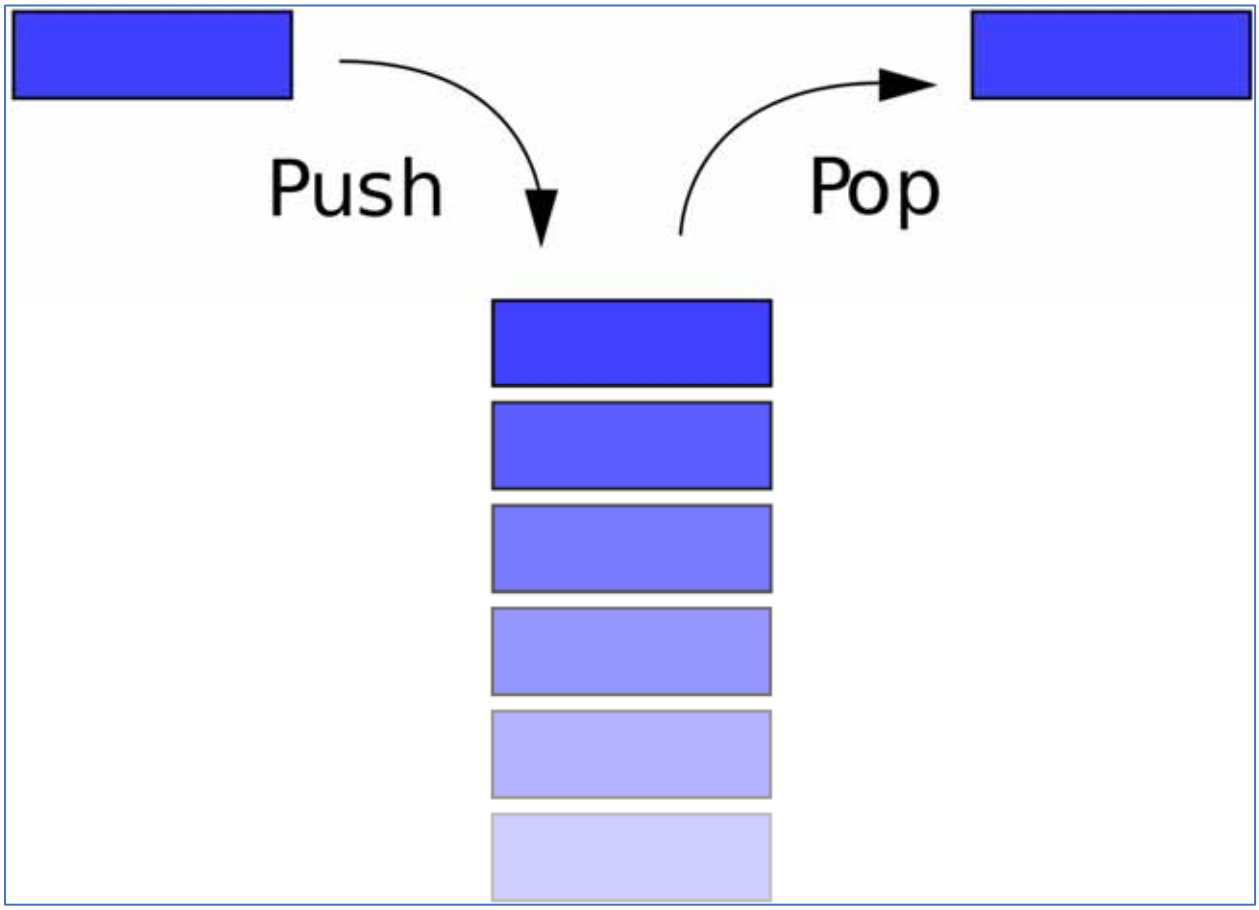
How to implement a simple stack data structure in TBASIC.

Overview

A stack is a basic data structure that shows up in all sorts of programming languages and is used to manage a collection of data items. Stacks are used to pass the arguments to functions. Stacks are used to manage return addresses for subroutines.

A physical implementation of a stack is found at the finer buffets is the form of a plate dispenser The dispenser holds a pile of plates and when you take a plate off of the top of the stack springs push up the rest of the stack so that you that you the next plate can be accessed. The action of removing a plate is a “pop” operation. The plate dispenser is loaded by pushing new clean plates onto the stack.

Here’s a lovely graphic that I stole from the internet that is a good model of a stack:

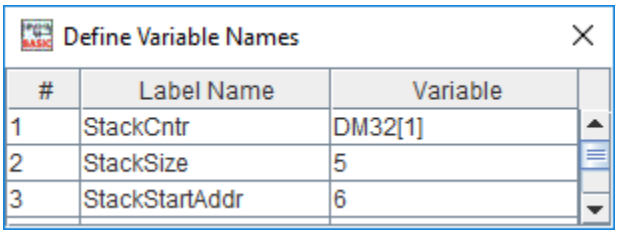


Stack behavior

The last thing that was pushed onto a stack will be the first thing that is popped off the stack. This gives rise to the term, LIFO, for “last in, first out”. One usage of stacks is to reverse the order that data is processed. As an example, most algorithms to convert a binary value to a series of ASCII characters work from left to right but the characters need to be printed from left ot right. The algorithms to print a binary representation of 12345 would first come up with the “5”, then the “4”, “3”, 2 and finally “1”. A stack makes a great data structure to store the digits as they are calculated and then pop them off the stack when it was time to print it and you’d get something like “12345”. The stack provides both the storage space and as a bonus reverses the order of the data.

Stack.PC6 Custom Functions

The TBASIC program, “Stack.PC6”, is very simple stack that is based on the DM32[] data that the TRI PLCs support. The program takes advantage of #Define mechanism of TBASIC to determine the dimensions and placement of the stack structure. The is the #Define table:



- **StackCntr** – is a 32-bit variable that is used to keep track of the number of items that the stack is holding, right now.
- **StackSize** is the maximum number of items that the stack can hold. If you need a bigger or smaller stack, just change the number. For this test code, 5 items were enough to test and debug the code.
- **StackStartAddr** this is the offset into DM32 for the first item in the stack. DM32[StackStartAddr] will hold the first item that is pushed onto the stack. DM32[StackStartAddr + StackSize] will hold the last item that the stack can hold. Change this define to move the stack structure start within the DM32[] array that your PLC supports.

Implementation Details.

Three TBASIC custom functions are used to manage the stack operation:

- **InitStack** – Is called to initialize the variable, StackCntr, and tracking RELAYS StackEmpty and StackFull
- **PushStack** – is used to push the value of the variable, A, onto the stack. If the stack is full, “no room at the in”, then the code does nothing. The tracking RELAY, StackFull will be set when the stack is full.
- **PopStack** is used to pop the newest element off the stack and copy it into the variable, B. If the stack is empty, then nothing happens. The tracking RELAY StackEmpty will be set when the stack is empty.

Load up the Stack.PC6 and look at the 3 custom functions. It is possible that I am just making this stuff up.

InitStack

This custom function must be called before the Stack mechanism can be used. The initialization of the stack is very simple, one 32-bit variable and two RELAYs:

```
' InitMsgFifo - Initialize FIFO tracking variables in DM and RELAY based state flags
'
StackCntr = 0       ' This value is the number of items written into the
                    ' stack array.
                    '
                    ' The value of StackCntr is between 0..StackSize.
                    '
                    ' StackCntr is incremented after data is pushed
                    ' StackCntr is decremented after data is popped

SetIO StackEmpty    ' RELAY, when set, indicates that the stack contains no data.
                    ' When set, the stack pop operation will not be allowed.

ClrIO StackFull     ' RELAY, when set, indicates that the stack is full.
                    ' When set, the stack push operation will not be allowed.
```

PushStack

This custom function copies the value of the variable, A, on to the stack. This is the code for PushStack:

```
' PushStack - function to push data element onto Stack
'
' On entry A holds value to push onto the stack
'
'  On exit: The value of A will be pushed onto the stack data structure.
'           If the stack is full then nothing is pushed onto the stack
'
'           The values of the StackCntr and the stack full/empty RELAYS may
'               be affected.

if (Testio(StackFull) = 0)      ' Verify that the stack can accept a new value

    ' The stack is not full...
    '
    '   Copy the value of the variable A into the first available location in the
    '   stack.
    '
    DM32[StackCntr + StackStartAddr] = A

    ' Update stack counter and flags
    '
    ' StackCntr is incremented after the data is written into the stack.
    '
    StackCntr = StackCntr + 1
    if (StackCntr = StackSize)

        ' Just set a RELAY to indicate that the stack
        '   cannot accept more data.

        SetIO StackFull
    endif

    ClrIO StackEmpty    ' stack cannot be empty.
endif
```

PopStack

This custom function copies the “newest” value on the stack and then assigns this value to the variable, B:

```
' PopStack - function to remove the newest value on the stack and copy this value
'               into the variable B.  This is a pop operation.
'
'  On exit: If the stack is not empty, then the newest stack entry will be removed
'               from the stack and this value will be assigned to B.
'
'           If the stack is empty then nothing happens. B is not changed.
'
'           The values of the StackCntr and the stack full/empty RELAYS may
'               be affected.
'
if (TestIO(StackEmpty) = 0)     ' Verify that the stack is not empty
    ' the Stack was not empty, so return value of most recent entry
    '
    ' StackCntr is decremented to "point" at the newest item in the stack.
    ' This removes this value from the stack and it's value will be copied
    ' to the variable, B.
    '
    StackCntr = StackCntr - 1
    B = DM32[StackStartAddr + StackCntr]
    ' Update flags
    '
    if StackCntr = 0
        SetIO StackEmpty        ' stack is now empty
    endif
    ClrIO StackFull             ' Stack can't be full
endif
```

Stack Behavior

The following diagrams illustrate the stack behavior. Remember that the StackCntr is a variable that tracks the number of items on the stack. The RELAYs StackEmpty and StackFull provide a simple mechanism to monitor the stack state.

Stack state after call to StackInit CF

Stack after Initialization

Status Info		DM32	Value	
StackCntr	0	DM32[06]	??	← Next Value
StackEmpty	1	DM32[07]	??	
StackFull	0	DM32[08]	??	
		DM32[09]	??	
		DM32[10]	??	

Note that the StackCntr variable is 0 and that the StackEmpty RELAY is SET. NextValue points at the location in DM32[] variable that will be used to store the next value that is pushed onto the stack via the PushStack custom function. Don't go looking for NextValue as a variable in the program. NextValue is only needed 2 times in the entire code and it took less time to just calculate the value as needed. The two uses can be found in the PushStack custom function as:

```
DM32[StackCntr + StackStartAddr] = A
```

And in the PopStack custom function as:

```
B = DM32[StackStartAddr + StackCntr]
```

Stack response to StackPush CF

OK now let's push the value 1111 onto the stack. This is what happens:

After Push "1111"

Status Info		DM32	Value	
StackCntr	1	DM32[06]	1111	
StackEmpty	0	DM32[07]	??	← Next Value
StackFull	0	DM32[08]	??	
		DM32[09]	??	
		DM32[10]	??	

Note that the StackCntr is now 1. The value 1111 is stored in DM32[06]. DM32[[07] will hold the next value that is pushed. And finally, note that the StackEmpty RELAY is not SET as the stack is no longer empty.

What happens when the stack fills to capacity?

I will push the values 2222, 3333, 4444, 5555 and 6666 onto the stack. This is what you should see after all of this gets done:

After Push "6666"

Status Info		DM32	Value
StackCntr	5	DM32[06]	1111
StackEmpty	0	DM32[07]	2222
StackFull	1	DM32[08]	3333
		DM32[09]	4444
		DM32[10]	5555

← Next Value

Note that the StackCntr has been set to 5 as there are 5 items stored on the stack. Note, also, that the StackFull RELAY is set to indicate that the stack is full. The 6th value of 6666 was not pushed onto the stack because the stack was filled after the 5th value, 5555, was pushed.

Note that the NextValue is pointing beyond the end of the DM32[] storage area. This is not an error. The stack will not allow any more data to be pushed onto a stack that is full. The only stack operation that can be performed is a pop operation. Pop operations always start with backing up the NextValue.

Pop Operations

OK. Now let's pop the newest item off the stack with a call to PopStack custom function. This is what you should see after the pop operation:

After Pop "5555"

Status Info		DM32	Value
StackCntr	4	DM32[06]	1111
StackEmpty	0	DM32[07]	2222
StackFull	0	DM32[08]	3333
		DM32[09]	4444
		DM32[10]	5555

← Next Value

If you examine the variable, B, it will hold 5555 as a result of the pop. There is no reason to overwrite the 5555 value that is stored in DM32[10] as this serves no useful function. The StackCntr is now 4 indicating that the stack holds 4 values. And the StackFull tracking RELAY is not set as the stack is not full.

What happens when we pop the rest of the data off of the stack

If we keep popping things off the stack the the values 4444, 3333, 2222 and 1111 will be recovered and the final picture will look like this:

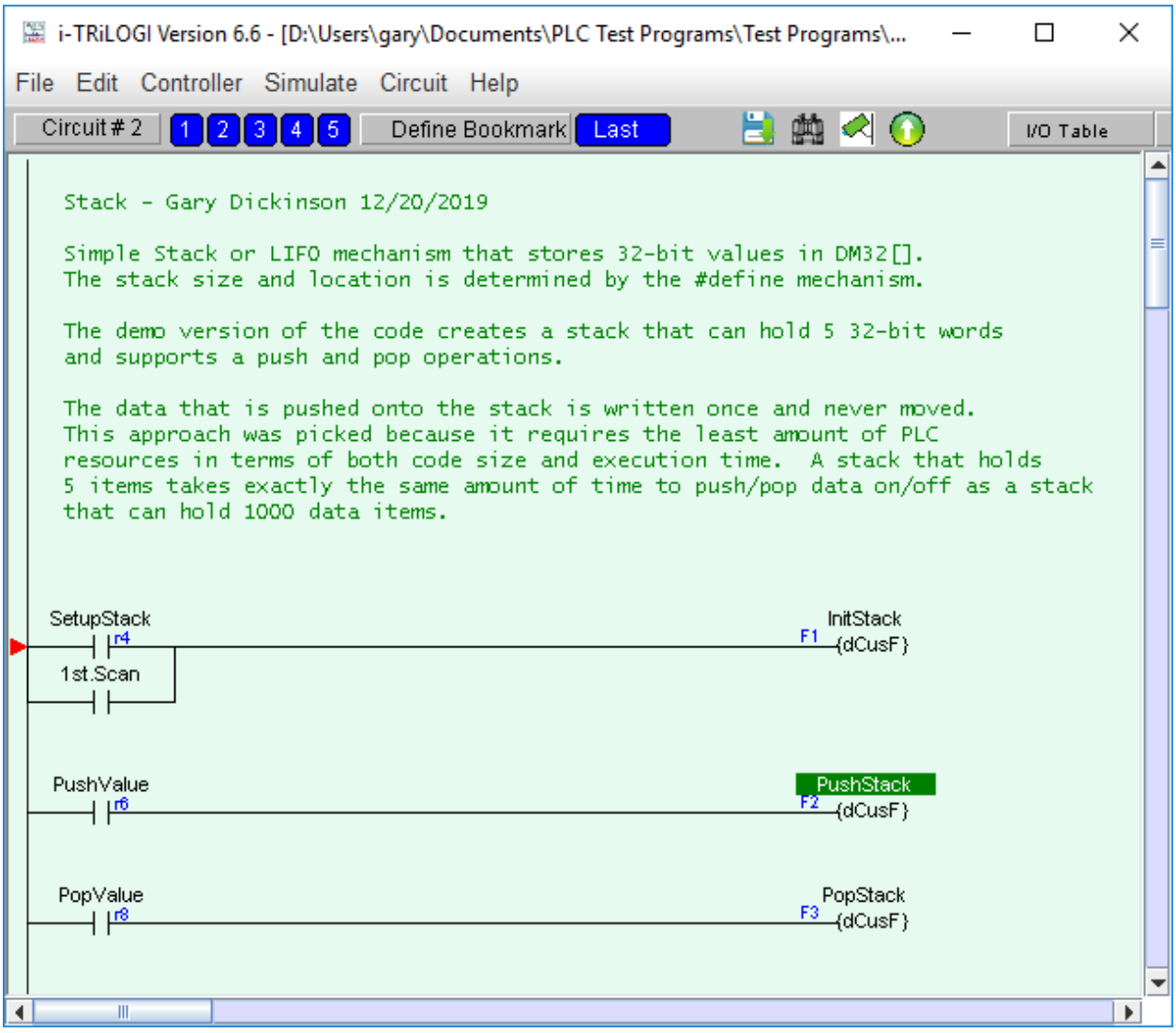
After Pop "1111"

Status Info		DM32	Value
StackCntr	0	DM32[06]	1111
StackEmpty	1	DM32[07]	2222
StackFull	0	DM32[08]	3333
		DM32[09]	4444
		DM32[10]	5555

← Next Value

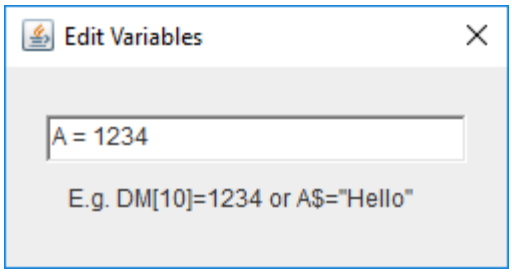
The stack is empty. The StackEmpty RELAY will be set and the StackCntr will be 0. If you attempt to pop a value off an empty stack nothing happens. The PopStack code will not change the StackCntr value and will not assign anything to the variable, B.

OK. Now let's glance at the ladder logic for the test bed:



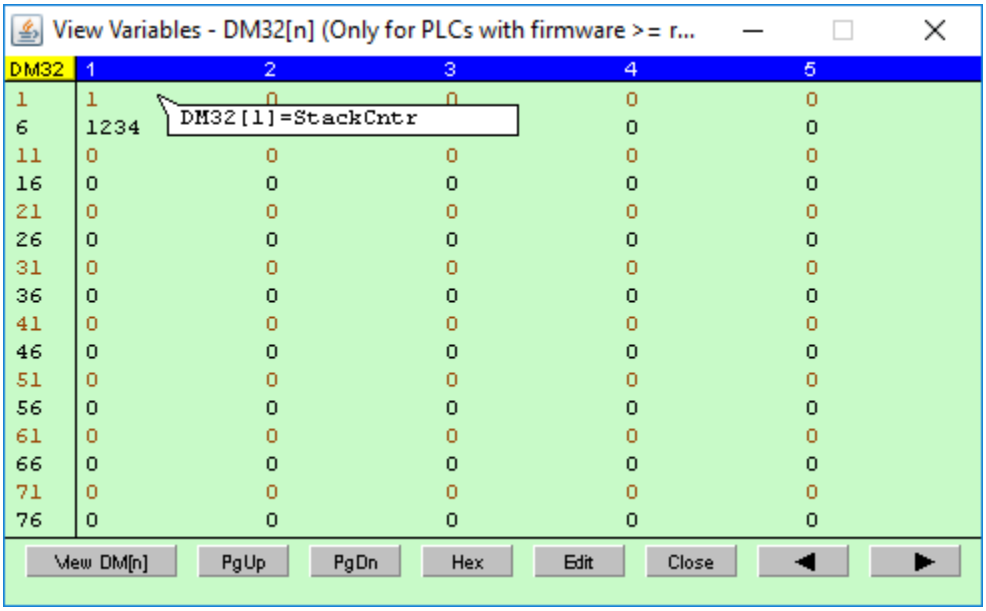
The test bed code uses 3 RELAYS, SetupStack, PushValue and PopValue to allow one to invoke the 3 custom function that manage the stack structure.

Run the code in the i-TRiLOGi simulator. Click on the "View" menu button to open the ViewVariable Integer window. This is where you can "see" the A and B variables. You can edit these variables by clicking on the "Edit" menu button to open the "Edit Variables" window. Enter something like this and press "Enter":



Now you have set variable A to 1234. If you click on the Push Value RELAY in the main simulator window you will push 1234 onto the stack.

Have a peek at the DM32[] data to see what happens. Press on the ">?" menu button to get to the "View Variables – DM[n]" and then click on the "View DM32[n]" menu button to see the DM memory as an array of 32-bit things. This is what you should see:



Look at DM32[1] this is the StackCntr. If you click on the "1" at DM32[1] the simulator will pop up a little bubble "DM32[1] = StackCntr" to remind you that this variable has a useful name. The "1" indicates that the stack now holds 1 value. Consider this a subtle hint to make use of the #define mechanism to help make debugging easier.

Now look at DM32[6] and it should hold the value of A, 1234. If you click on the value at DM32[6] you will get a bubble that says "DM32[6] (no name) as a reminded that you didn't assign a name to this location in the #define table.

Gary Dickinson 1/6/2020

garysdickinson@gmail.com