C68Port - LOCal Variables - The Internals

Contents

Introduction

The code in the `SBLocal` folder has been written in order that any SuperBASIC code, converted by what will (eventually) become `C68Port` can emulate the features of SuperBASIC `LOCal` variables within the genearated C68 application. However, because it is of some mildly useful use outside of a SuperBASIC conversion, it has been documented here - just in case.

If you have tried out the demo application, you will have seen how *relatively* easy it is to set up a C68 program that emulates the actions of the SuperBASIC `LOCal` variables. So how does it work?

You will know that in order to do this, you must set up a new *scope* for each C68 equivalent to a SuperBASIC PROCedure or FuNction, and you must end it on exit from the equivalent C68 function, as follows:

```
#include "SBLocal.h"

...

void proc_1() {
    SBLOCAL scope = beginScope();

    ...

    endCurrentScope();
}
```

You then create various LOCal variables within this scope, and these become visible to the current and all other scopes created by various called PROCedure or FuNction equivalents in your C68 code.

So how does it all work?

Internals - Structures

There are a couple of structures defined in SBLocal.h which can be used in a C68 application that needs to emulate SuperBASIC behaviour.

SBLocalVariableNode

Within each scope, LOCal variables are held together in a linked list. There is a single linked list for each *scope* and the root of each *scope* is held in an array of pointers to an SBLocalVariableNode (aka an SBLOCAL - see below). The array, SBLocalStack, is defined as static in SBLocal.c and is set to hold up to MAX_STACK_DEPTH entries. This gives up to 2048 levels of LOCal variables in any C68 application. MAX_STACK_DEPTH is easly changed by editing its definition in the file SBLocal.h.

Each new variable is added at the start of the linked list, so for minor efficiency reasons, put the variable you are going to be working with most frequently, at the end of the definitions, they will be found quicker when scanning the stack for a particular variable. For example:

```
SBLOCAL scope = beginScope();
...
SBLOCAL fred;
SBLOCAL wilma;
...
endCurrentScope();
```

The fred variable will be first on the list until wilma is added, then fred gets pushed down a level. This means that when looking for a variable, the scan begins at wilma and then goes to fred and so on, if there were more variables. By putting wilma last here, that one becomes the first one checked when looking for a variable, so if wilma is used more often than fred, put the declaration of wilma last, so that *she* ends up first on the list.

Each node in the linked list is defined as follows:

```
typedef struct SBLocalVariableNode {
    struct SBLocalVariable variable;
    struct SBLocalVariableNode *next;
} SBLocalVariableNode, *SBLOCAL;
```

A relatively simple node indeed, it contains nothing more than an embedded structure of type `SBLocalVariable` and a pointer to the next node in the list. The list is therefore implemented as a singly linked list. Analysis of what was required showed that there would not be any need to scan the list backwards, so a doubly linked list was not required.

The details of the `SBLocalVariable` structure are described next.

SBLocalVariable

This is the structure that describes, fully, each and every SuperBASIC `LOCal` variable equivalent for use in a C68 application. It has the following structure:

```
typedef struct SBLocalVariable {
    short variableType;
    char variableName[MAX_LOCAL_NAME_SIZE +1];
    union variableValue {
        short integerValue;
        double floatValue;
        void *arrayValue;
    } variableValue;
    unsigned short maxLength;
} SBLocalVariable;
```

The following fields are defined:

- `variableType` - describes the type of this variable. It sets the variable to be the equivalent of any SuperBASIC variable type such as floating point, integer, string, array etc.
- `variableName` - is simply an array of characters that holds the variable name, as per its declaration within the C68 application. There is a maximum size for a variable name, as defined by `MAX_LOCAL_NAME_SIZE` which can be changed in `SBLocal.h` if required. variables with names longer than this will simply have their names truncated.
- `variableValue` - a union, consiting of the actual value for an integer variable or a floating point variable. If the `LOCal` is a string or an array of some other type, then its value is actually a pointer to wherever the string or array lives in RAM. The `variableType` is used to determine which of the unioned fields is the actual one for the specific variable.
- `maxLength` - Used for strings and arrays to indicate the maximum size. If a string, for example, was declared as 10 characters, this field is used to limit any assignments to a maximum length of 10 characters. This is similar to how SuperBASIC operates with `DIM`ensioned strings.

SBLOCAL

`SBLOCAL` is simply a `#define` for a pointer to an `SBLocalVariableNode` and it is through these pointers that the system actually works. You use one of these, or, if you like typing, a `SBLocalVariableNode`

pointer, to start a new scope, and to declare a new LOCal variable. You don't have to use the actual pointer that is returned, so you could, if you wished, use a single pointer to declare any number of different LOCal variables, as follows:

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = beginScope();

    temp = LOCAL_INTEGER("fred");
    temp = LOCAL_FLOAT("wilma");
    temp = LOCAL_STRING("barney", 100);
    temp = LOCAL_STRING("betty", 250);
    ...

    endCurrentScope();
}
```

However, if you intend to manipulate these variables, within the proc_1 function, then it is perhaps better to keep a hold on the pointer returned:

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = beginScope();

    SBLOCAL fred = LOCAL_INTEGER("fred");
    SBLOCAL wilma = LOCAL_FLOAT("wilma");
    SBLOCAL barney = LOCAL_STRING("barney", 100);
    SBLOCAL betty = LOCAL_STRING("betty", 250);
    ...

    endCurrentScope();
}
```

Scope

The *scope* of a LOCal variable is the period of the application where that variable is visible to the code. In C (and therefore C68) applications, a variable's scope is the unit it was declared in. In the above example, then, the variables fred, wilma, barney and betty as well as temp are only accessible within proc_1 and not in any called function that proc_1 might happen to call - unless they are passed to the called code.

In SuperBASIC, LOCals are visible in any called PROCedure or FuNction. A totally different behaviour. (And hence this small part of the C68Port utility!)

Having said that, however, this part of C68Port does allow the variables declared above to be seen and manipulated from called C68 code, until the call to endCurrentScope is made. or example:

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = beginScope();

    SBLOCAL fred = LOCAL_INTEGER("fred");
    SBLOCAL wilma = LOCAL_FLOAT("wilma");
    SBLOCAL barney = LOCAL_STRING("barney", 100);
    SBLOCAL betty = LOCAL_STRING("betty", 250);
    ...

    proc_2();

    endCurrentScope();
}

void proc_2() {

    ...
    SET_LOCAL_INTEGER("fred", 666);
    ...
}
```

Because proc_2 doesn't have any LOCal variables of its own, it has no need to call beginScope or endCurrentScope, but it can still access variables declared at a higher level, those from proc_1 for example.

The most inefficient method of accessing a higher scope's LOCals is shown above. If you only need one single access, to set or read the value, then the above works fine. If you need to set or read a variable's value more than once, then the following is a much more efficient version of proc_2.

```
void proc_2() {

    SBLOCAL tempFred = FIND_LOCAL("fred");
    if (!tempFred) {
        /* Handle errors */
        return;
    }

    /* We have a pointer to the local variable fred. */
    setSBLocalVariable_i(tempFred, 666);
    ...
    setSBLocalVariable_i(tempFred, 616);
    ...
}
```

Starting a new Scope

When you call `beginScope` to start a new scope, a new `SBLOCAL` is created and if successful, is given the name "**ROOT**" and pushed onto the stack used to hold the existing scopes. This way, as we enter each called function in the C68 application, and searching for `LOCal` variables takes place by working its way from the current scope - at the top of the stack - and from there, backwards through each of the previous scope levels until it is found, or otherwise.

Each entry on the stack points to a list of variables which were declared as `LOCal` at that level of scope.

Ending a Scope

It is very important to end each scope that you started. If you forget to end a scope just prior to exiting from a C68 function, then the variables you created within that scope remain visible to the rest of the program. This is not the behaviour that a converted SuperBASIC program should be doing!

When you call `endCurrentScope`, the current scope's root pointer is popped off the top of the stack, and the linked list of `LOCal` variables is walked along, deleting all string and/or array data first, then deleting the node itself. This way, all `LOCal` variables for the scope just ending, are cleaned up.

Functions With Multiple Exit Points

Ah, the purists will be gnashing their teeth now that I mentioned that!

If you can write your code to have a single exit point, then just before you exit from the function, make a call to `endCurrentScope` and all will be well.

```
void proc_1() {

    SBLOCAL temp = beginScope();

    SBLOCAL fred = LOCAL_INTEGER("fred");
    ...

    endCurrentScope();
}
```

Remember, you only need to end a scope if you started one by calling `beginScope`. If you have no `LOCal` variables in your function, then there is no need to begin or end a new scope. It won't do any harm, however.

If you have (to have) multiple returns, then you should consider using a `goto` (that's the purists weeping into their real ale now!) and have a single exit point, as per the following contrived example:

```
void proc_77(int something) {

    SBLOCAL temp = beginScope();

    SBLOCAL fred = LOCAL_INTEGER("fred");
    ...
```

```
    switch (something) {
        case 0: goto endLocal;

        case 1: doSomething(x);
                goto endLocal;

        case 2: doSomethingElse(x);
                break;

        default:
                break;
    }

    /* We only get here if something was 2 or higher. */
    ...

    /* Lots of processing here! */
    doLotsOfProcessing(something);

    ...

endLocal:
    endCurrentScope();
}
```

So if the passed parameter was a zero, we simply have to return. That needs to `endCurrentScope` so we jump to the end of the function. If the parameter was one, then we call `doSomething` and then jump to the end as we are done processing.

If the passed parameter was a two, we `doSomethingElse` but then drop through into the code following the end of the `switch`. Any other value simply drops out of the `switch`.

We then do a lot more processing and finally, exit after calling the required `endCurrentScope`.

LOCal Variables

Creating LOCal Variables

You have already seen how to create `LOCal` variables in a C68 application:

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = beginScope();

    temp = LOCAL_INTEGER("fred");
    temp = LOCAL_FLOAT("wilma");
    temp = LOCAL_STRING("barney", 100);
    ...
```

```
    endCurrentScope();
}
```

The above example shows the creation of three separate `LOCal` variables. The creation of these returns a pointer (`SBLOCAL`) which need only be kept if the variable will be manipulated within the current scope. Normally you would set a value after creation - see below for details.

Deleting LOCal Variables

This is easy. You *don't*! Well, not directly. As with a SuperBASIC program, your `LOCal` variables cease to exists when the scope that they were defined in, ends. In a C68 application, the scope ends with a call to `endCurrentScope`. That will tidy up the entire list of `LOCal` variables declared at the current scope level.

Finding LOCal Variables

Reading Variable Values

```
#include <stdio.h>
#include "SBLocal.h"

...

void displayStuff() {

    int myFred = GET_LOCAL_INTEGER("fred");
    double myWilma = GET_LOCAL_FLOAT("wilma");
    char *myBarney = GET_LOCAL_STRING("barney");
    ...
    printf("LOCal Variables - Declared Elsewhere\n\n");
    printf("Integer fred = %d\n", myFred);
    printf("Float wilma = %f\n", myWilma);
    printf("String barney = '%s'\n", myBarney);
    ...
}
```

The version used above, reads well to anyone reading the source code, but is mildly inefficient as the calls to `GET_LOCAL_xxxx` need to search the current, and all previous, scopes looking for the first occurrence of any `LOCal` variable with the supplied name - which is, of course, case sensitive.

A more efficient method of reading a variable value, especially if you require to read it and/or write it within the same scope level, would be to use a pointer to set the value directly and without having to scan the scope for the variable's node in the linked list(s). This method is shown below and uses the `FIND_LOCAL` call to retrieve an `SBLOCAL` that points directly at the desired `LOCal` variable.

```
#include <stdio.h>
#include "SBLocal.h"
```

```
...

void displayStuff() {

    SBLOCAL tempFred = FIND_LOCAL("fred");
    SBLOCAL tempWilma = FIND_LOCAL("wilma");
    SBLOCAL tempBarney = FIND_LOCAL("barney");

    int myFred = getSBLocalVariable_i(tempFred);
    double myWilma = getSBLocalVariable(tempWilma);
    char *myBarney = getSBLocalVariable_s(tempBarney);
    ...
    printf("LOCal Variables - Declared Elsewhere\n\n");
    printf("Integer fred = %d\n", myFred);
    printf("Float wilma = %f\n", myWilma);
    printf("String barney = '%s'\n", myBarney);

    /* Do other stuff here with tempFred, etc */
    ...
}
```

Changing Variable Values

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = beginScope();

    temp = LOCAL_INTEGER("fred");
    SET_LOCAL_INTEGER("fred", 666);

    temp = LOCAL_FLOAT("wilma");
    SET_LOCAL_FLOAT("wilma", 3.14);

    temp = LOCAL_STRING("barney", 100);
    SET_LOCAL_STRING("barney", "Hello Fred!");
    ...

    endCurrentScope();
}
```

The version used above, reads well to anyone reading the source code, but is mildly inefficient as the calls to `SET_LOCAL_xxxx` need to search the current, and all previous, scopes looking for the first occurrence of any `LOCal` variable with the supplied name - which is, of course, case sensitive.

A more efficient method of setting a variable value, would be to use the pointer returned - in `temp` in the above example - to set the value directly, without having to scan the scope for the variable's node in the linked list(s). This method is shown below.

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = beginScope();

    temp = LOCAL_INTEGER("fred");
    setSBLocalVariable_i(temp, 666);

    temp = LOCAL_FLOAT("wilma");
    setSBLocalVariable(temp, 3.14);

    temp = LOCAL_STRING("barney", 100);
    setSBLocalVariable_s(temp, "Hello Fred!");
    ...

    endCurrentScope();
}
```

Indirect Variable Access

Indirect access to a variable allows you to simply use the variable's name, as it was declared when the `LOCal` was created at whatever scope level. This method of access *reads better* in an application's source code and is quite easy to understand. It is perfect for a single access to any `LOCal` variable, no matter how deeply nested in the scope levels it happens to be. You have already seen examples of indirect access to variables.

It is called indirect, as we have to find where the variable lives in RAM, and then access it.

```
#include <stdio.h>
#include "SBLocal.h"

...

void displayStuff() {

    int myFred = GET_LOCAL_INTEGER("fred");
    ...
    printf("LOCal Variables - Declared Elsewhere\n\n");
    printf("Integer fred = %d\n", myFred);
    ...
}
```

This method is quite inefficient as any time the variable is referenced, the scope stack is searched to find the most recent occurrence of any `LOCal` variable with the supplied name. However, as mentioned, if you only need a single access to the variable, it is not so bad.

Obviously, it's much better, at least in the function that the `LOCal` was declared in, to use the `SBLOCAL` pointer to set the newly declared variable's value, but it is not essential.

```
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = LOCAL_INTEGER("fred");
    setSBLocalVariable_i(temp, 666);
    ...
}
```

You cannot use the indirect method in a called function because the variable, `temp` above, is not accessible in a C68 application because of the rules of C scoping of variables. (Unless passed as a parameter of course.) In called functions, you must use the Direct Variable Access methods, described below.

Direct Variable Access

When you are in a function, called from another, which created some `LOCal` variables then you must use direct access to the variables by passing the name to various function calls to read or write the variable's value using the `GET_LOCAL_xxxx` and/or `SET_LOCAL_xxxx` function calls.

```
#include <stdio.h>
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = LOCAL_INTEGER("fred");
    setSBLocalVariable_i(temp, 666);
    ...
    proc_2();
}

void  proc_2() {
    printf("I see 'fred' is set to %d\n", GET_LOCAL_INTEGER("fred"));
    ...
}
```

And this is perfectly valid, and about as efficient as it gets, *unless* you need to access the variable more than once. The following is acceptable, and very readable too:

```
#include <stdio.h>
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = LOCAL_INTEGER("fred");
```

```
    setSBLocalVariable_i(temp, 666);
    ...
    proc_2();
    ...
    printf("I see 'fred' has been altered to %d\n", getSBLocalvariable_i(temp));

}

void  proc_2() {
    printf("I see 'fred' is set to %d\n", GET_LOCAL_INTEGER("fred"));
    SET_LOCAL_INTEGER("fred", 616);
    printf("I altered 'fred' to %d\n", GET_LOCAL_INTEGER("fred"));
}
```

However, it's inefficient in that the scope stack is being repeatedly scanned for the same variable. It is more efficient to write it with calls to `FIND_LOCAL` as follows, to return a `SBLOCAL` that can then be used for indirect, and much more efficient, access.

```
#include <stdio.h>
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = LOCAL_INTEGER("fred");
    setSBLocalVariable_i(temp, 666);


    ...
    proc_2();
    ...
    printf("I see 'fred' has been altered to %d\n", getSBLocalvariable_i(temp));

}

void  proc_2() {
    SBLOCAL tempFred = FIND_LOCAL("fred");
    ...

    printf("I see 'fred' is set to %d\n", getSBLocalvariable_i(temp));
    setSBLocalvariable_i(temp, 616);
    printf("I altered 'fred' to %d\n", getSBLocalvariable_i(temp));
}
```

By using a pointer (`SBLOCAL`) in this manner, you only need one single scan of the scope stack to find the location of the variable `fred`.

Readability or Efficiency?

As mentioned above, there are efficiency considerations when accessing `LOCal` variables. The scope stack must be searched each time you attempt indirect access to a variable, but when using direct access, the code is less readable. What to do?

Simple, do what suits you best. There is nothing wrong with the following code at all, it's just going to take a wee bit longer to execute - but will you actually notice?

```
#include <stdio.h>
#include "SBLocal.h"

...

void proc_1() {

    SBLOCAL temp = LOCAL_INTEGER("fred");
    SET_LOCAL_INTEGER("fred", 666);
    ...
    proc_2();
    ...
    printf("I see 'fred' has been altered to %d\n", GET_LOCAL_INTEGER("fred");

}

void  proc_2() {
    printf("I see 'fred' is set to %d\n", GET_LOCAL_INTEGER("fred"));
    SET_LOCAL_INTEGER("fred", 616);
    printf("I altered 'fred' to %d\n", GET_LOCAL_INTEGER("fred"));
}
```

The choice is yours.

SuperBASIC Arrays

When you declare an array of integers or floating point variables, you get *one extra* element. For example, this is perfectly valid code in SuperBASIC:

```
1000 DIM a%(5)
1010 FOR x = 0 to 5
1020   a%(x) = x
1030   PRINT 'a%('; x; ') = '; a%(x)
1040 END FOR x
```

The output will be:

```
a%(0) = 0
a%(1) = 1
a%(2) = 2
a%(3) = 3
a%(4) = 4
```

```
a%(5) = 5
```

With strings, however, you have to assign the whole thing, or start indexing at 1, not zero. Element 0 is usually the size of the string.

```
1000 DIM b$(10)
1010 B$ = "Hello"
1020 PRINT "B$(0) = "; CODE(B$(0)), b$
```

Would result in:

```
B$(0) = 5     Hello
```

For these reasons, arrays and strings are dimensioned with an extra element and in the case of strings, a further extra element to hold the C68 string terminator.

SBLocal Defines

The following have been set up to enable `LOCal` variables of different types to be declared within a converted application.

- SBLOCAL_UNDEFINED 0 - Used when a `SBLocalVariable` is initially allocated on the heap. The variable has no actual type at this point and is set to zero.
- SBLOCAL_INTEGER 1 - Used to create or access SuperBASIC integer equivalents, whcih are 16 bit signed variables, or `short` in C68.
- SBLOCAL_FLOAT 2 - Used when a variable is converted from a SuperBASIC flaoting point. These are set up as `double` in C68.
- SBLOCAL_STRING 3 - Used to create a simple SuperBASIC string variable. This may or may not be `DIM`ensioned but if not, will default to a particular size as defined by `SB_DEFAULT_STRING` as described below.
- SBLOCAL_INTEGER_ARRAY 4 - Used to create `LOCal` integer arrays. Any number of dimensions are permitted.
- SBLOCAL_FLOAT_ARRAY 5 - Used to create `LOCal` floating point arrays. Any number of dimensions are permitted.
- SBLOCAL_STRING_ARRAY 6 - Used to create `LOCal` string arrays. Any number of dimensions are permitted.

The following describe various internal limits on the converted application.

- MAX_STACK_DEPTH - defines the depth of the scope stack used in the converted application. Horribly recursive applications may need to increase this from the default setting of 2048 if you find stack overflow messages appearing.
- MAX_LOCAL_NAME_SIZE - limits the size of a `LOCal` variables name to 31 characters. Any variable with a name longer than this will be truncated to fit. Why 31? I'm an Oracle DBA by profession, and that's considered big enough for table or column names!
- SB_DEFAULT_STRING_LENGTH - limits the default dimension, in the converted application, of any un`DIM`ensioned `LOCal` strings 100. So `LOCal a$` will become a 100 character string.

SBLocal Functions