execute modes. In the calculate mode, you can print any desired intermediate results. In the program mode, you can print a complete listing of the stored program. And in the execute mode, print instructions encountered in the program cause automatic printing of the quantity just computed. Paper advance instructions can be used to set off groups of data. These printing features enable you to execute a program without the necessity of program halts that would be required to record multiple answers.

MANUAL CALCULATIONS

By now you are eager to use ASTROCAL. When you initialize ASTROCAL, a menu of register/display sizes are provided for your selection (the display size does not have the same number of digits as the display-register). Press one of the number keys **1** through **6** (I am presenting all of the examples in this manual using ASTROCAL[6] —— the **6** key). Once you have pressed an appropriate key, a single zero should appear in the display. If anything other than a single zero appears, press **CLEAR**. You may bypass the menu by keying ASTROCAL n<**ENTER**> (where n is 1 through 6). If you press <**BREAK>** at the menu prompt, then you will exit ASTROCAL.

The following examples will familiarize you with the basic operations of ASTROCAL. It is a safe practice to press **CLEAR** prior to beginning any new problem to be sure all incomplete calculations is eliminated. However, when a problem ends with an **=**, you can enter a new program without using **CLEAR**.

Example: 67.33 + 34.223 - 3.2 = ?

ENTER	PRESS	DISPLAY
67.33	+	67.33
34.223	-	101.553
3.2	=	98.353

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

Example: 23 x 4.238 ÷ 0.056 = ?

ENTER	PRESS	DISPLAY
	CLEAR	0
23	*	23.
4.238	/	97.474
.046	=	2119.

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: (8 + 7) x 3 = ?

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
	(	0.	
8	+	8.	
7	)	15.	Evaluates contents in parentheses.
	*	15.	
3	=	45.	

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

Example: $\log_e 7 = ?$

ENTER	PRESS	DISPLAY	REMARKS
7	n	1.945910149	Note: lower case **n**. Also only the first ten significant digits are displayed (ASTROCAL[6]).

To compute a trigonometric function, it is first necessary to select whether you plan to measure angles in degrees or radians. The current selection is displayed in the upper right on the screen. To change the selection press **z**.

Example: Sine of 30° = ?

ENTER	PRESS	DISPLAY	REMARKS
30	s	0.5	Angle selection "Degrees"

Note that this operation was basically the same as the previous example except for the additional step of selecting the angular mode. Later on I will show you how to have a program select the angular mode.

You have noticed that the lower case **n** and **s** were pressed in the last two examples. This implies that the letter keys have two functions or meanings. The mathematical or special function of the keys utilizes the lower case letter. Whereas the upper case letters are reserved for user-defined functions. Also, for the Model 4, the function keys **F1**, **F2**, **F3**, **RIGHT·SHIFT·F1**, **RIGHT·SHIFT·F2**, and **RIGHT·SHIFT·F3**, are programmed to be **A**, **B**, **C**, **D**, **E**, and **F** respectively. This does not apply to a Model III operating system running on a Model 4.

EXECUTING PROGRAMS

Now you have come to the most interesting and useful aspect of ASTROCAL —— its ability to execute a program. This is, after all, what ASTROCAL is all about —— executing programs to solve problems. To demonstrate this mode I have selected ANNUITY/CAL that is provided on your ASTROCAL disk. But, before you load this program, be sure you have selected ASTROCAL[6]. ASTROCAL[6] is selected from the ASTROCAL menu. You return to the ASTROCAL menu by pressing **SHIFT·BREAK** twice. The first time you press **SHIFT·BREAK**, a flashing "BREAK" appears. If you press any key other than **[SHIFT·]BREAK**, then the keystroke is ignored and the flashing "BREAK" disappears. By pressing **SHIFT·BREAK** with the flashing "BREAK" displayed, you are returned to the ASTROCAL menu. If you press <**BREAK**> with the flashing "BREAK", then you will exit ASTROCAL. Once you are in the ASTROCAL menu, press the **6** key to initialize ASTROCAL[6].

Example: Ordinary Annuity Computations. Given three of four variables as inputs, solve for the remaining variable in the annuity equations.

Present value: **PV** = MPMT x $(1-(1+I\div1200)^{-(N\times12)})\div(I\div1200)$
Monthly payment: **MPMT** = PV x $(I\div1200)\div(1-(1+I\div1200)^{-(N\times12)})$
Years: **Y** = $-\log_e(1 - PV \times ((I\div1200)\div MPMT))\div\log_e(1+I\div1200)\div12$
The interest, **I**, rate is solved using an iterative method.

To solve this problem, enter the program mode by pressing **CTRL·p** (when you see a key sequence similar to **CTRL·p**, this means press and hold the CTRL key; and, while holding this key down with one appendage, also press the P key). For the Model I and Model III, **SHIFT·↓** is the control key.
You should see:

```
                                                        PROG  Degrees
  Location  Code  Instruction


> 0000    000
  0001    000
  0002    000
  0003    000
```

Now press **CTRL·l**. You will then be prompted with a "Filespec:" prompt. Key in annuity/cal, then press <ENTER>. (NOTE: Whenever you enter the program mode, ASTROCAL resets the CAPS mode (you will be in lower-case). Also ASTROCAL converts the case, if necessary, when you enter a filespec.) After the program is loaded into ASTROCAL's program memory, press **CTRL·p** to return to the calculate mode. Now you are ready to solve some annuity problems.

The instructions for using the program are:

STEP	PROCEDURE	PRESS	DISPLAY
1	Initialize	Z	0.
2	Input 3 of 4 arguments (in any order):		
	Present value	V	PV
	Payment per month	O	MPMT
	Number of years	Y	Y
	Annual interest rate in percent	I	I
3	Compute unknown value:		
	Present value	U	PV'
	Payment per month	P	MPMT'
	Number of years	T	Y'
	Annual interest rate in percent	J	I'
4	To solve a new problem go to STEP 2. You need to enter only those variables that change.		

Are you wondering: How do I go from the calculate mode to the execute mode? The answer is every time you press upper case A through Z (the user-definable label keys) ASTROCAL changes from the calculate mode to the execute mode and begins executing the instructions in the program memory following the label. When ASTROCAL has completed execution, ASTROCAL returns to the calculate mode.

Example: Calculate the monthly payments on a 30-year basis for a $40,000 loan at an 8.75% annual interest rate. Also calculate the payments for this loan on a 20-year basis.

ENTER	PRESS	DISPLAY	REMARKS
	Z	0.	Initialize (upper case Z)
30	Y	30.	Y
40000	V	40000.00	PV
8.75	I	8.75	I
	P	314.68	MPMT' (30 years)
20	Y	20.	Y_1
	P	353.48	$MPMT'_1$ (20 years)

Example: What is the annual interest rate of a $45,900 mortgage, for 30 years, with $361.10 monthly payments?

ENTER	PRESS	DISPLAY	REMARKS
45900	V	45900.00	PV
30	Y	30.	Y
361.10	O	361.10	MPMT
	J	8.75	I'

To solve these examples, it was not necessary for you to perform all the detailed keystrokes necessary to set up and execute the problem in the calculate mode. You don't even have to know the equations used. Please press **SHIFT·Z** to clear ASTROCAL before you go to the next section.

CREATING YOUR OWN PROGRAMS

The program mode enables you to create programs by entering them directly into program memory. The basic operating instructions to accomplish this are simple: Enter the program mode by pressing **CTRL·p** and define the program by keying in the proper sequence of instructions. When your program is complete in program memory, you can save it on a disk. Transfer out of the program mode by pressing **CTRL·p** again. However, there is a certain amount of activity that is necessary prior to your keying in the program. This is the phase where you determine your objectives, assemble the equations you need, and finally write down the actual program instructions.

Example: In a compound-interest problem, what is the future value given the other three variables as input quantities? There is no single program sequence for this problem. I will discuss two different ways it can be achieved.

METHOD 1

The easiest way to write a program to solve future value, FV, is just to copy the keystrokes that would be used in the calculate mode, except that where you would enter data, a HALT instruction, **BREAK**, would be used to allow entry of data from the keyboard at that point. Execution would then be resumed by pressing **SHIFT·ENTER**, Execute. From the equation for future value, $FV = PV \times (1+(i \div 1200))^{n}$, the key sequence in the calculate mode would be as follows:

ENTER	PRESS
Present Value (PV)	*(
1	+(
Annual interest rate(i)	/
1200	))→
Number of periods(n)	=

Verify that the above sequence correctly solves for the future value with PV = 500, i = 5.75% annually, and n = 24 months.

ENTER	PRESS	DISPLAY	REMARKS
500	*	500.	PV
	(	500.	
1	+	1.	
	(	1.	
5.75	/	5.75	i
1200	)	.0047916667	i ÷ 1200
	)	1.004791667	1 + i ÷ 1200
	→	1.004791667	
24	=	560.7826821	FV

A program that is a literal automation of the foregoing calculate mode keystrokes would be the following:

KEY SEQUENCE	INSTRUCTION	REMARKS
SHIFT·@	LABEL	(Giving ASTROCAL a place to go when
SHIFT·K	K	you press K to solve the problem.)
:	x	: or * may be used for multiply
(	(	
1	1	
;	+	; or + may be used for add
(	(	
BREAK	HALT	to allow entry of i
/	/	divide
1	1	
2	2	
0	0	
0	0	
)	)	
)	)	
→	Exponent	
BREAK	HALT	to allow entry of n
ENTER	=	= or **ENTER** can be used for equal
BREAK	HALT	to display the answer

Note how I have labeled this program at the beginning so ASTROCAL will find its destination when **K** is pressed.

Now key this program into ASTROCAL's program memory. First, press **CTRL·p** to enter the program mode. Second, press **SHIFT·↑** to position the program counter to position 0000. Now press **CTRL·t** to clear the program already in program memory. (Note: If you had saved this program, and now are loading it from disk, the load function automatically performs the resetting [set to 0000] of the program counter and the clearing of program memory.) Carefully key in the instructions listed above. If you make a mistake, press **SHIFT·↑**, **CTRL·t** and start over again. (Later you will learn how to correct mistakes without starting over.) At the conclusion of keying in the program instructions you should see:

```
                                                          PROG  Degrees

  Location  Code  Instruction
    0016    002    HALT
    0017    061      =
    0018    002    HALT
  > 0019
    0020
    0021
    0022
```

This indicates the next instruction would go into location 0019, currently empty. To enable execution of the program, transfer from the program mode to the calculate mode by pressing **CTRL·p**. This now enables you to execute the program you have keyed into program memory. The instructions for using this program are as follows:

STEP	ENTER	PRESS	REMARKS
1	PV	SHIFT·K	Begins program execution
2	i	SHIFT·ENTER	Resumes execution
3	n	SHIFT·ENTER	Resumes execution

Try the case already worked out (remember upper case K), with PV = 500, i = 5.75, and n = 24, with the answer being 560.7826821.

This program is not as easy to use as ANNUITY/CAL. Obviously data had to be entered in a definite order. And, not so obvious, the calculator cannot be used for intermediate calculations during program halts. The next method demonstrates improved programming techniques to solve the interest problem.

METHOD 2

In this method I will design the program much more along the lines of ANNUITY/CAL. I will use the labels P, I, and N to store the present value, interest rate, and number of compound periods respectively into selected addressable memory registers. After the data is entered, the calculation of future value, at label K, recalls those data from addressable memory as needed. I have to decide which addressable memory registers to use for PV, i, and n. For simplicity, I will select register 01 for PV, register 02 for i, and register 03 for n. The segments of code necessary to store the data into addressable memory are the following:

KEY SEQUENCE	INSTRUCTION	REMARKS
SHIFT·@	LABEL	
SHIFT·P	P	
K	STO MEM	These three keys store PV into register 01.
0	0	
1	1	
BREAK	HALT	
SHIFT·@	LABEL	
SHIFT·I	I	
K	STO MEM	These three keys store i into register 02.
0	0	
2	2	
BREAK	HALT	
SHIFT·@	LABEL	
SHIFT·N	N	
k	STO MEM	These three keys store n into register 03.
0	0	
3	3	
BREAK	HALT	

I still need to define the program segment to calculate FV from the quantities in the addressable memory registers. This is achieved like the program in Method 1, except for the addressable memory reference.

KEY SEQUENCE	INSTRUCTION	KEY SEQUENCE	INSTRUCTION
SHIFT·@	LABEL	/	/
SHIFT·K	K	1	1
m	RCL MEM	2	2
0	0	0	0
1	1	0	0
:	x	)	)
(	(	)	)
1	1	→	Exponent
;	+	m	RCL MEM
(	(	0	0
m	RCL MEM	3	3
0	0	ENTER	=
2	2	BREAK	HALT

INTRODUCTION

To use the METHOD 2 program, first go to the program mode by pressing **CTRL·p**. Clear program memory by pressing **SHIFT·↑**, and **CTRL·t**. Now key in all instructions beginning with **SHIFT·@** in the program segment labeled **P**, and ending with **BREAK** in the program segment labeled **K**. If you performed these steps properly you will see the program counter positioned at location 0044, indicating that the next free program location is 0044.

Now press **CTRL·p** 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	P	PV
	annual interest rate(%) i	I	i
	number of periods n	N	n
2	Calculate future value	K	FV

Sample test:

ENTER	PRESS	DISPLAY	REMARKS
5.75	I	5.75	i (note order of input)
24	N	24.	n
500	P	500.	PV
	K	560.7826821	FV

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

PRECISION AND ACCURACY

Before we go on to the next section, let us discuss two important facts: **PRECISION** and **ACCURACY**.

The precision of ASTROCAL is selected from the ASTROCAL menu. The available precision also determines the display sizes. The number of digits in the display is represented in the following sections as [DISPLAY SIZE]. The precision/display sizes are: 119/72 (119/56 Model I/III) for ASTROCAL[1], 79/72 (63/56 Model I/III) for ASTROCAL[2], 39/32 for ASTROCAL[3], 21/14 for ASTROCAL[4] (n/a Model I/III), 19/12 for ASTROCAL[5], and 17/10 for ASTROCAL[6]. The display-register sizes are the same as the precision: 119, 72(63), 39, 21(n/a), 19, and 17 digits respectively.

The accuracy of ASTROCAL is typically one digit less than the selected precision when a non-linear math function (other than add, subtract, multiply, or divide) is used. The accuracy will typically, not necessarily, lose one digit per non-linear math function (this is why the display size is less than the selected precision —— to display an accurate result).

ENTERING AND DISPLAYING NUMBERS

Now I will return to basics and begin a more thorough description of ASTROCAL's operation. Remembering that generally any description that pertains to calculate mode keystrokes also applies when the keystrokes are made in the program mode and processed in the execute mode.

ENTERING NUMBERS

Any number up to [DISPLAY SIZE] digits can be keyed in by using the number keys and the decimal key. The procedure for entering a positive number is simply to press the keys in sequence exactly as the number appears.

Example: Enter 305.5050463

ENTER	PRESS	DISPLAY
	CLEAR	0
305.5050463		305.5050463

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

Example: Enter 9358

ENTER	PRESS	DISPLAY
	CLEAR	0
9358	-	9358.

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

Example: Enter 0.02345

ENTER	PRESS	DISPLAY
	CLEAR	0
.02345		0.02345

Negative numbers are entered just like positive numbers except that the change-sign (+/-) key, **o**, is pressed as the final step of entry.

Example: Enter -9358.02345

ENTER	PRESS	DISPLAY
	CLEAR	0
9358.02345	o	-9358.02345

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

As you enter a number in the calculate mode, that number is shown on the display, including the decimal point and sign. If you make a mistake, simple press the clear-digit/error key, **SPACE,** to remove (backspace) the last digit entered. If more than [DISPLAY SIZE] digits are attempted in number entry, all digits after [DISPLAY SIZE] are ignored.

ENTERING AND DISPLAYING NUMBERS

ENTERING π

As a convenience, ASTROCAL provides a means of entering π by pressing a single key, **p**. Use of this key to enter π turns out to be more than just a convenience. A full register size representation of π results from this operation. (You are restricted to only [DISPLAY SIZE] when the manual number entry method is used.) The display always rounds to [DISPLAY SIZE] or fewer digits, so the extra digits are not visible, *but they are carried in all subsequent calculations*.

Example: Enter π

ENTER	PRESS	DISPLAY
	p	3.141592654

The value of π used in ASTROCAL[6] is 3.1415926535897932 even though the display only shows the value rounded to ten significant digits.

SCIENTIFIC NOTATION

To enter very large or very small numbers you must 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 number is referred to as the mantissa, and the second, 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 key, **e**, and finally key in the power-of-ten.

Example: Enter 4.243×10^{109}

ENTER	PRESS	DISPLAY
	CLEAR	0
4.243	e	4.243 0000
109		4.243 0109

Regardless of how the number is entered, ASTROCAL normalizes the number, displaying a single digit to the left of the decimal when any operation key is pressed.

Example: Enter 4243×10^{106}

ENTER	PRESS	DISPLAY
	CLEAR	0
4243	e	4243 0000
106		4243 0106
	+	4.243 0109

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 -5.781×10^{-86}

ENTER	PRESS	DISPLAY
	CLEAR	0
5.781	o e	-5.781 0000
86	o	-5.781 -0086

If you wish to change the sign of the mantissa after the exponent has been entered, just press **. o** (decimal then **o**). If the numerical value of the mantissa needs to be changed, press **SPACE** to remove (backspace) any unwanted digits. Holding down the space bar will remove all digits. Also **CLEAR** may be pressed if you don't have any calculations pending.

Example: Change the number displayed in the last example to 5.781×10^{-86}

ENTER	PRESS	DISPLAY
	. o	5.781 -0086

The exponent size is limited to numbers no longer than four digits. This provides a range of power-of-ten from 10^{9999} to 10^{-9999}. Note, however, that ASTROCAL will experience **Too-small** whenever the calculations result in numbers less than 1×10^{-9999} and greater than zero, and **Overflow** whenever the calculations result in a number equal to or greater than 10×10^{9999} in magnitude. A flashing "ERROR" results from these situations.

ADVANCED EFFECTS AND USES OF THE ENTER-EXPONENT KEY

Pressing the ENTER-EXPONENT key, **e**, causes several things to take place:

1. The display mode is set to scientific notation.

2. The four-digit registers for holding the exponent are initialized to zero unless the number in the display-register has an assigned exponent.

3. Number key entries immediately following **e** along with the pre-existing value of the exponent (if any) are interpreted so that the last four digits entered form the exponent. The earlier entered digits are discarded.

4. The rounded value of the mantissa actually shown in the display is loaded into the display-register for subsequent calculations. This can be a useful feature.

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

Example: Solve the problem $(3 \times 10^{82}) \times (6 \times 10^{46})$ and change the exponent of the results to 3.

ENTER	PRESS	DISPLAY
	CLEAR	0
3	e	3 0000
82	*	3. 0082
6	e	6 0000
46	=	1.8 0129
	e	1.8 0129
0		1.8 1290
0		1.8 2900
0		1.8 9000
3		1.8 0003

Observe that the first two times **e** was pressed following a number entry, the exponent digits were initialized to 0000. When **e** is used following a result, the exponent digits are unchanged and only the last four numbers entered are used as exponent digits.

I will use the π key to illustrate the fourth effect of the **e** key.

Example: π-π <> 0

ENTER	PRESS	DISPLAY
	CLEAR	0
	p -	3.141592654
	p e	3.141592654 0000
	=	-4.102068 -0010

As previously indicated, when π is used, ASTROCAL internally uses 3.1415926535897932. Pressing **p** then **e**, however, caused ASTROCAL to use only the displayed number —— discarding all digits not displayed. The calculation that took place was actually (3.1415926535897932 - 3.141592654) which equals -0.0000000004102068 or $-4.102068 \times 10^{-10}$. Note that this last effect of the **e** key affects only the number displayed at the time **e** is pressed.

CLEARING INCORRECT NUMBER ENTRIES

To clear an incorrect digit entry from the keyboard, press the clear-digit/error key, **SPACE**, and key the proper digit. **SPACE** clears only entries from the keyboard, not results of calculations, pending calculations, or π. There are other types of clearing operations for ASTROCAL and other keys for accomplishing them.

CLEARING A CALCULATION

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

DISPLAY CONTROL

ASTROCAL display gives you a controllable-format representation of the number in the display-register. The display-register like all other registers in ASTROCAL is 119, 79, 39, 21, 19, or 17 digits long (ASTROCAL[1-6] respectively). For display, this number is rounded to [DISPLAY SIZE] (72, 72, 32, 14, 12, or 10) or fewer digits. See page 10 for Model I and Model III sizes.

When ASTROCAL in initialized 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
	CLEAR	0
6	/	6.
3	=	2.
2	/	2.
6	=	.3333333333
200	/	200.
6	=	33.33333333

The number of digits displayed following the decimal point is as many as required within the maximum limit of [DISPLAY SIZE] digits to represent the number.

Whenever you enter a number in scientific notation or the number resulting from a calculation is less than $10^{-([DISPLAY\ SIZE]+1)}$ or equal to or greater than $10^{[DISPLAY\ SIZE]}$ in magnitude, the display automatically converts to scientific notation. The exponent is shown by four 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 four digits. The mantissa shown on the display is always in the range 1<=|mantissa|<10. A mantissa greater than this can be keyed in, however, 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 [DISPLAY SIZE] digits is the initial mantissa format.

Example: $(56.23 \times 10^{12}) \times 7 = ?$

ENTER	PRESS	DISPLAY	
56.23	e	56.23	0000
12		56.23	0012
	*	5.623	0013
7	=	3.9361	0014

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 of the digits that can be present in the initial mantissa format, you can cause all displayed results to be rounded to a fixed number of places following the decimal point by pressing the fix decimal key, **f**, and then enter the desired number of digits 00 to [DISPLAY SIZE]-1. The contents of the display-register is shown with the mantissa rounded to the desired number of digits, however, *all calculations use the full unrounded value*.

Example: 6 ÷ 7 = ? Round to two decimal places.

ENTER	PRESS	DISPLAY
	CLEAR	0
6	/	6.
7	=	.8571428571
	f 02	.86

The result is shown rounded to hundredths. Similarly, work the following example in scientific notation.

Example: $(6 \times 10^{12}) \div 7$ = Rounded to five decimal places.

ENTER	PRESS	DISPLAY	
6	e	6	0000
12	/	6.00	0012
7	=	8.57	0011
	f 05	8.57143	0011

There are two methods to return the display to the initial mantissa format. Pressing **f nn** where **nn** is => [DISPLAY SIZE], or pressing **@f** restores the display to the initial mantissa format without affecting scientific notation. The **@** key is the inverse function key, "Inverse".

To convert the display from scientific notation press **@e**. This does not affect pending operations or the number of digits carried. If the display is in Fix Dec. mode and the number is less than $5 \times 10^{-[\text{NUMBER OF FIXED DECIMALS}]}$ in magnitude, then the mantissa will display zeroes. If the number is equal to or greater than $10^{[\text{DISPLAY SIZE}]}$ regardless of Fix Dec., the display remains in scientific notation.

Example: $2 \div (4 \times 10^{4}) = ?$

ENTER	PRESS	DISPLAY	
	CLEAR	0	
	@ f	0.	
2	/	2.	
4	e	4	0000
4	=	5.	-0005
	f 03	5.000	-0005
	@ e	0.000	
	f 04	0.0001	
	f 05	0.00005	
	f 06	0.000050	

There are several approaches to convert to scientific notation. First you could simply press **e =**. Remember this method has consequence of loading the display-register with the rounded quantity displayed. The other methods involve multiplying/dividing the number by 1 **x** 10^0 or adding/subtracting 0 **x** 10^0 to/from the displayed number.

ENTER	**PRESS**
	*
1	e =

ENTER	**PRESS**
	+
0	e =

ENTER	**PRESS**
	/
1	e =

ENTER	**PRESS**
	-
0	e =

Because **=** completes all pending operations, do not use these methods in the middle of a computation. Pending operations and the effect of **=** are discussed later in this manual. To avoid this problem, use these conversion methods after computations are complete, or properly interject the method without the **=** keystroke.

ERROR CONDITIONS

Various error conditions during computation in either the calculate or execute modes result in a flashing "ERROR" displayed. The quantity in the display is usually a clue to the type of error involved. I will deal with each error condition in its appropriate place, listing only the two most common errors here.

Overflow —— Signifies a result whose magnitude is larger than the maximum that can be handled by ASTROCAL (=> 10 **x** 10^{9999}) and results in a flashing "ERROR". This can be interpreted as infinity as in the case of division by zero.

Too-small —— Indicates a result that is not zero but whose magnitude is too small to be handled by ASTROCAL (< 1 **x** 10^{-9999}). This condition also displays a flashing "ERROR".

The flashing "ERROR" can be removed by pressing the clear-digit/error key, **SPACE**, without affecting the displayed number unless the number was just keyed in prior to pressing **SPACE** —— in this case, the entire number is cleared. Also, entering the program mode removes the flashing "ERROR".

Under the condition when the resulting calculation is less than 10×10^{9999}, but the prevailing mantissa format would round the number to 10×10^{9999}; ASTROCAL will not cause "ERROR" to flash, displaying 10×10^{9999}. However, an **e** press would cause 1×10^{9999} to be set in the display-register.

Example: Overflow and 10×10^{9999}

ENTER	PRESS	DISPLAY		REMARKS
	CLEAR	0		
	@ f	0.		
	r	9.999999999	9999	"ERROR" flashing caused by overflow.
	SPACE	9.999999999	9999	"ERROR" stops flashing
	f 08	10.00000000	9999	improper mantissa but valid number.
	/	10.00000000	9999	
2	=	5.00000000	9999	validates number
	*	5.00000000	9999	
2	=	9.99999999	9999	"ERROR" flashing caused by overflow.
	SPACE	9.99999999	9999	"ERROR" stops flashing
	f 08	10.00000000	9999	to re-display mantissa
	e	10.00000000	9999	Stores 1×10^{9999} into display-register.
	/	1.00000000	9999	evidenced by an operation key press.

This is the only condition that the mantissa has 2 digits to the left of the decimal when the display is in scientific notation. Its purpose is to avoid an overflow error condition caused by rounding the display to less than [DISPLAY SIZE] digits when an actual overflow did not occur.

Please press **@f** and **CLEAR** before you move on to the next section.

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 the first number
Press **+ - *** or **/**
Enter the second number
Press **=**

Pressing **CLEAR** at the beginning of this sequence clears any calculations in progress.

Example: $(3.2 \times 10^{-23}) \times (4.125 \times 10^{56}) = ?$

ENTER	PRESS	DISPLAY
3.2	e	3.2 0000
23	o *	3.2 -0023
4.125	e	4.125 0000
56	=	1.32 0034

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: 3.42 + 2.04 = ? then (3.42 + 2.04) ÷ 256 = ?

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
3.42	+	3.42	
2.04	=	5.46	3.42 + 2.04
	/	5.46	
256	=	0.021328125	5.46 ÷ 256

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

A SPECIAL TYPE OF OPERATIONAL CHAIN

The foregoing discussion of 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. You may do this with two types of chains. These are:

1) Chains containing only **+** and **-** operations.
2) Chains containing only **x** and **÷** operations.

To evaluate expressions of this type, simply key in the numbers and operations keys the way the problem is written and finally press **=** to get the answer.

Example: 56 + 856 - 23 + 123 - 12 = ?

ENTER	PRESS	DISPLAY	REMARKS
56	+	56.	
856	-	912.	56 + 856
23	+	889.	56 + 856 - 23
123	-	1012.	56 + 856 - 23 + 123
12	=	1000.	

Example: 35 x 4.1 ÷ 7 x 12 x 9 = ?

ENTER	PRESS	DISPLAY	REMARKS
35	*	35.	
4.1	/	143.5	35 x 4.1
7	*	20.5	35 x 4.1 ÷ 7
12	*	246.	35 x 4.1 ÷ 7 x 12
9	=	2214.	

PARENTHESES

To introduce this subject, you should try the following experiment: Press **CLEAR (5 * 7)**, and you will see the displayed value 35. ASTROCAL has evaluated 5 **x** 7 and replaced it with 35, even though the **=** key was not pressed. This behavior is a consequence of the following operating characteristic designed into ASTROCAL: whenever an expression is set off by parentheses, that is ([expression]), ASTROCAL evaluates that expression then use its value in any larger expressions of which it is a part. This evaluation is properly carried out even if the expression set off by parentheses contains other expressions that are also set off by parentheses. As the keystroke sequence containing parentheses is processed, ASTROCAL stores into internal processing registers those operands that cannot yet be combined with other operands.

Starting with **(** does not usually require using **CLEAR** because the number entry replaces the number displayed when **(** is pressed. Examples in this manual beginning with **CLEAR (** include the clear operation for the convenience of showing the display contents for each step.

Example: (3+(4÷(7-(6÷(1+2))))) = ?

KEYSTROKES	**ASTROCAL ACTION**
(	Set up to evaluate expression.
3 +	Store 3 internally, marked for pending addition.
(	Set up evaluation for second-level parentheses.
4 /	Store 4 internally, marked for pending division.
(	Set up evaluation for third-level parentheses.
7 -	Store 7 internally, marked for pending subtraction.
(	Set up evaluation for fourth-level parentheses.
6 /	Store 6 internally, marked for pending division.
(	Set up evaluation for fifth-level parentheses.
1 +	Store 1 internally, marked for pending addition.
2)	Recognize 1 +2 may now be performed. Replace 1 +2 with 3.
)	Recognize 6 /3 may now be performed. Replace 6 /3 with 2.
)	Recognize 7 -2 may now be performed. Replace 7 -2 with 5.
)	Recognize 4 /5 may now be performed. Replace with 0.8
)	Recognize 3 + 0.8 may mow be performed. Replace with 3.8

Key in this example and notice the display as each right parenthesis is pressed.

When the first right parenthesis was encountered in this example, ASTROCAL had stored five operands, each associated with an operation pending, into the internal processing registers. Closing the final parentheses in this example caused the whole expression to cascade down to a single evaluated number, ASTROCAL recognizing at exactly what point each pending operation could be completed. The rule that results from all this is 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 with ASTROCAL just as you do in your analytical work, it also saves your addressable memory for purposes other than for storing operands that have operations pending. In making optimal use of the internal processing registers, the natural keystroke sequence with parentheses is easy and natural, and is efficient in memory usage. In addition, when you program ASTROCAL to deal with parenthetical expressions, you are obtaining the most efficient operation from an execution time point of view as well as the program code whose intent remains clear long after it is written.

There are limits on how many pending operations and operands can be entered into the internal processing registers. This limit, though large enough that you will probably never be aware that it exists, can accommodate as many as 85 pending operations containing up to 36 open parentheses. If you attempt to open more than 36 parentheses, "ERROR" will flash and the left parenthesis is ignored. If you attempt to create more than 85 operations pending, "ERROR" will flash and the operation will be ignored. See page 1 for the number of pending operations available for the MAX-80, and the Model I/III.

THE USE OF = IN COMPLICATED EXPRESSIONS

There is one other feature related to pending operations that you will find particularly important. The effect of pressing **=** in any calculation (or encountering **=** in any program) is to complete all pending operations. It does not matter that the right parentheses associated with existing left parentheses have not all been keyed, for the **=** key has the effect of immediately supplying as many right parentheses as necessary to complete the expression. Again, key in the last example, substituting an **=** for the five right parentheses.

Example: -12.5+((8.12 -8)÷(8.12 +8)) = ?

ENTER	PRESS	DISPLAY	REMARKS
12.5	o + ((	-12.5	
8.12	-	8.12	
8	) / (	.12	8.12 - 8
8.12	+	8.12	
8	=	-12.49255583	

The **=** supplied the last two right parentheses in this evaluation.

SPECIAL FUNCTIONS

The simplest operations of all to describe and understand are probably the single-variable functions of ASTROCAL. I will, therefore, first describe these functions, then go on to the functions of two variables, and finally discuss angular unit conversions (that 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.

KEY	FUNCTION	PROGRAM INSTRUCTION LEGEND
!	Factorial	!
c	Cosine	Cosine
l	Common logarithm ($\log_{10}$)	LOG 10
n	Natural logarithm ($\log_e$)	LOG e
q	Square	Square
r	Reciprocal	1/x
s	Sine	Sine
t	Tangent	Tangent
y	Cube	Cube

Hyperbolic cosine, sine, and tangent functions of the displayed quantity is obtained by pressing the hyperbolic function key, **h**, before the **c**, **s**, or **t** key. Hyperbolic cosine is abbreviated as cosh, hyperbolic sine is abbreviates as sinh, and hyperbolic tangent is abbreviates as tanh.

Example: 70! = ?

ENTER	PRESS	DISPLAY
70	!	1.197857167 0100

Example: $\log_{10}2$ = ?

ENTER	PRESS	DISPLAY
2	l	.3010299957

Example: 3^2 = ?

ENTER	PRESS	DISPLAY
3	q	9.

Example: sin 0.785*r* = ?

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
	z	0	"Radians" displayed
.785	s	.7068251811	

SPECIAL FUNCTIONS

In addition to the functions listed, the following inverse functions are also available by use of the inverse key, **@**, prefix:

FUNCTION	**PRESS**
10^x, antilogarithm (common)	@ l
e^x, antilogarithm (natural)	@ n
arc cosh	@ h c or h @ c
arc cosine	@ c
arc sine	@ s
arc sinh	@ h s or h @ s
arc tangent	@ t
arc tanh	@ h t or h @ t
cube-root	@ y
square-root	@ q

Example: $10^{6.7}$ = ?

ENTER	**PRESS**	**DISPLAY**
6.7	@ l	5011872.336

Example: arc cosh 3.8 = ?

ENTER	**PRESS**	**DISPLAY**
3.8	@ h c	2.010367491

Example: arc tan 1 = ?

ENTER	**PRESS**	**DISPLAY**
1	@ t	.7853981634

Example: $\sqrt{7}$ = ?

ENTER	**PRESS**	**DISPLAY**
7	@ q	2.645751311

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: $9 + \sqrt{28 + 8}$ = ?

ENTER	**PRESS**	**DISPLAY**
	CLEAR	0
9	+	9.
	(	9.
28	+	28.
8	)	36.
	@ q	6.
	=	15.

SPECIAL FUNCTIONS

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

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
28	+	28.	28 is stored with **+** pending.
8	-	36.	28 + 8 is completed and then stored with **-** pending.
	@ q	6.	6 replaces 36 in the display-register. Now to verify that the pending operation will not be affected.
12	=	24.	The final pending operation (subtraction) is completed giving 36 - 12 = 24.

Some of the single-variable functions have restrictions on the values of their arguments, in addition to those necessary to avoid overflow and too-small. When these restrictions are violated "ERROR" flashes. These restrictions are summarized here:

FUNCTION	ARGUMENT RANGE	DISPLAY
$\sqrt{x}$	x => 0	$\sqrt{\lvert x \rvert}$
arc cosh x	x => 1	x
arc cosine x	|x| <= 1	x
arc sine x	|x| <= 1	x
arc tanh x	|x| < 1	x
cosine x	$\lvert x \rvert < 10^{[\text{PRECISION}]}$	|x|
$\log_{10} x$	x > 0	$\log_{10}\lvert x \rvert$
$\log_e x$	x > 0	$\log_e \lvert x \rvert$
sine x	$\lvert x \rvert < 10^{[\text{PRECISION}]}$	|x|
tangent x	$\lvert x \rvert < 10^{[\text{PRECISION}]}$	|x|
x!	3249 > integers => 0	|INT(x)|!

Unlike the other functions, the factorial function operates only on the rounded number displayed, not upon the number of digits in the display-register.

Angular Mode Selection —— The trigonometric functions including the inverse trigonometric functions and coordinate conversion (Polar to Rectangular or Rectangular to Polar) involve knowledge about units. Are the angles expressed in units of degrees or units of radians? You have noticed the "Degrees" or "Radians" displayed in the upper right on the screen. The angular mode has absolutely no effect except when a trigonometric function or coordinate conversion is being performed. You may therefore change this mode at any intermediate point in a calculation. The mode selected is indicated for each problem in this manual that depends on the angular mode —— **ANGLE:Degrees** means set the mode to "Degrees" and **ANGLE:Radians** means set the mode to "Radians".

Example: sin 30° = ?

ANGLE:Degrees

ENTER	PRESS	DISPLAY
	CLEAR	0
	z	0
30	s	0.5

Example: arc sin .5 = ? in radians

ANGLE:Radians

ENTER	PRESS	DISPLAY
	CLEAR	0
	z	0
.5	@ s	.5235987756

As previously mentioned, you can program ASTROCAL to select the angular mode during program execution. The labels "(" and ")" in these examples are not critical; you may use any label not used in the balance of your program.

Code segment to force the angular mode to radians:

Location	Code	Instruction
0010	096	LABEL
0011	040	(
0012	049	1
0013	045	-
0014	064	Inverse
0015	116	Tangent
0016	061	=
0017	037	If pos
0018	041	)
0019	122	Sw MODE
0020	096	LABEL
0021	041	)

Code segment to force the angular mode to degrees:

Location	Code	Instruction	
0010	096	LABEL	
0011	040	(	
0012	049	1	
0013	045	-	
0014	064	Inverse	
0015	116	Tangent	
0016	061	=	
0017	064	Inverse	
0018	037	If pos	[% key]
0019	041	)	
0020	122	Sw MODE	
0021	096	LABEL	
0022	041	)	

FUNCTIONS OF TWO VARIABLES

ASTROCAL provides two functions of two variables. The first function is powers, accessed by the → key. The second is roots, accessed by the ← key. The rules for these two functions are essentially identical:

1. To raise **x** to the **y** power:
   [1] enter **x**
   [2] press →
   [3] enter **y**
   [4] press **=**

2. To take the **y** root of **x**:
   [1] enter **x**
   [2] press ←
   [3] enter **y**
   [4] press **=**

Example: $4.32^{-.94}$ = ?

ENTER	PRESS	DISPLAY
4.32	→	4.32
.94	o	-0.94
	=	.2527232966

Example: What is the 4.97th root of 17.23?

ENTER	PRESS	DISPLAY
17.23	←	17.23
4.97	=	1.773166643

Either the variable **x** or **y** or both can be the results of other computations that may involve parentheses. Furthermore, parenthetical expressions may also contain these functions.

Example: $((23.8 - 3.14)/18)^{(22/19)}$ = ?

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
	((	0.	
23.8	-	23.8	
3.14	) /	20.66	
18	)	1.147777778	value of x
	→	1.147777778	
	(	1.147777778	
22	/	22.	
19	)	1.157894737	value of y
	=	1.173029801	value of x^y

SPECIAL FUNCTIONS

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

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
	(	0.	
3	* (	3.	
4	→ (	4.	
2	→ (	2.	
7	←	7.	
4	)	1.626576562	the 4th root of 7
	o	-1.626576562	-(4th root of 7)
	)	.3238557891	2 raised to - 4th root of 7
	)	1.566681134	4 raised to 0.323···
	)	4.700043401	3 times 4 raised to .323···

The **=** was not used in this example. The enclosure of the whole expression with parentheses is sufficient to cause it to be evaluated, and the fact that power and root functions occur does not in any way alter the basic parentheses disciplines already described.

There is a restriction on these functions. If the variable x is negative, then the root function produces an error. The variable x must also be non-negative for powers if the exponent is not an integer. i.e., $-2.3^{2.2}$ produces an error, but -2.3^{12} does not (the sign of the result is negative if the exponent is odd; otherwise, the sign is positive). In either case, if an error is produced, the result will be the respective root or power of the absolute value of x, and "ERROR" will flash. For powers, if the y value is 0 and the x value is not zero, then the result is 1. For roots, if both the x and y values are 0, then the result is also 1, but "ERROR" flashes indicating a mathematical indeterminate form.

ANGULAR UNIT CONVERSION

Four additional single-variable functions are provided. These are not standard mathematical functions, but are angular-unit conversions, as shown below:

Function	Press	Effect
Degrees-to-radians	d	Multiplies displayed value by π/180.
Radians-to-degrees	@ d	Multiplies displayed value by 180/π.
Degrees-minutes-seconds to decimal degrees	b	Converts a number entered in the degree-minute-second format to decimal degrees.
Decimal degrees to degrees-minutes-seconds format	@ b	Converts a number from decimal degrees to the degree-minute-second

Example: Convert 122° to radians

ENTER	PRESS	DISPLAY
122	d	2.129301687

Example: Convert .5235987756*r* to degrees

ENTER	PRESS	DISPLAY
.5235987756	@ d	30.

The next two functions require the explanation of the degree-minute-second format. These functions enable 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 decimal fractional part of seconds are not required to be three as shown, but are limited only by the display size.

Example: Convert 56°32'8.12" to decimal degrees

ENTER	PRESS	DISPLAY
56.320812	b	56.56558889

Example: Convert 22.13666667° to degrees-minutes-seconds

ENTER	PRESS	DISPLAY
22.13666667	@ b	22.0812

The displayed answer is interpreted as 22°8'12"

All of the angular unit conversion functions operate independently of the angular mode. 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.

COORDINATE (POLAR/RECTANGULAR) CONVERSIONS

A special capability is provided by ASTROCAL to enable you to convert easily between polar and rectangular coordinates. This coordinate conversion does involve the addressable memory register 00.

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

	Before	After
Display-register	Θ	Y
Addressable memory register 00	R	X

As shown in this table, with the radius **R** in addressable memory register 00, AM00, and the angle in the display-register, the polar-to-rectangular conversion places the cartesian coordinate **X** into AM00 and the cartesian coordinate **Y** into the display-register. Pressing j activates the polar-to-rectangular conversion.

The inverse transformation, rectangular-to-polar, produces the results summarized in the table below:

	Before	After
Display-register	Y	Θ
Addressable memory register 00	X	R

This is the reverse effect of the polar-to-rectangular transformation and is activated by the sequence **@j**.

To use these transformations you must store and recall quantities using the addressable memory register 00. The following examples show two possible key sequences.

Example: Convert to Cartesian coordinates: R = 13, Θ = 43°

ANGLE:Degrees

ENTER	PRESS	DISPLAY	REMARKS
13	k 00	13.	
43	j	8.865978681	Value of Y
	m 00	9.507598121	Value of X

Example: Convert to polar coordinates (radians): X = 12, Y = 5

ANGLE:Radians

ENTER	PRESS	DISPLAY	REMARKS
12	k 00	12.	
5	@ j	.3947911197	Θ
	m 00	13.	Value of R

ALGEBRAIC NOTATION: MORE ABOUT PENDING OPERATIONS

THE ALGEBRAIC HIERARCHY

You have already read about how parentheses can be used on ASTROCAL 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 also just like the usual algebraic practice, for in normal algebraic usage the sequence of operations appropriate to 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. Although the sequence chosen in such cases is generally 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: ASTROCAL 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; ASTROCAL 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 that should be made when the completely defining compliment of parentheses is not present. As you have probably anticipated, ASTROCAL is absolutely faithful to these additional rules as well. Suppose one has the expression 7 x (6 + 2) x 9, the parentheses leave no doubt as to the meaning: 7 x 8 x 9 = 504. What would be the meaning of the expression if the parentheses were removed? 7 x 6 + 2 x 9 = ?

There would appear to be several possibilities of interpretation. In addition to the interpretation already given (leading to the answer 504) you could construct the following:

1. (7 x 6) + (2 x 9) = 42 + 18 = 60

2. ((7 x 6) + 2) x 9 = (42 + 2) x 9 = 44 x 9 = 396

3. 7 x (6 + (2 x 9)) = 7 x (6 + 18) = 7 x 24 = 168

There appears 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 that 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, division or multiplication should be performed prior to addition or subtraction. So the answer to the earlier question is 7 x 6 + 2 x 9 = 60. If you key that sequence with ASTROCAL, you will get 60 as the answer.

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

$$8 \div 12 \times 9 + 4 \times \sin 30^{\circ\tan 30^{\circ}} = ?$$

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 forgoing expression should be interpreted to mean

$$(8 \div 12 \times 9) + (4 \times .5^{.5773502692})$$

The value of this expression is 8.680774244. Now see if you get that answer on ASTROCAL. You will if you use the natural keystrokes:

8 / 12 * 9 + 4 * 30 s → 30 t =

(Did you remember to set the angular mode to degrees?)

To summarize, the interpretation of an expression is affected by the presence of parentheses. In places where the absence of the parentheses would leave doubt as to the proper operation sequence, that ambiguity is resolved by the rules of algebraic hierarchy. ASTROCAL 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 determine completely 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 addressable memory to store and recall data. To know what quantity is being stored in response to a store command, **k**, "STO MEM" from a program (or from the keyboard), 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 instruction by instruction, trying to anticipate what will be displayed with each keystroke. This will facilitate your using the program mode much more than using the calculate mode, because in executing a program you do not see what is in the display-register as ASTROCAL races through the program code. So practice in the calculate mode and you will greatly improve your effectiveness in the program mode.

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

Example: **6 * 4 m94 =**

This sequence produces the result 6 **x** M94, where M94 is the number stored in addressable memory register 94. The **4** was obliterated by the **m94**.

Example: **m82 m43 → m32 m74**

This example is just to make the point again. The result of this sequence is M43 to the M74 power.

Example: **8 + 16.2 n 74 =**

In this example the answer produced is 82. The 74 entry obliterated the natural logarithm of 16.2 present in the display-register, but the pending addition and 8 stored in the internal processing registers were not affected. This example may appear silly, for there is no purpose served by the natural logarithm of 16.2 calculation interjected where it was. However, the example shows that you could harmlessly perform the natural logarithm of 16.2 calculation in the middle of the unrelated problem 8 + 74 = 82. The fact that nothing was then done with the natural logarithm of 16.2 result (such as storing it into addressable memory or using it as the basis for a decision as discussed later) is not the main point. You could have done something with the natural logarithm of 16.2 before obliterating it. Hopefully, this simple example gives you further insight into the workings of the internal processing registers and the display-register.

ALGEBRAIC NOTATION: MORE ABOUT PENDING OPERATIONS

Finally, let me show you a little more practical example.

Example: **25 + @ q =**

This may not appear to illustrate the same point; it really does, as well as a second point. The answer obtained is 25 + 5 = 30. What occurred is this: after the **25 +**, addition was pending, with 25 stored in the processing registers. Furthermore, 25 was in the display-register. Keying in 5 at this point would have had the same effect as what occurred as a result of @ q: Namely @ q replaced the 25 in the display-register with its square root just as though that number had been keyed in. With the addition still pending = completed the calculation. This example and the ones preceding fall into the "peculiar-sequence" category. However, you will note that in the last example 25 + the square root of 25 was computed without entering 25 more than once, so there was some economy realized. You must be careful though, because if you press **25 + =**, you do not get 25 + 25 = 50, 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.

ADDRESSABLE MEMORY REGISTERS

ASTROCAL has 100 addressable memory registers for holding data. The addressable memory registers are designated in text by a notation as AM23. The contents of addressable memory registers are designated in text by a notation as M43.

It is usually arbitrary what addressable memory register you use for storing various data as long as you keep them straight. You should exercise discretion in using register AM00 because ASTROCAL uses this register during polar/rectangular conversions and Dec JPNZ execution (this will be discussed in detail later). I therefore advise you to form the habit of not using AM00 for routine data storage.

STORING DATA INTO ADDRESSABLE MEMORY

Consider evaluating an expression such as $25X^4-12X^3+3X+7$ for a value of X to be keyed in: X = 45.714, for example. Obviously, you don't want to key in this value more than once. To avoid this, simply store the value into addressable memory the first time it is keyed in and recall it from addressable memory whenever it is needed in the course of evaluating expressions. The following example shows another effective use of addressable memory.

Example:

$$J = \frac{\sin[(3X^2+6X-12)/(X^3+23)]+\cos[(3X^2+6X-12)/(X^3+23)]}{\sin[(3X^2+6X-12)/(X^3+23)]-\cos[(3X^2+6X-12)/(X^3+23)]}$$

In addition to storing X for the evaluation of the expression $(3X^2+6X-12)/(X^3+23)$, the value of this expression should be stored because it is the argument of both the sine and cosine functions. I will return to these examples soon. For now, they are presented to show typical situations for use of addressable memory.

To store the value contained in the display-register into a given addressable memory register, you just key **k** followed by the two-digit number of the addressable memory register.

Example: Store 45.714 into addressable memory register 37.

ENTER	PRESS	DISPLAY
45.714	k37	45.714

Example: Calculate $\pi^2/6$ and store the results in AM75.

ENTER	PRESS	DISPLAY
	p q /	9.869604401
6	= k75	1.644934067

RECALLING DATA FROM ADDRESSABLE MEMORY

At any point in a calculation, a value stored in addressable memory can be introduced into the display-register as if it had been keyed in at that point. The keystrokes required to accomplish this are press **m** followed by the two-digit number of the addressable memory register that the value is stored in.

Example: D **x** H^T = ?

Where D is the value stored in AM23,
H is the value stored in AM34,
T is the value stored in AM28.

Press: **m23 * m34 → m28 =**

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

Example: Compute $25X^4-12X^3+3X+7$

Where X = 45.714

ENTER	PRESS	DISPLAY	REMARKS
45.714	k01	45.714	X into AM01
	→	45.714	
4	*	4367137.8	X^4
25	-	109178445.	$25X^4$
12	* m01	45.714	
	→	45.714	
3	+	108032054.2	$25X^4-12X^3$
3	* m01	45.714	
	+	108032201.3	$25X^4-12X^3+3X$
7	=	108032208.3	Answer

Example: Compute the following with X = π/6 radians.

$$J = \frac{\sin[(3X^2+6X-12)/(X^3+23)]+\cos[(3X^2+6X-12)/(X^3+23)]}{\sin[(3X^2+6X-12)/(X^3+23)]-\cos[(3X^2+6X-12)/(X^3+23)]}$$

ANGLE:Radians

ENTER	PRESS	DISPLAY	REMARKS
	p /	3.141592654	
6	= (k01	.5235987756	X into AM01
	q *	.2741556778	X^2
3	+	.8224670334	$3X^2$
6	* m01 -	3.964059687	$3X^2 + 6X$
12	) / (	-8.035940313	$3X^2 + 6X - 12$
	m01	.5235987756	
	y +	.1435475772	X^3
23	)	23.14354758	X^3+23
	=	-0.34722163	$Q=(3X^2+6X-12)/(X^3+23)$
	k02	-0.34722163	Q into AM02
	s	-.3402865623	sin Q
	k03	-.3402865623	sin Q into AM03
	+ m02	-0.34722163	
	c	.9403217829	cos Q
	k04	.9403217829	cos Q into AM04
	=	.6000352206	sin Q + cos Q
	/ (	.6000352206	
	m03	-.3402865623	
	- m04	.9403217829	
	)	-1.280608345	sin Q - cos Q
	=	-.4685548262	Answer

The above example would have appeared complicated if the arguments had to be entered each time they occurred. Addressable memory was used to hold **X** in AM01, **Q** in AM02, **sin Q** in AM03, and **cos Q** in AM04. The use of addressable memory has saved keystrokes (and time —— transcendental calculations for ASTROCAL[1] range from four to greater than twenty seconds) in the evaluation process.

CLEARING THE ADDRESSABLE MEMORY REGISTERS

When ASTROCAL is initialized, all addressable memory registers contain the number zero. As you proceed to use addressable memory, some of these registers acquire non-zero contents. It is frequently desirable to zero all of the addressable memory registers without re-initializing ASTROCAL (re-initializing would also abort internal processing and blank program memory). This addressable memory register clearing could be accomplished by storing zero in each one of the one hundred registers; however, it is far more convenient to use the Clear Memory instruction. To clear all one hundred addressable memory registers at any time without affecting the internal processing registers, display, or program memory, simply press **SHIFT·CLEAR.**

DIRECT ADDRESSABLE MEMORY ARITHMETIC

You can store a number 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 addressable memory without affecting the calculation in progress. You can add the display, Q, to the contents of any addressable memory register; you can subtract Q from any addressable memory register contents; you can multiply by Q; and you can divide the contents of any addressable memory register by Q. Of course, until you recall the resultant quantity, you cannot see that any operation has taken place.

Again denoting the display-register contents by Q, and now the addressable memory register number by nn, the direct addressable memory register arithmetic is performed as follows:

- To ADD Q to the contents of Amnn, press **u nn**.
- To SUBTRACT Q from the contents of Amnn, press **@ u nn**.
- To MULTIPLY the contents of AMnn by Q, press **v nn**.
- To DIVIDE the contents of AMnn by Q, press **@ v nn**.

Example: Calculate 12 **x** 6, 17 **x** 3.2, and 11.8 **x** 4; accumulate the sum of these three products into AM56.

ENTER	PRESS	DISPLAY	REMARKS
	CLEAR	0	
	SHIFT·CLEAR	0.	All addressable memory is set to zero.
12	*	12.	
6	=	72.	12 **x** 6
	u56	72.	Sum 72 into AM56
17	*	17.	
3.2	=	54.4	17 **x** 3.2
	u56	54.4	Sum 54.4 into AM56
11.8	*	11.8	
4	=	47.2	11.8 **x** 4
	u56	47.2	Sum 47.2 into AM56
	m56	173.6	Final sum

Example: Compute $\log_e 2$ and store into AM87, then compute $\log_e 3$, store in AM88 and multiply the contents of AM87 by this quantity, leaving the results in AM87.

ENTER	PRESS	DISPLAY
2	n	.6931471806
	k87	
3	n	1.098612289
	k88	1.098612289
	v87	1.098612289
	m87	.7615000104

ADDRESSABLE MEMORY REGISTERS

When direct addressable memory register arithmetic results in overflow or too-small, "ERROR" will flash, indicating the error. The error condition remains until the contents of that addressable register is changed.

Example:

ENTER	PRESS	DISPLAY		REMARKS
1	e	1	0000	
9999	k01	1.	9999	
	v01	1.	9999	"ERROR" is flashing.
	SPACE	1.	9999	"ERROR" stops flashing.
3	y	2.7	0001	Cube calculation performed properly.
	m01	9.999999999	9999	"ERROR" flashing again, indicating the error condition is still present.

ADDRESSABLE MEMORY/DISPLAY EXCHANGE

An additional memory instruction is available that combines the effects of a store and a recall instruction in a single step. This is the exchange instruction.

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

Example:

ENTER	PRESS	DISPLAY	REMARKS
23	k04	23.	23 in AM04
51	*	51.	
6	-	306.	51 **x** 6
	g04	23.	306 in AM04
	=	283.	
	m04	306.	

The effect of the exchange function in this example was to store 306 into AM04 while recalling the earlier content of AM04, namely 23. The pending substraction was unaffected by the exchange.

This instruction can 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 efficient from an instruction (keystroke) point of view, and also from an addressable memory usage point of view. One addressable memory register serves for two data, storing the second data value concurrently while recalling the first.

SUPPLYING MISSING OPERANDS WITH MEMORY FUNCTIONS

There is one effect of all memory functions that has not been mentioned. After these functions have taken place, it is just as though the quantity in the display-register has been keyed in. You may recall from the earlier discussion that single-variable functions behave the same way. Allowing us to key in **25 + @ q =** and obtain 30, whereas **25 + =** gives us an error indication (lacking a second operand) rather than 50 as one might guess. The square-root function replaced 25 in the display-register with 5, exactly as though 5 had been keyed in, and thereby providing the second operand to go with **25 +**.

The memory functions behave in the same way. As a matter of fact, it is not even necessary to perform a store, recall, sum, product, or exchange to accomplish this effect. The single keys **k**, **m**, **u**, **v**, and **g** serve the same purpose even when not followed by a two-digit number. Although this falls into the category of a peculiar key sequence, a very useful construction is the following:

[...Sub-Expression...] (k OP ... etc,

where OP denotes any operation. If the sub-expression in brackets is needed again as the first operand in the parenthetical expression, then, rather than storing it and immediately recalling it from addressable memory, the **k** alone can accomplish the same purpose. To demonstrate this let's return to **25 + =**.

ENTER	PRESS	DISPLAY	REMARKS
25	+	25.	Addition pending
	k	25.	Does not store, and has the effect of reentering contents of display.
	=	50.	25 + 25

Example: Evaluate $8.23562584^{8.23562584}$

ENTER	PRESS	DISPLAY
8.23562584	→	8.23562584
	k	8.23562584
	=	34780405.65

Example: $\dfrac{(25.3 + 54.2)}{(4.7 - 3.9)} \times \log_{10} \dfrac{(25.3 + 54.2)}{(4.9 - 3.9)}$

ENTER	PRESS	DISPLAY	REMARKS
25.3	+	25.3	
54.2	= / (	79.5	25.3 + 54.2
4.7	-	4.7	
3.9	= *	99.375	(25.3+54.2)/(4.7-3.9)
	l	1.997277142	
	=	198.479416	

EXECUTING PROGRAMS STORED ON DISK

The most effective use of ASTROCAL is realized when you EXECUTE a stored program. The instructions for using each program are unique for that particular program. For programs that you write, I strongly recommend that you include the detailed operating instructions in remarks. This should be considered 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 ASTROCAL, both the mechanics and questions of technique and style. There are two basic steps necessary to execute programs stored on a disk: loading the program from disk and beginning execution.

FILE NAMES

Any program that has been saved on disk is a file. You must assign each file a unique name to the logical disk drive or diskette it occupies. When you save a file to a logical disk drive or diskette, the file name and where the file is located are saved in that disk's directory. If you were to use a file name that you had already used on a particular logical drive or diskette, the more recent file would be written over the existing file. File names, however, are unique only to a logical disk drive or an individual diskette.

A proper file name may be no longer than eight characters in length, the first of which must be a letter. The rest of the file name may include numbers. Thus, RATETWO and RATE2 are both proper and unique file names. File names cannot include punctuation, spaces, or special characters. All letter characters in a file name are interpreted as upper case.

File Name Extensions

In addition to the file name, ASTROCAL also recognizes extensions. ASTROCAL adds the default extension /CAL to any file name entered. If a drive number is part of the filespec, then ASTROCAL will insert /CAL between the file name and the drive number. Thus RATE:1 and RATE/CAL:1 are equivalent.

You can override the default extension by creating your own, such as /AC1 or /QWE. ASTROCAL adds the /CAL extension if you do not include one in the "Filespec:" prompt. However, if you do not want an extension, then only include the slash in the filespec. e.g., RATE/ to load the file RATE.

EXECUTING PROGRAMS STORED ON DISK

FILE SEARCH

ASTROCAL searches the selected logical disk drive or diskette for all files with a /CAL extension if only **:d** (d is the logical drive number) is entered for the "Filespec:". ASTROCAL displays the files one at a time on the "Filespec:" prompt line. When you see the file name you want to load, merge, or save press <ENTER>. The → key displays the next /CAL file on the logical disk drive or diskette. If you inadvertently pass the desired file name, press the ← key to scan in the other direction. The files are not displayed sorted, and the order that they appear may change as you add additional files to the logical disk drive or diskette.

LOADING A PROGRAM FROM DISK

The following are the steps required to load a program from disk into program memory.

1. Press **CTRL·l** to receive the "Filespec:" prompt.

2. Enter the filespec of the stored program.

NOTE: You can load, merge, or save a program in the program mode or the calculate mode.

If there are no error messages, the program is now loaded into program memory and remains there until you exit ASTROCAL, load or merge another program, or key in new instructions.

When you load a program from disk, all 10,000 locations of program memory are blanked before the new program is loaded. If you want to retain all or part of the current program in program memory, please refer to the merge command that follows.

MERGING A LOAD MODULE FROM DISK

The purpose of the merge command is to enable you to create code segments to perform some specific function that is needed in several programs. Instead of continually keying the same instructions for each program that needs a particular set of instructions performed, key the instructions once and save this code segment on disk for future needs.

The following are the steps required to merge a program from disk to another program in program memory.

1. Press **CTRL·p** to transfer to the program mode.

2. Press **CTRL·/**, then the four-digit location you want the merged code segment to start.

3. Press **CTRL·m** to receive the "Filespec:" prompt.

4. Enter the filespec of the stored program.

If there are no error messages, then the program is now merged with the current program in program memory and remains there until you exit ASTROCAL, load or merge another program, or key in new instructions.

5. Press **CTRL·p** again to transfer back to the calculate mode, or repeat steps 2 through 4 to merge another program segment.

Of considerable significance is the fact that loading or merging a program into program memory does not affect the addressable memory registers, program flags, or display format. In addition, a program can be loaded or merged without affecting internal processing registers or the displayed number. This means that if you find you cannot fit an entire program into 10,000 instructions, then partition the total program into segments or load modules that do fit into program memory separately. The program is then executed by sequentially loading and executing the load modules.

Obviously, the merge command could overlay part of or the entire current program in program memory. Because the merge command does not blank program memory, any prior instructions in program memory that are not overlaid are still present. This may be the intended purpose, or perhaps you do not care that there may be extraneous instructions in program memory. However, whenever a save to disk is executed, ASTROCAL will save every instruction in program memory, adjusting the length of the save as required to include the last instruction in program memory.

EXECUTING PROGRAMS STORED ON DISK

SAVING A PROGRAM TO DISK

You can save any program in program memory to disk. But, if you exit ASTROCAL, the data in program memory may be lost.

The procedure for saving a program to disk is similar to loading a program from disk.

1. Press **CTRL·s** to receive the "Filespec:" prompt.

2. Enter the name you want this version of the program saved as.

Saving a program does not affect the addressable memory registers, program flags, or the display format; however, the program counter is set to 0000.

NOTE: You can load, merge, or save a program in the program mode or the calculate mode.

EXECUTING A PROGRAM

Execution of a well designed program is started by pressing one of the 26 user defined label keys: upper case **A** through **Z**. For the Model 4, upper case **A** through **F** can be simulated with **F1**, **F2**, **F3**, **RIGHT·SHIFT·F1**, **RIGHT·SHIFT·F2**, and **RIGHT·SHIFT·F3**, respectively. These 26 keys are special in that pressing one causes ASTROCAL to position the program counter to the point in program memory so labeled and to transfer into the execute mode as soon as the program counter is positioned.

Another method to start program execution is to position the program counter to the desired starting point while ASTROCAL is in the calculate mode, perhaps enter data with the keyboard, and finally enter the execute mode by pressing **SHIFT·ENTER**. Positioning the program counter can be accomplished with the go-to instruction that will be discussed in the **TRANSFER INSTRUCTIONS** section. More likely, if the program counter needs positioning it is to the start of the program memory; and you can accomplish this by the reset instruction. The reset instruction is entered from the keyboard by pressing comma. The reset instruction may not be appropriate as it also clears program flags and subroutine return-pointer registers.

GENERAL PROGRAMMING INSTRUCTIONS

From what you have learned in the preceding sections, you are already basically prepared to write useful programs for ASTROCAL. This comes from the fact that each keystroke in the calculate mode can be stored in a program location in the program mode (where it is called an instruction). And when this instruction is executed in the execute mode it has the 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 ASTROCAL. The first group of ingredients is essential: a handful of controls enabling you to specify when the program should halt for data entry or for you to look at the answer, the label statements that enable you to conveniently start execution at different points in the program and that sort of thing.

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

The third set of programming ingredients completes your full capabilities as an ASTROCAL user: They relate to a very orderly method of problem solving that ASTROCAL provides. This 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 (that is a label) and go on with the main problem definition. When you are ready to define those quantities that have only been named until now, you do so immediately following the appropriate label.

The actual ASTROCAL evaluation of the problem solution in the execute mode invokes all those detailed definitions that have been delayed until you completed the statement of the whole problem at a higher level. It invokes those definitions simply upon recognizing the names (or labels) assigned, and inserts the labeled definitions and evaluations of the deferred quantities. This method of top down problem solution is made possible by the subroutine capability. ASTROCAL provides for automatic performance of program execution at 73 levels. This means that not only can you 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, and this process of deferring quantities can be deferred 71 additional levels!

The 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 10,000 locations in program memory, numbered from 0000 to 9999. Each location can hold one keystroke.

When ASTROCAL is in the execute mode, the sequencing of the program steps is accomplished by means of the program counter (PC). This may be thought of as a marker that moves through the program memory indicating the next instruction to be executed. The normal operating sequence in the execute mode is straightforward.

```
  instruction fetched; PC advanced one location; instruction executed;
  instruction fetched; PC advanced one location; instruction executed;
  instruction fetched; PC advanced one location; instruction executed;
  instruction fetched; PC advanced one location; instruction executed;
    ...
    ...
    ...
  etc.
```

In this normal sequence, ASTROCAL steps through program memory, executing the instructions in exactly the order they exist. If ASTROCAL attempts to go past location 9999 it will flash "ERROR" and return to the calculate mode.

Certain types of instructions alter this simple top-to-bottom execution of the program. These are the transfer instructions. Detailed discussions of these are 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 will unconditionally reposition the program counter to the location specified in the transfer instruction. The destination, therefore, becomes the next instruction fetched. The program counter executes this instruction and is then advanced a step at a time until another transfer instruction is executed.

The conditional transfer instruction is similar to an unconditional transfer except for one thing: a conditional transfer instruction first performs some test (for example, is the display-register positive or is a flag set?); the transfer occurs only if the test is affirmative. Otherwise, the program counter fetches the instruction at the program location.

The time history of a typical program using transfer instructions might be as follows.

```
     Fetch instruction;              Fetch instruction;
     advance counter;                advance counter;
     execute instruction.            execute instruction.
     Fetch instruction;              Fetch instruction;
     advance counter;                advance counter;
     execute instruction.            execute instruction.
     Fetch instruction;              Fetch instruction;
     advance counter;                advance counter;
     execute instruction.       [3]  Perform test (affirmative result);
     Fetch instruction;              Reposition counter.
     advance counter;           [4]  Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
[1]  Reposition counter.             advance counter;
[2]  Fetch instruction;         [5]  Perform test (negative result);
     advance counter;                Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
     execute instruction.            advance counter;
     Fetch instruction;              execute instruction.
     advance counter;                Fetch instruction;
     execute instruction.            etc.
```

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

MECHANICS OF PROGRAMMING

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

1. Gather equations that pertain to the problem.

2. Define numerical approach (algorithms) for solving equations.

3. Determine how you would like the program to be used (input data, quantities computed, 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 addressable memory register assignments. This task continues through the programming process. (Do not store a quantity in addressable memory without making a written note that the register in question contains that quantity. And, I recommend adding remarks at the end of the program for all addressable memory usage.)

6. Actual coding: Write down each instruction with a method for easily identifying the insertion of additional instructions.

7. Make corrections to code, addressable memory assignments, or even procedure for program use if necessary.

8. Transfer ASTROCAL into the program mode.

9. Key in program (including remarks in appropriate places).

10. Save the program on disk, if desired.

11. Transfer ASTROCAL into the calculate mode.

12. Check out program on test problems.

13. Make necessary corrections.

14. Document complete 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 use it conveniently at any time. Another way of stating this advice: The programming language of ASTROCAL 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 requirements so that the resulting program meets your problem solving needs.

USING LABELS

When you execute a program, it is necessary to properly position the program counter and switch to the execute mode. Positioning the program counter can be accomplished with the help of labels. Automatic switching to the execute mode can also be accomplished by using the special labels **A** through **Z**. These special upper case letter keys are also known as the ***user-defined keys***, because the effect of pressing one is to execute the program defined and labeled. I will discuss these special labels first and then discuss the other types of labels.

Using the concept of 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 ASTROCAL into the execute mode.

This means if you wish to begin execution at 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 **SHIFT·@**, followed by the label.

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

> Store the quantity in the display register into AM01 and multiply this quantity by the sine of the value in AM23.

To do this, code the following sequence into program memory:

```
LABEL
W
STO MEM
  0
  1
  x
RCL MEM
  2
  3
Sine
  =
```

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

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 **k01**, 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 other instruction steps.

Any key except the digits **0** through **9** may be used as a label.

I 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 **@** 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 **LABEL Inverse** sequence in the program. I will discuss how to use such labels in the next section. However, you should know that the placement of those labels in the code sequence is identical to what has already been described: To establish a location labeled **Inverse** in the program memory, just key in the instructions **LABEL Inverse** (<SHIFT>@ @) in the location immediately preceding the one to acquire that name. If you have more than one location with the same label, only the first one (the one with the lower program location) will be found.

USING EXECUTE 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 "HALT", the **BREAK** key. 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 **SHIFT·ENTER**, Execute, the execute mode is restored and execution is resumed from the point indicated by the program counter.

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

Pressing the **BREAK** from the keyboard when ASTROCAL is in the execute mode stops program execution and returns control to the keyboard. The program counter is remains wherever it happened to be at the time of program interruption (you cannot interrupt a program between a memory operation). Program execution is resumed at that location when **SHIFT·ENTER** 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 SHIFT·ENTER and see the answer.

KEY SEQUENCE	INSTRUCTION	REMARKS
SHIFT·@	LABEL	Labels start of program
A	A	to calculate volume
q	Square	r was entered before A is pressed
:	x	
p	pi	
:	x	πr^2 in display-register
BREAK	HALT	enable h to be entered
ENTER	=	$V = \pi r^2 h$
BREAK	HALT	Answer

ENTERING A PROGRAM

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

1. With ASTROCAL in the calculate mode press **CTRL·p**, placing ASTROCAL in the program mode.

2. Then press **SHIFT·↑** to position the program counter to 0000, the top of program memory.

3. Press **CTRL·t** to erase any program currently in program memory. You will see four rows of seven digits (one group of four and one group of three) below the display. The four digits, under "Location", to the right of a ">" is the location of the program counter.

4. Key in the program completely. As you key in an instruction, the instruction appears under "Instruction", and the program counter advances one location. The program format enables you to see the previous three instructions keyed in. If you press a wrong key, use ↑ to reposition the program counter to the previous location and over key the incorrect instruction.

5. Return to the calculate mode by pressing **CTRL·p**.

6. Execute a test problem and correct (edit) the program as required.

7. The program can now be saved on a disk.

EDITING PROGRAMS

When keying in a program, you have the following capabilities:

1. Display the current location and instruction with up to three preceding and three following instructions;
2. Replace the current instruction;
3. Delete current instruction and move all following instructions up;
4. Insert a remark prior to the current instruction (moves all following instructions down 40 locations);
5. Insert a message (up to 79 characters) prior to the current instruction (moves all following instructions down by the length of the message);
6. Insert a blank instruction at the current location and move all following instructions down;
7. Reposition the program counter up or down by one instruction;
8. Reposition the program counter to any location;
9. Delete all instructions from the current location;
10. Print all instructions from the current location;
11. Print all instructions from location 0000 to the last instruction;
12. Find the next blank instruction;
13. Find the last instruction;
14. Find the next occurrence of an instruction.

These capabilities enable you to modify a program easily without reentering instructions that require no change.

Displaying the Program

In the program mode, there is additional information displayed below the display: "Location Code Instruction". This information, program code, is designed to show you where the program counter is located and the current instruction. The first four digits are the location, the second three digits are the key code of the instruction that is displayed under "Instruction". The current location, code, and instruction are preceded with a **>** symbol.

Replacing an instruction

In accordance with the foregoing discussion, the current instruction is indicated with a **>** symbol (the location of the program counter). To substitute another instruction into that location (except for remarks), simply key in the new instruction and it will replace the old one in that location. Remarks can be deleted (see below) or edited with **CTRL·e**.

Deleting an instruction

Pressing **CTRL·d** deletes the current instruction and moves all instructions with a higher location up to close the space. As a consequence of the closing-up operation in instruction deletion, the last location in program memory (9999) is blanked.

Inserting Remarks

Pressing **CTRL·r** enables you to insert program remarks. All remarks in ASTROCAL use 40 locations with a text size of 38 characters (if you key in 20 characters, ASTROCAL pads the right with 18 spaces, and remarks cannot be entered above location 9959). Inserting a remark pushes down the current (and all instructions with a higher location) 40 locations. **CTRL·i** (see below) is not required.

Inserting Messages

Pressing **CTRL·c** enables you to insert messages conveniently. Messages may be entered one character at a time by pressing LEFT-SHIFT·F3. Inserting a message pushes down the current instruction (and all instructions with a higher location) the length of the message. **CTRL·i** (see below) is not required.

Inserting an instruction

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

Step 1. Press **CTRL·i** to push down the current instruction (and all instructions with a higher location) one location, leaving a blank instruction in its place.

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

As you insert instructions, the instruction in the last location (9999) is pushed out of program memory and lost.

Single-Step and Back-step

You will frequently wish to examine portions of the program as stored in program memory. There are several editing commands that enable you to accomplish this. To single-step, press ↓ or **CTRL·z** in the program mode and the location of the program counter is incremented by one. To back-step, press ↑ or **CTRL·q** and the location of the program counter is decremented by one. Both the single-step and back-step commands do not affect the contents of the program in program memory in any way.

Other program counter relocation commands

Pressing **SHIFT·↑**, **SHIFT·←**, or **CTRL·,** repositions the program counter to location 0000. Pressing **SHIFT·↓**, **SHIFT·→**, or **CTRL·-** repositions the program counter to location 9999. Pressing **CTRL·/** followed by four digits repositions the program counter to the location specified by the four digits. This command enables you to directly reposition the program counter to any location in program memory. In the calculate mode, you can press **?** followed by a label or four digits to position the program counter to a location, before you transfer to the program mode. In the program mode you cannot use labels for positioning the program counter.

Erasing several instructions

When you have a program in program memory, and you wish to input another program, you can simply key in the new program by over keying the instructions there. However, you may wish to completely erase the current program and start with all blank program memory. Without re-initializing ASTROCAL to accomplish this, press **CTRL·t** to blank all instruction from the current location to the end of program memory.

PROGRAM KEYS, CODES, and INSTRUCTIONS

On the following pages are the program keys, the ASTROCAL key code value, the instruction that is displayed and the function of the instruction.

KEY	CODE	INSTRUCTION	FUNCTION
SHIFT·0 or SHIFT·SPACE	000		No operation (Blank instruction)
BREAK	002	HALT	Stop execution (return to calculate mode)
CTRL·h	008	$ hold $	Suspend execution
CTRL·x	024	break pt	Stop execution enter single-step mode
SHIFT·ENTER	027	Execute	Execute from current instruction
!	033	!	Factorial
"	034	Set flag	Set user flag (I)
#	035	If flag	Test user flag (I)
$	036	If error	Test error condition (I)
%	037	If pos	Test sign (I)
&	038	If zero	Test value (I)
'	039	Dec JPNZ	Conditional loop (I)
(	040	(	Open parenthesis
)	041	)	Close parenthesis
*	042	x	Multiply
+	043	+	Add
,	044	Reset	Resets flags, return pointers, & program counter
-	045	-	Subtract
.	046	.	Decimal
/	047	/	Divide
0	048	0	Digit zero
1	049	1	Digit one
2	050	2	Digit two
3	051	3	Digit three
4	052	4	Digit four
5	053	5	Digit five
6	054	6	Digit six
7	055	7	Digit seven
8	056	8	Digit eight
9	057	9	Digit nine
:	058	x	Multiply (functions as ***** with different code)
;	059	+	Add (functions as **+** with different code)
<	060	Return	Return from subroutine
ENTER or =	061	=	Equal
>	062	Gosub	Perform subroutine
?	063	Goto	Unconditional transfer
@	064	Inverse	Inverses appropriate functions [functions suffixed with (I)]
SHIFT·A	065	A	User-defined key, also F1 on MAX-80 and Model 4
SHIFT·B	066	B	User-defined key, also F2 on MAX-80 and Model 4
SHIFT·C	067	C	User-defined key, also F3 on MAX-80 and Model 4
SHIFT·D	068	D	User-defined key, also F4 (RIGHT·SHIFT·F1) Mod 4
SHIFT·E	069	E	User-defined key, also F5 (RIGHT·SHIFT·F2) Mod 4
SHIFT·F	070	F	User-defined key, also F6 (RIGHT·SHIFT·F3) Mod 4
SHIFT·G	071	G	User-defined key
SHIFT·H	072	H	User-defined key
SHIFT·I	073	I	User-defined key
SHIFT·J	074	J	User-defined key

KEY	CODE	INSTRUCTION	FUNCTION
SHIFT·K	075	K	User-defined key
SHIFT·L	076	L	User-defined key
SHIFT·M	077	M	User-defined key
SHIFT·N	078	N	User-defined key
SHIFT·O	079	O	User-defined key
SHIFT·P	080	P	User-defined key
SHIFT·Q	081	Q	User-defined key
SHIFT·R	082	R	User-defined key
SHIFT·S	083	S	User-defined key
SHIFT·T	084	T	User-defined key
SHIFT·U	085	U	User-defined key
SHIFT·V	086	V	User-defined key
SHIFT·W	087	W	User-defined key
SHIFT·X	088	X	User-defined key
SHIFT·Y	089	Y	User-defined key
SHIFT·Z	090	Z	User-defined key
→	091	Exponent	Raise to
←	092	Root	Root by
SHIFT·CLEAR	093	CLR MEM	Clear all memory
CLEAR	094	CLEAR	Clear calculator
SPACE	095	CLR ERR	Clear error condition/digit
SHIFT·@	096	LABEL	Label preface
a	097	paper	Line feed paper
b	098	D.MS/D.d	Degrees.MinutesSeconds to Degrees.decimal (I)
c	099	Cosine	Cosine of display (I)
d	100	Deg/Rad	Degrees to radians (I)
e	101	EE	Scientific notation exponent (I)
f	102	Fix Dec.	Fix decimal digits (I)
g	103	EXC MEM	Exchange display with memory
h	104	Hyperbol	Hyperbolic function
i	105	Indirect	Indirect memory addressing
j	106	Pol/Rect	Polar to rectangular coordinates (I)
k	107	STO MEM	Store display into memory
l	108	LOG 10	Common logarithm of display (I)
m	109	RCL MEM	Recall memory
n	110	LOG e	Natural logarithm of display (I)
o	111	+/-	Change sign
p	112	pi	π (3.14159265358979323846264338 3···)
q	113	Square	Square display (I)
r	114	1/x	Reciprocal display
s	115	Sine	Sine of display (I)
t	116	Tangent	Tangent of display (I)
u	117	SUM MEM	Add display to memory (I)
v	118	PROD MEM	Multiply memory by display (I)
w	119	print	Print display
x	120	trace	Execute program in trace mode
y	121	Cube	Cube display (I)
z	122	Sw mode	Angular mode —— degree or radians

Message Characters

KEY	CODE	INSTRUCTION	FUNCTION
ALTCTRL·↑	145	Carr Rtn	Positions cursor to starting position
			Codes 129 through 149 have equivalent function.
ALTCTRL·↓	154	New Line	Positions cursor to starting position and erases line
			Codes 150 through 157 have equivalent function.
ALTCTRL·!	161	|!|	Display **!** at cursor position
ALTCTRL·"	162	|"|	Display **"** at cursor position
ALTCTRL·#	163	|#|	Display **#** at cursor position
ALTCTRL·$	164	|$|	Display **$** at cursor position
...			
ALTCTRL·A	193	|A|	Display **A** at cursor position
ALTCTRL·B	194	|B|	Display **B** at cursor position
ALTCTRL·C	195	|C|	Display **C** at cursor position
ALTCTRL·D	196	|D|	Display **D** at cursor position
...			
ALTCTRL·W	215	|W|	Display **W** at cursor position
ALTCTRL·X	216	|X|	Display **X** at cursor position
ALTCTRL·Y	217	|Y|	Display **Y** at cursor position
ALTCTRL·Z	218	|Z|	Display **Z** at cursor position
...			
ALTCTRL·a	225	|a|	Display **a** at cursor position
ALTCTRL·b	226	|b|	Display **b** at cursor position
ALTCTRL·c	227	|c|	Display **c** at cursor position
ALTCTRL·d	228	|d|	Display **d** at cursor position
...			
ALTCTRL·w	247	|w|	Display **w** at cursor position
ALTCTRL·x	248	|x|	Display **x** at cursor position
ALTCTRL·y	249	|y|	Display **y** at cursor position
ALTCTRL·z	250	|z|	Display **z** at cursor position

ALTCTRL is the CLEAR key ***after*** pressing LEFT-SHIFT·F3. To produce a displayed space, use ALTCTRL·LEFT-SHIFT·SPACE (ALTCTRL·SPACE produces _). CAPS mode is ignored using ALTCTRL.

To reset the CLEAR key to the standard ASTROCAL value, press LEFT-SHIFT·F1.

Partial listing of DECTOHEX/CAL's first page (set for two-column listing).

```
0000   176      |0|                    0138    096    LABEL
0001   060    Return                   0139    073    I
0002   177      |1|                    0140    049      1
0003   060    Return                   0141    054      6
0004   178      |2|                    0142    091    Exponent
0005   060    Return                   0143    109    RCL MEM
0006   179      |3|                    0144    048      0
0007   060    Return                   0145    055      7
0008   180      |4|                    0146    061      =
0009   060    Return                   0147    060    Return
0010   181      |5|                    0148    000
0011   060    Return                   Make positive and fix decimals to zero
0012   182      |6|                    0189    096    Label
```

PROGRAM EDITING COMMANDS

KEY	FUNCTION
CTRL·c	Insert message characters (up to 79)
CTRL·d	Delete current instruction moving following instructions up and blanking location 9999.
CTRL·e	Edit a remark (similar to editing a DOS command line).
CTRL·f	Preface for "FIND".
CTRL·g or CTRL·/	GOTO nnnn preface.
CTRL·i	Move all instructions from the current instruction down and insert a blank instruction into the new current instruction.
CTRL·j	Entire program listing in two columns.
CTRL·k	Entire program listing in three columns.
CTRL·l	Load program starting at 0000.
CTRL·m	Load program starting at current instruction.
CTRL·n	Form fitted program listing in two columns.
CTRL·o	Form fitted program listing in three columns.
CTRL·p	Return to calculator mode.
CTRL·q or ↑	Back-step one instruction.
CTRL·r	Insert a remark.
CTRL·s	Save all instructions in program memory.
CTRL·t	Delete from current instruction to end of program memory.
CTRL·u	Seek next blank instruction.
CTRL·v	Single column listing from current instruction.
CTRL·w	Position to first free location in program memory.
CTRL·z or ↓	Step one instruction.
SHIFT·↑ or SHIFT·← or CTRL·,	Set program counter to 0000.
SHIFT·↓ or SHIFT·→ or CTRL·-	Set program counter to 9999.

Listing a Program

To list a program in a single column from the current instruction, press **CTRL·v**. The program listing prints all instructions from the current location to the last instruction in program memory including any intervening blank instructions. To list the entire program in a single column, position the program counter to location 0000 by pressing **SHIFT·↑**, **CTRL·,**, or **SHIFT·←** before you press **CTRL·v**.

Two and three column listings of all instructions in program memory can be obtained, regardless of the program counter location, by pressing **CTRL·j**, **CTRL·k**, **CTRL·n**, or **CTRL·o**.

CTRL·j, two-column, and **CTRL·k**, three-column, differ from **CTRL·n**, two-column, and **CTRL·o**, three-column, in the way they print the last page of instructions. Both **CTRL·j** and **CTRL·k** prints to the last instruction and balances the columns on the last page. Whereas **CTRL·n** and **CTRL·o** prints as many instructions on the last page that will fit. This usually results in the last page having many blank instructions, but keeps all listings in the same format (provided the DOS Library Command FORMS is the same for each listing).

EXAMPLE: You have 162 instruction in program memory (uses locations 0000 through 0161), and the forms is set for 60 text lines. **CTRL·j** prints 120 instructions on the first page and 42 instructions on the second page: 0120 to 0140 in the left column and 0141 to 0161 in the right column; printing 21 lines on the second page.

CTRL·n for 162 instructions prints 120 instructions on both the first page and the second page. The second page has instructions 0120 to 0179 in the left column and instructions 0180 to 0239 in the right column. Instructions 0162 through 0239 are blank. If the Library Command FORMS is set for the printer, both the **CTRL·n** and **CTRL·o** commands result in the paper at top-of-form after a program listing.

The program listings uses the full width of the FORMS setting. e.g., if the FORMS is set to a width of 80, then a two-column listings has the first column starting in position 1 and the second column starting in position 41.

Any of the listing commands can be stopped manually by pressing **BREAK**.

What prints for a remark depends on the width of the column. If your printer is set for 80 characters per line, the following is what is printed for single column, two-column, and three-column listings (ANNUITY/CAL - partial).

```
  Single: 0000    158    Remark: Program is initialized by pressing Z
          0040    158    Remark: To enter Present Value, press V

  Two:    Program is initialized by pressing Z    0489    052      4
          To enter Present Value, press V         0490    038    If zero

  Three:  0000    158    Remark:    0489    052    4          0549    091    Exponent
          0040    158    Remark:    0490    038  If zero      0550    040      (
```

NOTE: If there is insufficient width, remark text takes precedence over the location, code, and the instruction "Remark: " (two-column). The three-column listing has insufficient width to print the entire remark; therefore, only "Remark: " is printed.

Partial listing of DECTOHEX/CAL's last page (set for three-column listing with printer width set to 118 characters per line).

```
0604    236      |l|                Hexadecimal value into display charset 0842    071    G
0605    225      |a|                0724    096    LABEL                   0843    039    Dec JPNZ
0606    242      |r|                0725    122    Sw Mode                 0844    099    Cosine
0607    231      |g|                0726    109    RCL MEM                 0845    060    Return
0608    229      |e|                0727    048      0                     0846    000
0609    161      |!|                0728    052      4                     Here if zero is decimal value
0610    160      | |                0729    047      /                     0887    096    LABEL
0611    208      |P|                0730    072    H                       0888    103    EXC MEM
0612    236      |l|                0731    079    O                       0889    105    Indirect
0613    229      |e|                0732    107    STO MEM                 0890    063    Goto
0614    225      |a|                0733    048      0                     0891    048      0
0615    243      |s|                0734    049      1                     0892    051      3
0616    229      |e|                0735    042      x                     0893    000
0617    160      | |                0736    072    H                       0894    096    LABEL
0618    229      |e|                0737    061      =                     0895    081    Q
0619    238      |n|                0738    064    Inverse                 0896    160      | |
0620    244      |t|                0739    117    SUM MEM                 0897    232      |h|
0621    229      |e|                0740    048      0                     0898    229      |e|
0622    242      |r|                0741    052      4                     0899    248      |x|
0623    160      | |                0742    071    G                       0900    225      |a|
0624    225      |a|                0743    039    Dec JPNZ                0901    228      |d|
0625    160      | |                0744    122    Sw Mode                 0902    229      |e|
0626    238      |n|                0745    064    Inverse                 0903    227      |c|
0627    245      |u|                0746    102    Fix Dec.                0904    233      |i|
0628    237      |m|                0747    081    Q                       0905    237      |m|
0629    226      |b|                0748    174      |.|                   0906    225      |a|
0630    229      |e|                0749    109    RCL MEM                 0907    236      |l|
0631    242      |r|                0750    048      0                     0908    060    Return
0632    160      | |                0751    053      5                     0909    000
0633    236      |l|                0752    002    HALT                    0910    096    LABEL
0634    229      |e|                0753    000                            0911    085    U
0635    243      |s|                Decimal value into display charset     0912    160      | |
0636    243      |s|                0794    096    LABEL                   0913    228      |d|
0637    160      | |                0795    089    Y                       0914    229      |e|
0638    244      |t|                0796    107    STO MEM                 0915    227      |c|
0639    232      |h|                0797    048      0                     0916    233      |i|
0640    225      |a|                0798    051      3                     0917    237      |m|
0641    238      |n|                0799    038    If zero                 0918    225      |a|
0642    160      | |                0800    103    EXC MEM                 0919    236      |l|
0643    073    I                    0801    105    Indirect                0920    160      | |
0644    089    Y                    0802    108    LOG 10                  0921    060    Return
0645    174      |.|                0803    043      +                     0922    000
0646    109    RCL MEM              0804    049      1                     0923    000
0647    048      0                  0805    061      =                     0924    000
0648    053      5                  0806    107    STO MEM                 AM00: loop counter
0649    002    HALT                 0807    048      0                     AM01: indirect pointer
0650    000                         0808    048      0                     AM02: used
0651    096    LABEL                0809    096    LABEL                   AM03: used
0652    080    P                    0810    099    Cosine                  AM04: used
0653    109    RCL MEM              0811    109    RCL MEM                 AM05: decimal number entered
0654    048      0                  0812    048      0                     AM07: number of hexadecimal digits
0655    053      5                  0813    048      0                     1205    000
0656    107    STO MEM              0814    045      -                     Program is initialized by entering the
0657    048      0                  0815    049      1                     number of hexadecimal digits,
0658    052      4                  0816    061      =                     and pressing <F3>
0659    156    New Line             0817    064    Inverse                 1326    000
0660    212      |T|                0818    108    LOG 10                  1327    000
0661    232      |h|                0819    107    STO MEM                 1328    000
0662    229      |e|                0820    048      0                     1329    000
0663    160      | |                0821    050      2                     1330    000
0664    246      |v|                0822    109    RCL MEM                 1331    000
0665    225      |a|                0823    048      0                     1332    000
0666    236      |l|                0824    051      3                     1333    000
0667    245      |u|                0825    047      /                     1334    000
0668    229      |e|                0826    109    RCL MEM                 1335    000
0669    160      | |                0827    048      0                     1336    000
0670    239      |o|                0828    050      2                     1337    000
```

Find Commands

ASTROCAL has three find commands. One of the find commands enables you to relocate the program counter to the next blank instruction. This is usually the first free instruction in program memory. Finding the next occurrence of a blank instruction is accomplished by pressing **CTRL·u**. If the program does not have any blank instructions, **CTRL·u** will position the program counter to the first free instruction. Subsequent presses of **CTRL·u** simply single-steps in the blank area of program memory. In cases where there are one or more blank instructions in program memory —— intervening actual program code —— press **CTRL·w** to relocate the program counter to the first free instruction.

The last find command requires two steps to complete:

1. Press **CTRL·f** to set up the instruction find command.

2. Press the key for the instruction you wish to find.

The instruction find command requires the **CTRL·f**/**key** pair for each instruction sought. This means if you have more than one occurrence of the same instruction in program memory, you have to press **CTRL·f** followed by the sought after instruction for each find. You find remarks with **CTRL·f** followed by **CTRL·r**.

PROGRAM DEBUGGING

Single-Step Execution

The ↓ key in the calculate mode causes actual execution of the stored program one step at a time, and enables the display-register as well as the program code and instructions. **SHIFT·ENTER** exits the single-step instruction mode and resumes normal execution.

Trace

Pressing **x** in the command mode effects repeating single-stepping. You can insert an **x** in program memory to enable tracing whenever this instruction is encountered. Similarly you can disable tracing with **@x** in the execute mode. Similar to the execute mode, the trace mode can be stopped by pressing **BREAK**. Although the single-step and trace modes enable the displaying of the program code, you are not in the program mode as indicated with the absence of "PROG" present in the upper right on the screen. The program code will blank whenever any key is pressed except ↓. Another feature of an **@x** sequence encountered in the execute mode is to un-blank the display, displaying the contents of the display-register at that time. The display will not blank again, and only changes when a halt, hold, break point, trace, or another inverse trace instruction is encountered regardless of the contents of the display-register. This feature can be used to observe the results of an iterative calculation to see if the results are diverging or converging.

Hold

Another debugging aid is the hold instruction, **CTRL·h**, that is only recognized in the execute mode. The hold status is indicated with the presence of "$Hold" in the upper right on the screen. Pressing any key except **BREAK** while on hold causes the program to resume execution. The hold command can be used for halt without transferring to the calculate mode. This feature enables you to look at results without the capability to overwrite the display-register for arguments used for the instructions that follow. Of course you can press **BREAK** to override this. Also, the hold instruction can be used in places you will insert print instructions, **w**, after the program has been completely debugged. You may find the hold instruction useful as a substitute for print commands while the program is still in the development stage without using excessive paper.

Break Point

The last debugging aid is the break-point instruction, **CTRL·x**. The break-point instruction transfers the execute mode to the single-step mode and suspends execution. You can single-step the program at this point with the ↓ key, or resume execution with **SHIFT·ENTER**.

DEVELOPMENT OF PROGRAMMING STYLE

I would like to make a last point about the nature of the programming process, whether for ASTROCAL 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 ASTROCAL, you will develop your own unique style. That style may become one of incredible craftiness and ingenuity that makes frequent use of all 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 be easily discerned 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 conditional transfer instructions or indirect addressing, then just don't use those instructions (until your needs change).

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

PRACTICE PROBLEMS

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

C = (5/9)(F - 32)
K = C + 273.15

Possible solution:

Location	Code	Instruction
0000	158	Remark: <A> to convert Fahrenheit to Celsius
0040	158	Remark: <B> to convert Fahrenheit to Kelvin
0080	096	LABEL
0081	066	B
0082	059	+
0083	052	4
0084	057	9
0085	040	1
0086	046	.
0087	054	6
0088	055	7
0089	096	LABEL
0090	065	A
0091	045	-
0092	051	3
0093	050	2
0094	061	=
0095	058	x
0096	053	5
0097	047	/
0098	057	9
0099	061	=
0100	002	HALT

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 p, ϕ, and Θ 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 **SHIFT·ENTER** to display y, and **SHIFT·ENTER** again to display z.

$z = p \cos \phi$ $x = p \sin \phi \cos \Theta$ $y = p \sin \phi \sin \Theta$

Possible solution:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0026	109	RCL MEM
0001	065	A	0027	048	0
0002	107	STO MEM	0028	050	2
0003	048	0	0029	106	Pol/Rect
0004	049	1	0030	103	EXC MEM
0005	002	HALT	0031	048	0
0006	096	LABEL	0032	048	0
0007	066	B	0033	107	STO MEM
0008	107	STO MEM	0034	048	0
0009	048	0	0035	052	4
0010	050	2	0036	109	RCL MEM
0011	002	HALT	0037	048	0
0012	096	LABEL	0038	051	3
0013	067	C	0039	106	Pol/Rect
0014	107	STO MEM	0040	103	EXC MEM
0015	048	0	0041	048	0
0016	051	3	0042	048	0
0017	002	HALT	0043	002	HALT
0018	096	LABEL	0044	109	RCL MEM
0019	068	D	0045	048	0
0020	109	RCL MEM	0046	048	0
0021	048	0	0047	002	HALT
0022	049	1	0048	109	RCL MEM
0023	107	STO MEM	0049	048	0
0024	048	0	0050	052	4
0025	048	0	0051	002	HALT

Again, the proposed solution is a bit tricky; and, provided that your program gives the correct answer, mine 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 Pol/Rect sets up the polar to rectangular conversion to produce **p sin ϕ** in the display and **z** in AM00. Next, I exchange the contents of the display and AM00 and store **z** away in AM04 for later recall. Next, I recall **Θ** into the display-register from AM03 and perform another polar to rectangular conversion leaving **y** in the display-register and **x** in AM00. Another exchange places **x** where I want it for the first program result, the display-register. Following the HALT to enable us to see the value of **x**, **y** is recalled from AM00 and another program halt is provided. Finally, **z** is recalled from AM04 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, X_3, ..., X_n. The desired operation is that after initializing using key **D**, 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:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0014	096	LABEL
0001	068	D	0015	066	B
0002	093	CLR MEM	0016	040	(
0003	002	HALT	0017	109	RCL MEM
0004	096	LABEL	0018	048	0
0005	065	A	0019	049	1
0006	117	SUM MEM	0020	047	/
0007	048	0	0021	109	RCL MEM
0008	049	1	0022	048	0
0009	049	1	0023	050	2
0010	117	SUM MEM	0024	041	)
0011	048	0	0025	002	HALT
0012	050	2			
0013	002	HALT			

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 I wanted to demonstrate it to you through these examples. (Try performing these problem solutions in the calculate mode.) The type of problems you will be able to solve reliably through the several-stage process (problem definition, coding, program execution) using ASTROCAL is much, much more complicated than would be possible without ASTROCAL. The execution of some problems produces the equivalent result of hundreds or even thousands of automated keystrokes, perhaps involving decisions to be made, as well.

TRANSFER INSTRUCTIONS

The background relating to transfer instructions was first presented in the **GENERAL PROGRAMMING** section. However, this section enlarges on that basic information to help you fully understand the uses of the transfer instructions. There are two types: ***unconditional*** and ***conditional***.

UNCONDITIONAL TRANSFER INSTRUCTIONS

There are two types of unconditional transfer instructions: the go-to instruction **?**, "Goto", and the subroutine calls **>**, "Gosub". I 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 **?** followed by the four-digit address (program location number) of the destination, is encountered in the program. For example the sequence **?0324** causes immediate repositioning of the program counter to location 0324. The destination of a go-to instruction can also be specified by means of a label name following the go-to instruction. For example, the sequence **?s** would cause a transfer to the first location in the program after the sequence LABEL Sine.

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

CONDITIONAL TRANSFER INSTRUCTIONS

There are basically four different conditional transfer instructions available (in addition to the "Dec JPNZ", that is discussed later). They are "If flag", **#**, "If error", **$**, "If pos", **%**, and "If zero", **&**. Each of these has an inverse instruction as well. These four types of conditional transfer instructions all operate in the same manner. 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 four-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 four-digit address or label.

The inverse instructions to these four are realized by prefixing the basic conditional transfer instruction with **@**, "Inverse". These inverse conditional transfer 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.

TRANSFER INSTRUCTIONS

The eight types of tests made by these conditional transfer instructions are shown below:

If error	Is an error condition present (overflow, too small, invalid argument, invalid operation)? Transfer if the answer is yes.
Inverse If error	Is an error condition present? Transfer if answer is no.
If flag	Is the program flag specified by the next digit (0 through 9) set? Transfer if the answer is yes.
Inverse If flag	Is the program flag specified by the next digit set? Transfer if answer is no.
If pos	Is the content of the display-register positive (equal or greater than zero)? Transfer if the answer is yes.
Inverse If pos	Is the content of the display-register positive. Transfer if the answer is no.
If zero	Is the content of the display-register exactly zero. Transfer if the answer is yes.
Inverse If zero	Is the content of the display-register zero. Transfer if the answer is no.

These conditional transfer instructions do not affect pending operations, hence they can appear anywhere desired in the program, except between multi-location operands such as m35.

The If error and If flag conditional transfer instructions require a bit more explanation. You will note that an "If error" instruction tests for an error condition, that would cause "ERROR" flashing. Such an error will not halt execution of a program unless programmed to do so by the "If error" instruction. "ERROR" will continue to flash after the program halts unless CLR ERR or CLEAR were executed in the program before it halted. Using "CLR ERR" will stop "ERROR" flashing without affecting the displayed number.

The "If flag" instruction refers to the ten program flags that are indicators which can be set (turned on, raised, set to one) 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.

SETTING AND RESETTING PROGRAM FLAGS

The ten program flags are initially set to zero when you initialize ASTROCAL. 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 Set flag, **"**, followed by the single-digit (0 through 9) identification number of the flag affected. They are reset by means of the inverse instruction Inverse Set flag, **@"**, followed by the single-digit 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 conditional transfer 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 three uses, assume that you wish to run a program that normally prints out four or five different quantities. However, at times some of these 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 eighteen instructions (which is abbreviated [S18]) and then go to location 0345. If the results are not positive, you wish to execute the same set of eighteen instructions then not go to location 0345 but rather continue without a transfer. Thus, the point of the conditional transfer is eighteen steps beyond the point of the desired test. This problem is conveniently handled using the program flags as the following sequence demonstrates.

```
Location  Code  Instruction          Location  Code  Instruction
  0000    037    If pos                0011    096    LABEL
  0001    098    D.MS/D.d              0012    105    Indirect
  0002    064    Inverse                       ···    [S18]
  0003    034    Set flag              0031    035    If flag
  0004    055      7                   0032    055      7
  0005    063    Goto                  0033    048      0
  0006    105    Indirect              0034    051      3
  0007    096    LABEL                 0035    052      4
  0008    098    D.MS/D.d              0036    053      5
  0009    034    Set flag              0037    ···     etc.
  0010    055      7
```

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 the program counter only knows where it is and has no recollection as to how it came there. Yet such recollection 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 , (comma) "Reset" instruction resets all ten flags as well as clears the subroutine return-pointer registers and positions the program counter to the top of the program memory (0000).

CONDITIONAL TRANSFER EXAMPLES

You can test your understanding of the conditional transfer instructions by studying the following examples. Each example is a segment of code whose effect is given under "Explanation:"

Example:

```
Location  Code  Instruction
  0000    064    Inverse
  0001    036    If error
  0002    051      3
  0003    050      2
  0004    049      1
  0005    054      6
  0006    002    HALT
```

Explanation: If no error condition is present transfer is made to location 3216; otherwise, the program executes the next instruction (HALT). This is an example where you would halt execution if an error condition is present.

Example:

```
Location  Code  Instruction
  0000    035    If flag
  0001    056      8
  0002    048      0
  0003    048      0
  0004    050      2
  0005    050      2
  0006    116    Tangent
```

Explanation: If flag 8 is set, transfer is made to location 0022; otherwise, the program continues taking the tangent of the display-register contents.

Example:

```
Location  Code  Instruction
  0017    064    Inverse
  0018    037    If pos        [% key]
  0019    041      )
  0020    122    Sw MODE
  0021    096    LABEL
  0022    041      )
```

Explanation: If the contents of the display-register is positive then transfer to the location after the sequence **LABEL)**; otherwise, change the angular mode.

Example:

```
Location  Code  Instruction            Location  Code  Instruction
  0000    096    LABEL                  0010    096    LABEL
  0001    083    S                      0011    120    trace
  0002    038    If zero                0012    048      1
  0003    120    trace                  0013    096    LABEL
  0004    047      /                    0014    119    print
  0005    115    Sine                   0015    107    STO MEM
  0006    061      =                    0016    053      5
  0007    114      1/x                  0017    055      7
  0008    063    Goto                   0018    002    HALT
  0009    119    print
```

Explanation: This is an entire program for computing y = (sin x)/x. The test for x = 0 causes the execution to bypass much of the calculation if x = 0, arriving at the location behind the sequence **LABEL trace** (location 0012). If x <> 0, x/(sin x) is formed. Then we form (sin x)/x and skip over the 1 to store the answer into AM57 and then halt.

Example:

```
Location  Code  Instruction
  0000    037    If pos
  0001    044    reset
  0002    111      +/-
  0003    096    LABEL
  0004    044    reset
  0005    064    Inverse
  0006    113    Square
```

Explanation: If the display-register contents 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 basis conditional transfer instructions you can synthesize other basic conditional transfer instructions as shown in the table below:

Transfer when:	Instruction sequence		
A = B	(A - B)	If zero	
A <> B	(A - B)	Inverse	If zero
A > B	(B - A)	Inverse	If pos
A => B	(A - B)	If pos	
A < B	(A - B)	Inverse	If pos
A <= B	(B - A)	If pos	

Example: Suppose you wish to determine if the display-register content, Q, is less in magnitude than the quantity J stored in AM24. If that is the case, you wish to recall AM45 and halt execution; otherwise, you wish to add Q to AM45 and go to location 4321. This could be accomplished as follows:

```
Location  Code  Instruction
  0000    058      x          Begin formation of quantity for
  0001    040      (          testing. x stores Q into internal
  0002    113    Square       processing register for safekeeping.
  0003    064    Inverse
  0004    113    Square       Square-Inverse-square performs |Q|
  0005    045      -
  0006    109    RCL MEM
  0007    050      2
  0008    052      4
  0009    041      )          (|Q| - J)
  0010    064    Inverse
  0011    037    If pos       Transfer on |Q| < J
  0012    008    $ hold $     to point labeled hold
```

```
0013    049      1           Overwrite |Q| - J with 1.
0014    061      =           Complete pending multiply of Q:
0015    117    SUM MEM       Q x 1 = Q.
0016    052      4
0017    053      5           Q added to AM45
0018    063    Goto
0019    052      4
0020    051      3
0021    050      2
0022    049      1           Transfer
0023    096    LABEL
0024    008    $ hold $
0025    094    CLEAR
0026    109    RCL MEM
0027    052      4
0028    053      5
0029    002    HALT          When |Q| < J
```

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 conditional transfer 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.

DECREMENT AND JUMP NON-ZERO (Dec JPNZ)

A powerful instruction, especially useful in programming iterative routines, is the decrement and jump non-zero command. At this point, I will assume that AM00 contains an integer. If this is not the case, the effect is as though AM00 contains the next larger integer. The effect of the "Dec JPNZ", ', is the following:

1. First decrement the magnitude of the quantity stored in AM00 by 1.

2. If the resulting contents of AM00 is zero, do not transfer to the specified address or label, but fall through to the next instruction.

3. If the resulting contents of AM00 is not zero, transfer (jump) to the specified address or label.

The "Dec JPNZ" instruction also has its inverse, obtained by the sequence @', "Inverse Dec JPNZ". 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 "Dec JPNZ" function and polar/rectangular conversions are both used in the same program, it will be necessary to temporarily store the contents of AM00 in another addressable memory register, perform the polar/rectangular conversion, and return the proper data for the "Dec JPNZ" function to AM00.

To illustrate the use of the **Dec JPNZ** instruction, consider the following simple examples.

Example: Using the **Dec JPNZ** instruction, compute the sum of all integers 1 through N. (Enter the value N, then press U.)

Solution:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0011	117	SUM MEM
0001	085	U	0012	048	0
0002	093	CLR MEM	0013	049	1
0003	107	STO MEM	0014	039	Dec JPNZ
0004	048	0	0015	117	SUM MEM
0005	048	0	0016	109	RCL MEM
0006	096	LABEL	0017	048	0
0007	117	SUM MEM	0018	049	1
0008	109	RCL MEM	0019	002	HALT
0009	048	0			
0010	048	0			

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

$$y = \sum_{k=1}^{N} \log_e N$$

Solution:

Location	Instruction	Location	Instruction	Location	Instruction
0000	LABEL	0008	1	0016	0
0001	A	0009	LABEL	0017	1
0002	STO MEM	0010	/	0018	Dec JPNZ
0003	0	0011	RCL MEM	0019	/
0004	0	0012	0	0020	RCL MEM
0005	0	0013	0	0021	0
0006	STO MEM	0014	LOG e	0022	1
0007	0	0015	SUM MEM	0023	HALT

Like the other transfer instructions, the destination used in a **Dec JPNZ** instruction can be a label or a four-digit absolute address. Also, the **Dec JPNZ** performs the test on the value in AM00 without recalling it from addressable memory. Thus, pending operations are completely unaffected.

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 passes back to the main (or calling) routine at the next instruction past the point of the call.

Such sub-processes that are invoked and then pass control back to the calling sequence of code are known as subroutines. In ASTROCAL, subroutines called by the main routine may themselves call subroutines which return control to them just as though the first-level of subroutines were a main routine. Finally, upon invoking all required second-level subroutines and completing their assigned tasks, the first-level subroutines passes control back to the main routine. This main routine can then call additional subroutines until the problem is complete. In addition, the second-level subroutines may call third-level subroutines, which can call fourth-level subroutines, which can call fifth-level subroutines, etc. up to a seventy-second level subroutine. 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 "Gosub", **>**, followed by the label name of the subroutine being called. Thus **>s** calls the subroutine labeled Sine, or **>h** calls the subroutine labeled Hyperbol.

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.

$$27X^5 - 4X^4 + 32X^3 + 42X^2 - 8 = ?$$

Solution: Assume X_1 is stored in AM01, X_2 is stored in AM02, and X_3 is stored in AM03. This problem is solved without duplicating the code to evaluate the fifth-degree polynomial by using a subroutine, that I have named paper. The code for the subroutine might be defined essentially as you write the polynomial:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0020	109	RCL MEM
0001	097	paper	0021	052	4
0002	040	(	0022	050	2
0003	107	STO MEM	0023	091	Exponent
0004	052	4	0024	051	3
0005	050	2	0025	058	x
0006	091	Exponent	0026	051	3
0007	053	5	0027	050	2
0008	058	x	0028	059	+
0009	050	2	0029	109	RCL MEM
0010	055	7	0030	052	4
0011	045	-	0031	050	2
0012	109	RCL MEM	0032	113	Square
0013	052	4	0033	058	x
0014	050	2	0034	052	4
0015	091	Exponent	0035	050	2
0016	052	4	0036	045	-
0017	058	x	0037	056	8
0018	052	4	0038	041	)
0019	059	+	0039	060	Return

Notice in the 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 **=**.

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 program code calls the subroutine to evaluate the polynomial for each value of X and sums the results.

Location	Code	Instruction	Location	Code	Instruction
0040	096	LABEL	0051	062	Gosub
0041	086	V	0051	097	paper
0042	109	RCL MEM	0052	059	+
0043	048	0	0053	109	RCL MEM
0044	049	1	0054	048	0
0045	062	Gosub	0055	051	3
0046	097	paper	0056	062	Gosub
0047	059	+	0057	097	paper
0048	109	RCL MEM	0058	061	=
0049	048	0	0059	002	HALT
0050	049	2			

In this program, the subroutine is called by the sequence **Gosub paper**, where **paper** is the label for the subroutine. In the subroutine, the last instruction **<**, "Return", returns control to the calling routine past the point of the call. The foregoing example could also have been solved with a **Dec JPNZ** instruction in combination with indirect addressing, an advanced technique discussed in the next section.

Example: Construct a program to compute the expression $Y1^{Y2} + (\log_e Y3) * (Y4 + e^{Y5})$, where each of the quantities Y1 through Y5 is sufficiently complicated that its computation requires a subprocess.

Solution:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0011	058	x
0001	065	A	0012	040	(
0002	062	Gosub	0013	062	Gosub
0003	103	EXC MEM	0014	117	SUM MEM
0004	091	Exponent	0015	059	+
0005	062	Gosub	0016	062	Gosub
0006	107	STO MEM	0017	118	PROD MEM
0007	059	+	0018	064	Inverse
0008	062	Gosub	0019	110	LOG e
0009	109	RCL MEM	0020	061	=
0010	110	LOG e	0021	002	HALT

This solution uses the subroutines labeled **EXC MEM**, **STO MEM**, **RCL MEM**, **SUM MEM**, and **PROD MEM** just as it would use the quantities Y1 through Y5. In other words, the main routine is programmed as one would normally write algebra. Everywhere that **Gosub RCL MEM** appears, the subroutine provides the actual value of Y3 to the main routine.

It is irrelevant where the subroutine is stored relative to the location of the calling routine. In fact, except for the obscurity of program structure, 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 a subroutine pertains to subroutines labeled A through Z. Subroutines with these labels may be called by just their names. That is, you may omit the **Gosub** instruction.

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

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0004	066	B
0001	082	R	0005	091	Exponent
0002	065	A	0006	067	C
0003	059	+	0007	061	=
			0008	002	HALT

The third method enables you to call unlabeled sequences of code as subroutines. To do this, use **Gosub** followed by a four-digit address of the first program location in the desired subroutine. Thus **Gosub 8231** calls the subroutine beginning at location 8231.

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 practice, however, because it adds to the clarity of the program code —— particularly if the labels are chosen well and the program documentation records the meaning of each label. In addition, by labeling subroutines you can write the code that calls the subroutines before you know where those subroutines will be positioned. This also means inserting 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 = IN SUBROUTINES

One should use **=** (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, I easily created a program to compute $\mathbf{a + b^c}$ with calls to subroutines A, B, and C. I did not bother to define the subroutines in the example. Now imagine that subroutine B contained an equals instruction. 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 I would have **(a+b)** at that point. Whatever then occurred in subroutine C, would produce the wrong answer. If subroutine C had no equals instruction, I would obtain $\mathbf{(a+b)^c}$ as the final result.

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

Whenever the subroutine requires repeated access to Q (the display-register contents at the time of the call), the subroutine may include a store instruction prior to performing arithmetic. If Q 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:

```
Location  Code  Instruction
  1234    096    LBL
  1235    088    X
  1236    040      (
  1237    109    RCL MEM
  1238    059      +
  1239    ···    etc.
```

THE RETURN INSTRUCTION

The last instruction in a subroutine, the one that returns control to the calling routine, is always a return instruction **<**, "Return". I have 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 routine.

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 after the return instruction.

Occasionally programs are designed so that completion of execution can occur inside a 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. 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 ASTROCAL is re-initialized, the very next time a return instruction (used for HALT) is encountered in a new problem, any pending returns remaining from the last problem will be satisfied. This would rarely be the intended effect, and an improper execution would result. To prevent such leftover return pointers from ruining proper execution of the next problem, the reset instruction **,** (comma) "Reset" should be used to reset the return-pointer registers. This may be done manually, but it is preferable, when possible, to include the reset instruction at the proper point in the program code, with a halt instruction at location 0000.

SUBROUTINE PRACTICE PROBLEMS

Example: Construct a subroutine to take the integer part of Q and place the fractional part into AM33. It should be so designed that it can be used without disturbing pending operations in the calling routine.

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

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0010	102	Fix Dec.
0001	073	I	0011	048	0
0002	040	(	0012	048	0
0003	107	STO MEM	0013	101	EE
0004	051	3	0014	064	Inverse
0005	051	3	0015	117	SUM MEM
0006	045	-	0016	051	3
0007	046	.	0017	051	3
0008	053	5	0018	060	Return
0009	041	)			

You will notice that this subroutine does leave the display in scientific notation 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: I can use Newton's method of solution. (This well known method does 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 is estimated by finite differences:

$$f'(X_n) = \frac{f(X_n + \delta) - f(X_n - \delta)}{2\delta}$$

Preserving all data registers except AM98 and AM99 for you to use in the definition of f(x), I will produce the following main routine that you should find not only instructive as to the use of subroutines but useful in your problem solving as well.

Location	Instruction	Location	Instruction	Location	Instruction
0000	LABEL	0028	EE	0056	EE
0001	Z	0029	8	0057	8
0002	STO MEM	0030	+/-	0058	+/-
0003	9	0031	)	0059	)
0004	8	0032	)	0060	SUM MEM
0005	HALT	0033	N	0061	9
0006	LABEL	0034	-	0062	9
0007	X	0035	(	0063	If pos
0008	STO MEM	0036	RCL MEM	0064	+
0009	9	0037	9	0065	+/-
0010	9	0038	9	0066	LABEL
0011	LABEL	0039	x	0067	+
0012	C	0040	(	0068	-
0013	(	0041	1	0069	RCL MEM
0014	RCL MEM	0042	+	0070	9
0015	9	0043	EE	0071	8
0016	9	0044	8	0072	=
0017	N	0045	+/-	0073	If pos
0018	/	0046	)	0074	C
0019	(	0047	)	0075	RCL MEM
0020	(	0048	N	0076	9
0021	RCL MEM	0049	)	0077	9
0022	9	0050	x	0078	Inverse
0023	9	0051	RCL MEM	0079	EE
0024	x	0052	9	0080	HALT
0025	(	0053	9	0081	LABEL
0026	1	0054	x	0082	N
0027	-	0055	2		

You can save this useful routine to disk leaving the subroutine labeled N user programmable for various problems.

SUBROUTINES

The instructions for using this program are as follows:

1. Press **CTRL·p** to enter the program mode.

2. Load the program from disk (**CTRL·l** then filespec).

3. Press **CTRL·w** to go to location 0083.

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 cannot use = in this subroutine, and all addressable memory registers except AM98 and AM99 are available.

5. End your subroutine with a Return instruction.

6. Press **CTRL·p** to return to the calculate mode.

7. Input the desired accuracy (.001, .0000001, etc.) and press **Z**.

8. Input guess answer and press **X** to execute using the guess answer as a starting point. If the problem converges you will receive an answer.

Using this example, suppose you wished to solve for x such that $\log_e x = 0.2x$. Therefore, $f(x) = 0.2x - \log_e x$ and the subroutine at N could be as follows:

Location	Code	Instruction	
0081	096	LABEL	The first two lines shown here
0082	078	N	are part of the saved program.
0083	107	STO MEM	
0084	048	0	
0085	049	1	
0086	040	(	
0087	109	RCL MEM	
0088	058	X	
0089	046	.	
0090	050	2	
0091	045	-	
0092	109	RCL MEM	
0093	048	0	
0094	049	1	
0095	110	LOG e	
0096	041	)	
0097	060	Return	

The eight-digit answer to this problem is 1.2958555.

THIS PAGE INTENTIONALLY LEFT BLANK

INDIRECT INSTRUCTIONS

Every addressable memory register operation and every four-digit transfer instruction has a counterpart instruction in the 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 that you can use to accomplish processing that you simply could not otherwise do.

INDIRECT ADDRESSABLE MEMORY REGISTER INSTRUCTIONS

There are seven basic addressable memory register instructions:

1. STO MEM
2. RCL MEM
3. EXC MEM
4. SUM MEM
5. Inverse SUM MEM
6. PROD MEM
7. Inverse PROD MEM

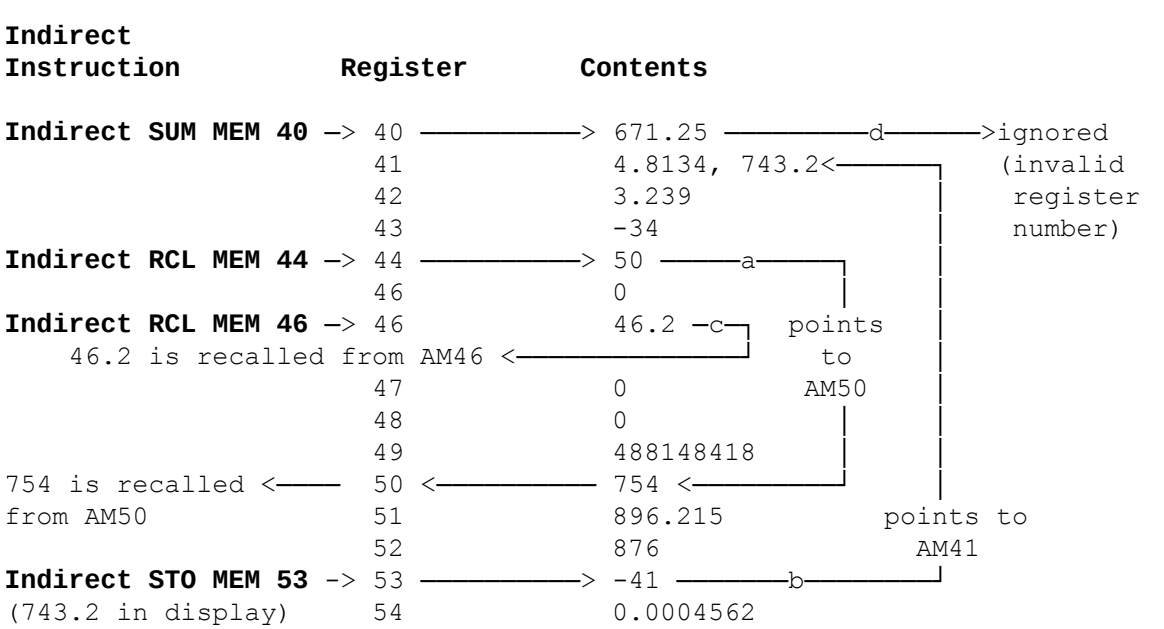
They all have one thing in common: In the instruction sequence that uses them, a two-digit number must appear in the sequence just after each one to designate the addressable memory register affected. The sequence **RCL MEM 62** would recall the value in AM62.

An indirect instruction is formed by preceding the normal direct instruction by **i**, "Indirect". What would be the effect of **Indirect RCL MEM 44**? This instruction sequence recalls the value not in AM44 (addressable memory register 44) but rather the value in the addressable memory register named by the contents in AM44. For example if the value stored in AM44 were 50 then **im44** would recall the value stored in AM50.

Whenever a value stored in an addressable memory register is not used as another number but as an address (in this case an addressable memory register number, hence an addressable memory register address) it is called a pointer. The pointer points to the addressable memory register; its address is the value of the pointer. In these indirect instructions, the content of the addressable memory register directly designated in the instruction is used as a pointer.

In the above example, **Indirect RCL MEM 44** means: "Use the contents of AM44 as a pointer to the addressable memory register whose contents are to be recalled." Similarly, **Indirect STO MEM 92** would mean: "Use the contents of AM92 as a pointer to the addressable memory register into which to store the display-register value." The indirect instructions use the absolute integer value of the addressable memory registers for the pointer. That is, if AM13 contained -33.999, then the instruction sequence of Indirect RCL MEM 13 would recall the contents of AM33.

The diagram below illustrates these concepts graphically.



This diagram shows the effect of **Indirect RCL MEM 44** as the path of events marked a: the result is to recall the value AM50 = 754. The effect of **Indirect STO MEM 53** is shown in the path of events marked b: the result is to store 743.2 into AM41, formally containing 4.8134. Finally, the result of **Indirect RCL MEM 46** in this example is marked c: the pointer points back to addressable memory register 46 so that the result is to recall the value AM46 = 46.2.

It is implied that any pointer used as the consequence of any indirect instruction must point to a realizable addressable memory register. Accordingly, the result of the **Indirect SUM MEM 40** in the example diagram would be ignored —— there is no addressable memory register 671.

Example: You enter a varying number of data items and have them stored successively in addressable memory registers AM01, AM02, etc., with the total number of items entered in AM00.

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0010	096	LABEL
0001	065	A	0011	066	B
0002	035	If flag	0012	105	Indirect
0003	048	0	0013	107	STO MEM
0004	066	B	0014	048	0
0005	049	1	0015	048	0
0006	107	STO MEM	0016	049	1
0007	048	0	0017	117	SUM MEM
0008	048	0	0018	048	0
0009	034	Set flag	0019	048	0
0010	048	0	0020	060	Return

INDIRECT INSTRUCTIONS

Example: Five quantities, X_1, X_2, X_3, X_4, and X_5 have been computed and stored (in order) into addressable memory registers AM01 through AM05. Five other quantities Y_1 through Y_5 have been similarly stored into AM06 through AM10. You wish to create a short segment of code in your program to compute the average value of the five quantities $Z_k = X_k/Y_k$ (k = 1, 2, 3, 4, 5).

Solution:

Location	Code	Instruction
0000	053	5
0001	107	STO MEM
0002	048	0
0003	048	0
0004	049	1
0005	048	0
0006	107	STO MEM
0007	049	1
0008	050	2
0009	048	0
0010	107	STO MEM
0011	049	1
0012	049	1
0013	096	LABEL
0014	110	LOG e
0015	040	(
0016	105	Indirect
0017	109	RCL MEM
0018	048	0
0019	048	0
0020	047	/
0021	105	Indirect
0022	109	RCL MEM
0023	049	1
0024	050	2
0025	041	)
0026	117	SUM MEM
0027	049	1
0028	049	1
0029	049	1
0030	064	Inverse
0031	117	SUM MEM
0032	049	1
0033	050	2
0034	039	Dec JPNZ
0035	110	LOG e
0036	040	(
0037	109	RCL MEM
0038	049	1
0039	049	1
0040	047	/
0041	053	5
0042	041	)

You could have coded the solution to this problem differently, but if your solution is concise, you doubtless used an indirect instruction. Now try doing the problem without indirect instructions and note the economy achieved by using indirect instructions.

The next example illustrates not only the programming design considerations, but also demonstrates the desirable method of top-down design.

INDIRECT INSTRUCTIONS

Example: Design a program to build a ten-window histogram given that the data items are in the range a < x <= b, where a and b are inputs. You desire the program to store the results for the number of counts in each bin in addressable memory register 01 though 10. The data will be entered one at a time through **X**. This is a long problem, but it illustrates the top-down approach that makes things easy.

Solution: First write the main routine, assigning AM99 to contain the bin number that the current value of data contributes a count into.

MAIN ROUTINE

Location	Code	Instruction
0000	096	LABEL
0001	088	X
0002	067	C
0003	107	STO MEM
0004	057	9
0005	057	9
0006	049	1
0007	105	Indirect
0008	117	SUM MEM
0009	057	9
0010	057	9
0011	002	HALT

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

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

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

where INT[y] = "integer part of y." Allocating AM98 and AM97 to a and b respectively, one can write the following without difficulty.

BIN NUMBER SUBROUTINE

Location	Code	Instruction	Location	Code	Instruction
0012	096	LABEL	0028	045	-
0013	067	C	0029	109	RCL MEM
0014	040	(	0030	057	9
0015	040	(	0031	056	8
0016	040	(	0032	041	)
0017	107	STO MEM	0033	041	)
0018	045	-	0034	068	D
0019	109	RCL MEM	0035	058	x
0020	057	9	0036	049	1
0021	056	8	0037	048	0
0022	041	)	0038	059	+
0023	047	/	0039	049	1
0024	040	(	0040	041	)
0025	109	RCL MEM	0041	060	Return
0026	057	9			
0027	055	7			

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? I 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 you can use the integer subroutine given as an example in the last section dealing with subroutines.

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

Location	Code	Instruction	Location	Code	Instruction
0042	096	LABEL	0050	102	Fix Dec.
0043	068	D	0051	048	0
0044	040	(	0052	048	0
0045	107	STO MEM	0053	101	EE
0046	045	-	0054	064	Inverse
0047	046	.	0055	101	EE
0048	053	5	0056	064	Inverse
0049	041	)	0057	102	Fix Dec.
			0058	060	Return

There are only two more details to complete and the problem is solved: I should make it convenient to enter the values of a and b and to initialize AM01 through AM10 to zero. Because it is natural to enter **a** at **A** and **b** at **B**, I write:

Location	Code	Instruction	Location	Code	Instruction
0059	096	LABEL	0066	096	LABEL
0060	065	A	0067	066	B
0061	093	CLR MEM	0068	107	STO MEM
0062	107	STO MEM	0069	057	9
0063	057	9	0070	055	7
0064	056	8	0071	002	HALT
0065	002	HALT			

and the problem is solved. Furthermore, the top-down method enabled 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 Indirect SUM MEM used to tally the counts in the ten bin-registers.

Next, I 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:

```
Location  Code  Instruction
  0000    105    Indirect
  0001    109    RCL MEM
  0002    048      0
  0003    049      1
  0004    107    STO MEM
  0005    057      9
  0006    057      9
  0007    105    Indirect
  0008    109    RCL MEM
  0009    057      9
  0010    057      9
```

What is the effect of this short instruction sequence? Imagine that the number 15 is stored in AM01, that the number 4 is stored in AM15, and that the number 1.41421356 is stored in AM04. The first indirect recall instruction automatically establishes that the pointer in AM01 points to AM15 and recalls its contents, the integer 4. This value is placed into AM99. The next indirect recall instruction then uses the pointer in AM99 to recall the contents of AM04, namely 1.41421356. 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 (AM01), use this pointer to find the location of the next pointer (AM15), and finally use the pointer found there to point and bring the actual number recalled to the display. It is invalid to code the sequence:

```
Indirect
Indirect
RCL MEM
  0
  1
```

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

INDIRECT PROGRAM-TRANSFER INSTRUCTIONS

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

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

Goto	Gosub
If flag n	Inverse If flag n
If error	Inverse If error
If zero	Inverse If zero
If pos	Inverse If pos
Dec JPNZ	Inverse Dec JPNZ

When any of these instructions are preceded by Indirect the absolute address for the transfer is found in the addressable memory register designated by the two-digit number in the instruction sequence.

Example: Suppose AM14 contains 2368.2133: Then the instruction sequence **Indirect Goto 14** would cause an unconditional transfer to location 2368.

Example: Suppose AM34 contains 7: Then the instruction sequence **Indirect Inverse If flag 3 34** would cause transfer to location 0007 if flag 3 is not set.

Example: AM19 contains 21: Then the sequence **Indirect Dec JPNZ 19** would cause a decrement of AM00, transfer to location 0021 if the result (decrementing of AM00) is non-zero, or execute the next instruction if the result is zero.

Example: The sequence **Indirect Gosub 23** would call the subroutine beginning at the location pointed to by the contents of AM23.

There is something to note about the indirect transfer instructions. They all eventually reach the destination address through the absolute location (0000 through 9999), and never by means of a label. One cannot store a label in the addressable memory register; but one can store a pointer to the destination program memory location. A number stored in an addressable memory register for indirect addressing need not be entered with leading zeroes and is the only exception to the four-digit requirement for specifying program locations. The indirect transfer specification requires two less digits in the sequence than the direct form. This results from the fact that two digits are required in the instruction sequence to specify the addressable memory register rather than the four digits necessary to specify the absolute transfer address.

Example: A quantity K is stored in AM77. 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, Address 4 if K = 4.

Solution: Now consider a specific case where Address 1 = 0162, Address 2 = 0064, Address 3 = 0111, and Address 4 = 0201.

Location	Instruction	Location	Instruction	Location	Instruction
0000	(	0014	7	0028	-
0001	RCL MEM	0015	7	0029	3
0002	7	0016	-	0030	)
0003	7	0017	2	0031	If zero
0004	-	0018	)	0032	0
0005	1	0019	If zero	0033	1
0006	)	0020	0	0034	1
0007	If zero	0021	0	0035	1
0008	0	0022	6	0036	Goto
0009	1	0023	4	0037	0
0010	6	0024	(	0038	2
0011	2	0025	RCL MEM	0039	0
0012	(	0026	7	0040	1
0013	RCL MEM	0027	7		

Although this may be a solution, it is not as concise as one available to you using indirect transfer instructions. Instead of the above code, you could store the numbers 162 into AM16, 64 into AM17, 111 into AM18, and 201 into AM19. Then the following code performs the case-statement transferring:

Location	Code	Instruction
0000	040	(
0001	109	RCL MEM
0002	055	7
0003	055	7
0004	059	+
0005	049	1
0006	053	5
0007	041	)
0008	107	STO MEM
0009	049	1
0010	048	0
0011	105	Indirect
0012	063	Goto
0013	049	1
0014	048	0

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.

INDIRECT INSTRUCTIONS

INDIRECT FIX-DECIMAL INSTRUCTION

Another sophisticated programming tool, the **Indirect Fix Dec.** instruction, enables you to use an addressable memory register to control the number of decimal digits displayed.

Example: Suppose AM46 contains 4.75: Then the instruction sequence **Indirect Fix Dec. 46** would be the same as the instruction sequence **Fix Dec. 04**. What this means is you can continually calculate the maximum number of decimal digits to be displayed, store the number in an addressable memory register, then use the **Indirect Fix Dec.** instruction to display the calculated number of decimal digits.

Whenever a value stored in an addressable memory register is not used as another data number but as a display control number it is called a formatter. The formatter controls the display format. In these indirect instructions, the content of the addressable memory register directly designated in the instruction is used as a formatter.

INDIRECT PRINT INSTRUCTIONS

Just as the **Indirect Fix Dec.** instruction uses an addressable memory register to format the display, the **Indirect print** instruction uses addressable memory to control printing data. The **Indirect print** instruction will be covered in the next section.

INDIRECT LOG 10 INSTRUCTION

The **Indirect LOG 10** instruction is the only "Indirect" instruction that does not use addressable memory. The **Indirect LOG 10** instruction is used to quickly obtain the magnitude of the displayed number. The result of an **Indirect LOG 10** instruction is the power-of-ten exponent of the displayed number, i.e., $\mathrm{INT}(\log_{10}Q)$.

Example: Indirect LOG 10 893.23923 = ?

Enter	**Press**	**Display**
893.23923	i l	2.

Example: Indirect LOG 10 .0323 = ?

Enter	**Press**	**Display**
.0323	i l	-2.

If the display contains zero, then an **Indirect LOG 10** instruction will return -9.999999999 9999 and "ERROR" will flash.

INDIRECT INSTRUCTIONS

Example: You wish to control the display format to control the number of significant digits displayed; however, if the number of decimals would to exceed a specific value, convert the display to scientific notation. The number of significant digits is entered using **B**, and the maximum number of decimals for non-scientific displays is entered using **C**. The corresponding code sequences for labels B and C can be executed early in the program, then, when the result is calculated (in the display-register), a transfer to label A is made to display the result in the desired format.

Location	Instruction	Location	Instruction	Location	Instruction
0000	LABEL	0036	=	0072	3
0001	B	0037	If zero	0073	Inverse
0002	STO MEM	0038	EE	0074	If flag
0003	0	0039	Inverse	0075	9
0004	4	0040	If pos	0076	Reset
0005	HALT	0041	EE	0077	Return
0006	LABEL	0042	LABEL	0078	LABEL
0007	C	0043	Pol/Rect	0079	Reset
0008	STO MEM	0044	RCL MEM	0080	EE
0009	0	0045	0	0081	Indirect
0010	5	0046	4	0082	LOG 10
0011	HALT	0047	-	0083	+
0012	LABEL	0048	1	0084	1
0013	A	0049	=	0085	-
0014	Inverse	0050	Set flag	0086	RCL MEM
0015	Set flag	0051	9	0087	0
0016	9	0052	EE	0088	2
0017	Inverse	0053	LABEL	0089	=
0018	Fix Dec.	0054	EE	0090	Inverse
0019	Inverse	0055	STO MEM	0091	If zero
0020	EE	0056	0	0092	Fix Dec.
0021	STO MEM	0057	1	0093	RCL MEM
0022	0	0058	+	0094	0
0023	3	0059	RCL MEM	0095	3
0024	Indirect	0060	0	0096	Inverse
0025	LOG 10	0061	5	0097	EE
0026	+	0062	=	0098	Return
0027	1	0063	Inverse	0099	LABEL
0028	=	0064	If pos	0100	Fix Dec.
0029	STO MEM	0065	Pol/Rect	0101	RCL MEM
0030	0	0066	Indirect	0102	0
0031	2	0067	Fix Dec.	0103	3
0032	-	0068	0	0104	EE
0033	RCL MEM	0069	1	0105	Goto
0034	0	0070	RCL MEM	0106	A
0035	4	0071	0		

This example stores the result in AM03. If the result would round to $1.00000 \times 10^{?}$, then the contents of AM03 could be modified.

PRINTING DATA

From the keyboard, the contents of the display-register can be printed at any time by pressing print, **W**. The same instruction encountered in the program code causes that action to take place in the execute mode. To illustrate this, consider the following program example.

Example:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0010	119	print
0001	071	G	0011	041	)
0002	040	(	0012	119	print
0003	109	RCL MEM	0013	059	+
0004	054	6	0014	109	RCL MEM
0005	052	4	0015	055	7
0006	047	/	0016	057	9
0007	109	RCL MEM	0017	119	print
0008	051	3	0018	061	=
0009	050	2	0019	119	print
			0020	002	HALT

In this example the following quantities are consecutively printed on separate lines before the program halts:

1. M32
2. (M64 ÷ M32)
3. M79
4. M79 + (M64 ÷ M32)

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

PROGRAMMING IMPLICATIONS

You have seen that you can print data without halting the executing program. Having the capability to print 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.

3. You may monitor where the program execution is currently taking place through printing clues 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.

PAPER ADVANCEMENTS

The paper can be advanced (line-feed) in the calculate mode or execute mode by the instruction paper, **a**. This feature is useful for separating groups of data or positioning the paper to top-of-form.

INDIRECT PRINT INSTRUCTIONS

The Indirect print instructions use addressable memory registers as a formatter for printing data. The **Indirect print** instructions do not send a carriage-return character (13 decimal) after each data is printed. This enables you to print more than one data on a print line.

The formatting of the printed data is controlled by the relative positions used in the addressable memory register and not the value of the contents. i.e., 23.455 creates the same format as 17.001. The printed data prints a character position for each digit (up to [DISPLAY SIZE]) in the addressable memory register plus one trailing space. If the data to be printed has fewer digits than the addressable memory register, then the data will print with leading spaces or trailing zeroes as appropriate. If the data to be printed has more decimal digits than the addressable memory register, then the additional digits will be truncated (not rounded). If the data to be printed has whole numbers and the format pattern does not have any whole digits, then the whole part will not print. However, if the data has more whole digits (for formatting, a negative sign uses a digit position) than the formatter then the whole part will print **?**'s as appropriate. And the exponent is only printed if the addressable memory register sign is negative. Several examples are given to demonstrate the **Indirect print** instructions.

Example: AM48 contains .345

Display-Register	**Instruction**	**Printed data**
8.23	i w 48	.230
94.34499	i w 48	.344
0.03239	i w 48	.032
-3.13	i w 48	.130
0.87234	i w 48	.872
-89.789923	i w 48	.789
4.23933 1234	i w 48	.239

Example: AM48 contains 34.345

Display-Register	**Instruction**	**Printed data**
8.23	i w 48	8.230
94.34499	i w 48	94.344
0.03239	i w 48	0.032
-3.13	i w 48	-3.130
0.87234	i w 48	0.872
-89.789923	i w 48	??.789
4.23933 1234	i w 48	4.239

Example: AM57 contains 892.238247

Display-Register	Instruction	Printed data
44.324	i w 57	44.324000
2.33764 9323	i w 57	2.337640
-7.7	i w 57	-7.700000
0.4892	i w 57	0.489200
111.11	i w 57	111.110000
-892.238247	i w 57	???.238247

Example: AM03 contains -2.45886

Display-Register	Instruction	Printed data
3.233	i w 03	3.23300{seven spaces here}
3.233 0123	i w 03	3.23300 0123
3.233 -0123	i w 03	3.23300 -0123
-3.33848 3202	i w 03	?.33848 3202

Example: AM23 contains 10000.00001

Display-Register	Instruction	Printed data
3.233	i w 23	3.23300
3.233 0123	i w 23	3.23300
3.233 -0123	i w 23	3.23300
-3.33848 3202	i w 23	-3.33848
-3345.3234	i w 23	-3345.32340
-83234.9999	i w 23	?????.99990
-9.99999999	i w 23	-9.99999

Example: AM23 contains 99999.99999

Display-Register	Instruction	Printed data
3.233	i w 23	3.23300
3.233 0123	i w 23	3.23300
3.233 -0123	i w 23	3.23300
-3.33848 3202	i w 23	-3.33848

Example: AM23 contains 0

Display-Register	Instruction	Printed data
849.39494	i w 23	{one space}

A zero in an addressable memory register is the way to print one space, regardless of the contents of the display-register.

Circumspectly use of the DOS Library Command FORMS along with the **paper** and **Indirect print** commands enable you to control the position where data is printed.

Example: You want to print a four place table of common logarithms from 1.00 to 9.99 with the characters 10, 11, 12, etc. to 99 printed in the left column.

Solution:

Location	Code	Instruction	Location	Code	Instruction
0000	096	LABEL	0046	048	0
0001	065	A	0047	101	EE
0002	049	1	0048	064	Inverse
0003	048	0	0049	101	EE
0004	107	STO MEM	0050	045	-
0005	048	0	0051	109	RCL MEM
0006	050	2	0052	048	0
0007	046	.	0053	049	1
0008	048	0	0054	061	=
0009	048	0	0055	064	Inverse
0010	048	0	0056	038	If zero
0011	049	1	0057	122	Sw Mode
0012	107	STO MEM	0058	097	paper
0013	048	0	0059	096	LABEL
0014	051	3	0060	120	trace
0015	034	Set flag	0061	109	RCL MEM
0016	049	1	0062	048	0
0017	057	9	0063	049	1
0018	048	0	0064	105	Indirect
0019	048	0	0065	119	print
0020	107	STO MEM	0066	048	0
0021	048	0	0067	050	2
0022	048	0	0068	096	LABEL
0023	096	LABEL	0069	122	Sw Mode
0024	066	Sine	0070	109	RCL MEM
0025	049	1	0071	048	0
0026	048	0	0072	049	1
0027	048	0	0073	047	/
0028	048	0	0074	049	1
0029	045	-	0075	048	0
0030	109	RCL MEM	0076	061	=
0031	048	0	0077	102	Fix Dec.
0032	048	0	0078	048	0
0033	061	=	0079	052	4
0034	047	/	0080	108	LOG 10
0035	049	1	0081	105	Indirect
0036	048	0	0082	119	print
0037	061	=	0083	048	0
0038	107	STO MEM	0084	051	3
0039	048	0	0085	064	Inverse
0040	049	1	0086	034	Set flag
0041	035	If flag	0087	049	1
0042	049	1	0088	039	Dec JPNZ
0043	120	trace	0089	066	Sine
0044	102	Fix Dec.	0090	097	paper
0045	048	0	0091	002	HALT

ERROR CONDITIONS

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

Too-small and Overflow

When a calculation results in a non-zero quantity and the magnitude is less than 1 x 10^{-9999} a too-small condition exists and "ERROR" flashes with 1. -9999 in the display-register. Similarly, if a magnitude equal to or greater than 10 x 10^{9999} should occur, "ERROR" will flash and the display-register will contain 9.999999999 9999 to indicate an overflow condition.

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.

Function Argument Outside of Range

The mathematical functions have certain restrictions placed on their arguments in addition to those imposed by the overflow and too-small criteria. The functions, invalid arguments, and the display-register contents are summarized below.

Function	**Invalid Argument**	**Display-Register**
$\sqrt{x}$	x < 0	$\sqrt{\lvert x \rvert}$
arc cosh x	-1 < x < 1	|x|
arc cosh x	x <= -1	arc cosh |x|
arc cosine x	|x| > 1	x
arc sine x	|x| > 1	x
arc tanh x	|x| = 1	Overflow
arc tanh x	|x| > 1	x
cosine x	|x| => $10^{[precision]}$	|x|
cosh x	x > 23025.8509299···	Overflow
$\log_{10}x$	x <= 0	$\log_{10}\lvert x \rvert$
$\log_{e}x$	x <= 0	$\log_{e}\lvert x \rvert$
sine x	|x| => $10^{[precision]}$	|x|
sinh x	x > 23025.8509299···	Overflow
tangent x	|x| => $10^{[precision]}$	x
tanh x	x > 23025.8509299···	1
x!	x < 0 or non-integer	|INT(x)|! where INT(x) is the integer part of x.
x!	x => 3249	Overflow
x^{y}	x < 0, y is non-integer	$\lvert x \rvert^{y}$ sign per INT(y)
y root of x	x < 0	y root of |x|
y root of x	x = 0, y = 0	1

Exceeding Capacity of Internal Registers

The internal processing registers can accommodate up to 85 (see page 1 for MAX-80 and the Model I/III) pending operations, and the parenthesis register can accommodate up to 36 open left parentheses. Any calculation attempting to exceed these maxima results in an error indication. "ERROR" flashes, and the extra operator or extra left parenthesis is ignored.

ERROR CONDITIONS

Undefined Transfer

Whenever an attempt is made to transfer to an undefined location, "ERROR" will flash with the current contents of the display-register. The program counter will proceed to execute the next instruction. Examples of such errors are:

1. Attempting to transfer to a label address that has not been defined by means of a label instruction.
2. Indirect transfers where the pointer is greater than 9999.

Attempt to Execute past Location 9999

Whenever the program counter reaches location 9999, if there is no HALT, Return, Reset or other transfer instruction there, an error similar to Undefined Transfer occurs except the program counter will remain at location 9999, and ASTROCAL will transfer to the command mode.

Improper 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 MEM 01 +)	operand missing after +
4.3 ÷ =	operand missing after ÷ (= is ignored)

Such sequences with) or **=** immediately following an arithmetic operator or with two consecutive operators (or two-variable functions) not separated by an operand are improper. "ERROR" flashes with the display-register indicating the current quantity.

Clearing an Error Condition

Pressing **SPACE**, "CLR ERR", stops the flashing "ERROR". In most cases this 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, and quantities stored in addressable memory registers. However, there is one error condition that remains latent after the flashing "ERROR" has been stopped with "CLR ERR". When direct addressable memory register arithmetic results in too-small or overflow, the error condition remains until the contents of that addressable memory register is changed.

Errors Encountered in the Execute Mode

When any of the foregoing errors occur in the execute mode, what happens next depends upon the programmer. Program halts are not an automatic consequence of an error condition. "ERROR" flashes, the program continues, using the value in the display-register for subsequent calculations. The answer may or may not be the correct answer, depending upon the problem and the type of error condition. However, it is the best selection that 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 conditional transfer instructions. On the other hand, your problem may be one that executes until it reaches an error condition; if correctly programmed, it would know what direction to take.

GLOSSARY

Address – A location in program memory, designated by either an absolute address or a label assigned in a program.

Addressable Memory Register – One of the 100 storage areas in memory.

Algebraic Hierarchy – The rules providing a unique interpretation of an expression that lacks a completely definitive set of parentheses.

Angular Mode – The two options, degrees or radians, in which angles are to be expressed.

Back-step – To decrement the program counter in the program mode for editing purposes.

Blank Instruction – An instruction that does nothing. This is used during step insertion in program editing. When ASTROCAL is initialized, program memory is filled with blank instructions.

Calculate Mode – The type of ASTROCAL operation in which calculations are performed under step-by-step control from the keyboard.

Case Statement – A type of programming element wherein transfer is to be made to one of n prescribed locations, depending upon a value (1 through n) of a control variable K.

Chain Operation – 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, clear display and pending operations, clear addressable memory only, or reset program flags and clear subroutine return-pointer registers.

Code – To write down the step-by-step instructions of a program. Or, the three-digit representation of each key based upon its decimal value.

Conditional Transfer Instruction – A decision making program statement offering a choice of ways to continue processing.

Control Flag – On/off program switches number 0 to 9 used as markers for various events in a program. Two-state devices that can be changed from the keyboard or in a program.

Coordinate Transformation – Conversion from polar coordinates to rectangular coordinates or vice versa.

GLOSSARY

Decision - The results obtained through use of the conditional transfer instructions.

Degree Mode - One of the two options, degrees or radians, in which angles are to be expressed.

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.

Direct Register Arithmetic - Performing addition, subtraction, multiplication, or division upon the contents of an addressable memory register, leaving the answer in that register, without recalling the register contents from memory.

Display - The n digit representation of the display-register, plus a four-digit scientific notation exponent.

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 that contains the quantity most recently computed, recalled from memory, or entered from the keyboard.

Dummy Operation - Those program steps that serve solely to supply the current display-register value as an operand.

Editing - The process of altering, adding, and deleting instructions as a final process of creating a working and satisfactory program.

Error Condition - A variety of situations that arise when the calculation encounters ill-defined quantities, improper operations, or numbers beyond the capacity of ASTROCAL.

Exchange - The operation in which the content of the display-register is exchanged with that of a specified addressable memory register.

Execute Mode - The ASTROCAL type of operation in which execution is under program control.

Execution - The phase during which ASTROCAL is running under program control. The controlling program is said to be in the process of execution.

Exponent - As used here, the power-of-ten associated with scientific notation number representation.

Exponentiation - A two-variable function for raising x to the y power.

Expressions - Instruction sequences that acquire a value depending upon register contents and which, when taken alone, leave no operation pending or operators unsatisfied. Expressions set off by parentheses may be combined with other such operations to form larger expressions.

Extraction of Roots - A built-in two-variable function for obtaining the y root of x.

GLOSSARY

Flags (Program Flags) - On/off program switches number 0 to 9 used as markers for various events in a program. Two-state devices that can be changed from the keyboard or in a program.

Flow Chart - A programming design device that graphically charts the paths of processing through a program.

Format - Either Display Format that is the manner in which numbers are being displayed. Or Mantissa Format that is a 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.

Functions - The keyboard mathematical operations of one/two variables.

Function Key - One of the keys labeled F1, F2, or F3.

Improper Instruction - Sequences that are not proper in ASTROCAL discipline.

Indirect Instruction - Any instruction that uses the contents of a specified addressable memory register as a pointer to the actual addressable memory register, program address or print image format.

Initial Display Format - The display mode invoked when ASTROCAL in initialized. It uses the initial mantissa format, but does not indicate scientific notation.

Initial Mantissa Format - The type of display wherein up to n digits are used to represent a number. A decimal point can be present anywhere in the number, and trailing zeroes are suppressed.

Insert - A program editing procedure whereby one pushes down all instructions from the current location leaving a blank instruction. A new program instruction is then inserted to take its place.

Instruction - One or more key steps that define an action to be taken in a program.

Internal Processing Registers - The 85 registers which, along with the display-register, are used by ASTROCAL to evaluate expressions with pending operations without affecting the addressable memory registers. The MAX-80 and Model I/III have fewer pending registers - see page 1.

Interruption - The act of halting program execution from the keyboard without precise knowledge of the state of processing at the time of intervention.

Inverse Operation - Those operations that result when an operation is prefixed by **@**, which reverses the effect of the operation.

GLOSSARY

Key Code – The three-digit representation of each key based on its decimal value.

Key, User-Defined - One of the 26 functions, A through Z that provide a starting point for program execution by merely pressing that key.

Label - A name assigned to a particular point in a program that can be referenced by a transfer instruction or by program initialization.

Levels of Parentheses - The number of operations made pending by means of open (left) parentheses.

List - To effect a step-by-step printed record of the instructions of a program.

Load Module - Any portion of a partitioned program that resides, in its entirety, in program memory at some given time.

Location - Positions in program memory, 0000 through 9999.

Logical Test - 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.

Magnitude - The numerical size of a number regardless of its sign.

Mantissa - The number in scientific notation that 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 - The generic term referring to both to program memory and addressable memory. Program memory contains the program steps to solve a given problem and is addressed 0000 through 9999. Addressable memory consists of the one hundred addressable registers.

Memory Register - One of the 100 addressable storage areas in memory.

Modes of Operation - The calculate mode, program mode, and execute mode.

Operand - A number or numerical expression in a mathematical operation.

Operation - One of the four arithmetic functions, or one of the two-variable functions.

Operator - Any function that mathematically alters a number.

Overflow - The situation that results when the magnitude of a calculated number is equal to or greater than 10×10^{9999}.

Overwriting - To replace the contents of an addressable memory register or a step in program memory, with another value/step, obliterating the previous contents.

Parentheses - Devices used to set off expressions as in algebra, to ensure they will be evaluated properly before being combined with other expressions.

Pending Operation - Those operations that cannot immediately be completed —— pending evaluation of expressions opened by parentheses, or because of the algebraic hierarchy.

Pointer - A number that resides in addressable memory registers and is used to specify a program address or another addressable memory register.

Polar/Rectangular Conversion - Conversion from polar coordinates to rectangular coordinates.

Program - The logical sequence of keystrokes that are stored and executed from program memory in the execute mode, which 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 that keeps track of where ASTROCAL currently resides in the instruction sequence.

Program Flag - On/off program switches number 0 to 9 used as markers for various events in a program. Two-state devices that can be changed from the keyboard or in a program.

Program Instruction - One or more keystrokes that define an action to be taken in a program.

Program Location - Any of 10,000 available positions in program memory, 0000 through 9999.

Program Memory - 10,000 locations where a program can be stored.

Radian Mode - One of the two options, degrees or radians, in which angles are to be expressed.

Rectangular/Polar Conversion - Conversion from rectangular coordinates to polar coordinates.

Registers - A generic term for any calculation storage unit that can be use to hold a numeric value. See Internal Processing Registers, Addressable Memory Registers, and Return-Pointer Registers.

Reset - To restore to zero: especially to restore a flag to zero. Also, the instruction that resets all flags, resets the return-pointer registers, and positions the program counter to 0000.

Return - The transfer of control back to a calling program segment. If there is no call in effect, the transfer is back to the calculate mode.

Return Pointer - A pointer holding where to return control in a program after the processing sequence has been diverted to a subroutine.

Return-Pointer Registers - The 72 registers that provide the return pointers for subroutines.

Root Extraction - A built-in two-variable function for obtaining the y root of x.

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.

Scientific Notation - The method of representing a number by a mantissa, M (in the range $1 <= |M| < 10$), times a power-of-ten.

Single-Step - The process of execution or observing a program one step at a time.

Store - To place a copy of the contents of the display-register into a specific addressable memory register.

Subroutine - An isolated program segment used primarily for repetitive calculations. It returns to the calling routine upon completion of its task.

Subroutine Return Pointer - A pointer holding where to return control in a program after the processing sequence has been diverted to a subroutine.

Too-small - The situation obtained when the magnitude of a calculated number is greater than zero but less than 1×10^{-9999}.

Top-Down - The approach whereby a problem is solved in the large before details are filled in.

Trace - A capability for automatically displaying each step executed and its results.

Transfer Instruction - Those instructions that 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 make a test and either transfer or not depending upon the outcome of the test.

Unconditional Transfer - Program instruction that unquestioningly repositions the program counter to some out-of-sequence location.

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

X

Z