Programming and Working with Indirectly Addressed Memory on the HP-35s

Introduction

The HP-35s calculator is a continuation of the line of scientific calculators that began with the HP-45 in the 1970s. Its immediate ancestor is the HP-33S, a calculator still in production. One of the significant advances of the HP-35s beyond the HP-33S is the much greater access to the 30K of memory that the calculator possesses.

By being able to transfer control within a program to any program line, using XEQ or GTO, it is possible to do a great deal more in programming, without running out of program labels. In addition, the calculator can address 26 variables (identified with a letter), 6 statistical registers, and an additional 801 storage locations. This memorandum is primarily concerned with working with the 801 additional storage locations.

Accessing Memory

Memory locations that are accessible by letter, e.g., STO A and RCL A, may also be address indirectly. Similarly, the statistical registers may be addressed indirectly. The 800 additional storage locations must be addressed indirectly.

Indirect access to memory is done through two special memory locations, the variables I and J. These are ordinary memory locations, but the calculator can take the contents of the I and J locations and use it as the address of another memory location. In this sense, I and J can be used like pointers in regular programming.

To indirectly access a memory location, an integer is stored in I or J. To store a value in the memory location pointed to by the value in I or J, the key sequence STO (I) or STO (J) is used, respectively. This stores the value currently in the x register of the stack (the bottom of the stack, and the lower line of the display) into the memory location pointed to by the value in I or J, respectively. Similarly, RCL (I) and RCL (J) recall the value in the memory location pointed to by the value in I or J, respectively, and copies it to the x register of the stack. Register arithmetic, e.g. STO + (I), RCL ÷ (J), is also possible, and indirect memory locations can be used in the same way as ordinary memory locations. Note that I and J can be used in any way; there are two of them to make programming easier when working with pairs of values, such as co-ordinates of paired statistical data.

The (I) and (J) symbols cannot be entered using the parentheses key. They have their own keys, i.e., 0 and the decimal point keys. They can be seen there in red, acting like the letters for variables and labels. Use these keys to operate indirect memory addressing.

The addresses used in the I and J variables are as follows. To access the variables A through Z, use the numbers –1 through –26. The statistical registers use the numbers –27 through –32. The 801 additional locations use positive numbers 0 through 800.

Examples

Suppose that I want to store a value (say 123.456) into memory location 321. I put the value 321 (the memory location) into the x register of the stack and press STO I. Then I put 123.456 (the value I want to store) into the x register of the stack and press STO (I). The STO function then moves the value in the x register of the stack to the memory location specified in the I variable.

If I want to recall the value in memory location 230, I put 230 into the x register of the stack and press STO J. I then press RCL (J), and the value in memory location 230 is recalled to the x register of the stack.

Memory Allocation

The calculator has the memory locations A through Z and the statistical registers enabled at all times. However, the 801 additional memory locations are not allocated until needed, and can be de-allocated (freed) when no longer needed. As this indirectly addressed memory is shared with program memory, this allows it to be held for either purpose. However, the calculator has no explicit functions for allocating and de-allocating memory. This happens semi-automatically.

When a non-zero value is stored in a specific memory location (from the 801), using STO (I) or STO (J), memory is allocated for storage up to the memory location used for that STO operation. For example, if I store the value 123.456 in memory location 500, using STO (I) or STO (J), memory locations 0 through 500 will be allocated to storage and will be accessible. They should also be given initial values of zero.

So memory allocation takes place automatically, and there is no need to initialize the memory to zero. In fact, attempting to do so may cause an INVALID (I) or (J) error, because you tried to access memory that wasn’t yet allocated. This will happen if you try to STO a value of zero into memory that isn’t allocated.

However, if you are depending upon the STO function to allocate memory, remember that it requires a non-zero value being stored at a location to allocate that memory. If there is any possibility that the value that you are storing is zero, that memory location will not be allocated. This may not be a problem if you are simply storing a string of values, as unless the last value is zero, later non-zero values will allocate the memory and set it to zero.

A better solution is to decide about how much memory you will need, and allocate that ahead of time. That avoids problems with valid data values of zero, as well as initializing the memory locations to zero. If you make sure that there is an initial STO (I) or STO (J) operation, rather than STO+ or similar, then there will be a valid value in the memory location before you start any register arithmetic.

For a code example, try the following to allocate 101 memory locations (0 through 100):

```
100
STO I
STO (I)
```

You could insert some other value between the STO I and STO (I) lines, but the above code fragment will allocate all memory locations from 0 through 100 that are not already allocated, store the value 100 in memory location 100, and for those memory locations not already allocated, set their contents to zero.

De-allocating memory also has no explicit command on the calculator. Memory is de-allocated when it is returned to zero, but this cannot happen if there are still memory locations allocated above the memory location being set to zero (i.e., with a greater/higher address number). If you explicitly set the memory location at the top of allocated memory to zero, i.e., STO a value of zero in that memory location, that location will be de-allocated, but no other locations will be de-allocated, even if they are filled with zeros.

The solution to de-allocate memory is to set up a loop that sets memory locations to zero from the top down, such as the following code fragment, based on de-allocating the memory locations from 100 down to 20:

Line	**Instruction**
K240	100
K241	STO I
K242	0
K243	STO (I)
K244	20
K245	RCL I
K246	x < y?
K247	GTO K251
K248	1
K249	STO— I
K250	GTO K242
K251	PROGRAM END

Of course, care must be taken to see that the starting value is at the top of allocated memory, otherwise the code simply sets memory locations to zero without any de-allocation. Unfortunately, there appears to be no easy way to ascertain how many memory locations are currently allocated within a program. You can see it as the left number on the lower line of the display when the ↰ **MEM** function is used, but this function is not programmable. Note that this value is the number of memory locations allocated, so 25 means the ‘top’ memory location is 24, as memory locations 0 through 24 are allocated.

When programming to de-allocate memory locations, it is best to de-allocate only those that you are sure you allocated, otherwise you will get an INVALID (I) or (J) error. So there are advantages to being quite explicit about allocating and de-allocating memory locations. Remember also that if a program that allocates memory is halted before the end, it will not have de-allocated memory locations, so control may have to be moved to this section of the code and it executed to de-allocate the memory locations.

Why De-allocate Memory Locations?

The calculator’s 30K of memory is shared by programs, memory locations, equations, and a range of internal applications, such as SOLVE, and integration (∫ FN). These all use a fair amount of memory, integration quite a large amount (albeit only while running). To avoid a MEMORY FULL error, which will halt execution, it is wise to de-allocate memory locations that are not needed, so that they can be used by other calculator applications which need temporary memory.

Note also that if you have a lot of program memory space used, this may cut into the available memory locations, so that there may not be the full 801 available. Program memory appears to be allocated and de-allocated automatically, without any need for explicit operation on the user’s part.

Dr. Bill Hazelton.

March, 2008.