1 The RPL Loop

This is a mini-tutorial on how to create your own prologue using a library. To start, let's take a look at the RPL loop. We will start with a simple program, and its hexadecimal representation. (If you haven't already, switch to the a fixed size font for improved legibility).

```
System RPL Program              Hexadecimal representation

::                              02D9D
  "Hello"             -->           02A2C 0000F 48 45 4C 4C 4F
  NEWLINE$&$                        361DA
;                               0312B
```

From the assembly language point of view, how is this program run? To the system, everything is considered to be a pointer, i.e. an address to some code block somewhere in memory—be it ROM or RAM or even flash ROM. All addresses are 5 nibbles in length. The RPL loop is coded as:

```
ASSEMBLE
        A=DAT0  A
        D0=D0+  5
        PC=(A)
RPL
```

During program execution, `D0` generally acts as the pointer into the run-stream, which in this case is the program above. Of course our program must be stored somewhere in memory. So `D0` initially set to the memory location containing the value `02D9D` (this is the prologue corresponding to the `::` in our source code). It reads the content of that memory location and stores it into `A[A]`. Then it increments `D0` to point to the next memory location, which contains the value `02A2C` (this is the prologue of our `"Hello"` string). Then the CPU executes the code contained within the address located at `A[A]` using the opcode `PC=(A)`. So when `A[A] = 02D9D`, the program counter would be set to whatever address is contained at the address `02D9D`. Looking at the memory address `02D9D` (this is the address for `DOCOL`), we see

```
ASSEMBLE
=DOCOL
        CON(5)  #028FC          =PRLG
RPL
```

So the program counter is set to `028FC`. Now when we look at `028FC`, we see the following code:

```
ASSEMBLE
=PRLG
```

```
        LC(2)   10              #0Ah
        A=A-C   B               #02D9Dh - #0Ah = #02D93
        PC=(A)
RPL
```

Remember that at this point, `A[A] = 02D9D`. The code residing at `028FC` (`PRLG`) says to resume execution at the address contained at `02D93`. Peeking into `02D93`, we see

```
ASSEMBLE
L02D93
        CON(5)  #02DE0
RPL
```

So now the program counter gets set to `02DE0` and at this memory location is the following code:

```
ASSEMBLE
        D0=D0-  5               backtrack to previous
                                pointer in runstream
        CD0EX
        D0=C
        RSTK=C                  save current D0
        GOSUB   SKIPOB          after SKIPOB, D0 is now
                                points to the end of program
        C=RSTK
        A=C     A               A[A] contains the address of
                                the start of our program
        D=D-1   A               decrease the num of available
                                ptrs (for our new rtn addr)
        GOC     L02E15          do garbage if not enough room
        C=B     A               B[A] contains the addr of
                                where we store the rtn addr
        CD0EX
        DAT0=C  A               address of the end of our
                                program saved as a rtn addr
        D0=D0+  5               advance the rtn stack for
                                future writes
        AD0EX                   D0 points back to current
                                runstream
        B=A     A               rtn stack updated w/ the new
                                rtn addr (end of our prog)
        D0=D0+  5               we backtracked; so now advance
                                to next pointer in runstream
        A=DAT0  A               and loop
        D0=D0+  5
        PC=(A)
RPL
```

At this point, D0 now points to the nibbles 02A2C. This is the string prologue, and it has its own, similar set of machine code to handle the execution of a string object. Any adjustments to the runstream pointer D0 is done by the code located at 02A2C.

Let's skip forward to the entry NEWLINE$&$ (address 361DA). If we apply the RPL loop to this address, then the program counter gets set to the address contained at 361DA. Here's what the ROM looks like:

```
ASSEMBLE
=NEWLINE$&$                 source code             hexadecimal
        CON(5)  =DOCOL      ::                      02D9D
        CON(5)  =NEWLINE$     NEWLINE$&$                33B39
        CON(5)  =&$           &$                        05193
        CON(5)  =SEMI       ;                       0312B
RPL
```

So now when the CPU executes PC=(A), the program counter is set to the address 02D9D. In the explanation above, when A[A] = 02D9D, the program counter was set to 028FC because the address 02D9D contained the nibbles 028FC. This time, however, we have A[A] = 361DA (the value of entry NEWLINE$&$), and the program counter is set to 02D9D (the nibbles contained at the address 361DA). We saw earlier, however, that the ROM contained the nibbles 028FC at the address 02D9D. While on the one hand 028FC is the address for the entry PRLG, it also translates to the opcodes:

```
        D=D-1   A               opcode CF
        HS=0    0               opcode 820
```

With this in mind, let's look at address 02D9D (the address of DOCOL) one more time, except with the nibbles 028FC translated to opcodes:

```
ASSEMBLE
=DOCOL
        D=D-1   A               decrease num of ptrs available
        HS=0    0               equivalent to a NOP
        GOC     L02E15          if CRY, we need to do a garbage
                                collection and resume execution
        C=B     A
        CD0EX                   D0 -> rtn stack; C[A] = address
                                of next runstream obj
        DAT0=C  A
        D0=D0+  5
        AD0EX                   in our example, A[A] = 361DA;
                                D0 -> NEWLINE$ within NEWLINE$&
$
        B=A     A               B[A] = new rtn stack addr
        D0=D0+  5
```

```
        A=DAT0  A
        D0=D0+  5
        PC=(A)
RPL
```

So now that we have a (vague?) understanding of how System RPL and machine language interact, we can then look at the ROM code near the address `02D9D` (the `DOCOL` prologue) to see how we might build our own prologue. Here's what the ROM looks like:

```
ADDRESS NIBBLES MNEMONIC

02D93   0ED20   CON(5)  =spancol   direct executor for docol
02D98   77030   CON(5)  =skipcol   code to skip docol objs
02D9D   CF820   CON(5)  =PRLG      indirect execution for docol
...     ...     GOC     L02E15
...     ...     C=B     A
...     ...     CD0EX
...     ...     DAT0=C  A
...     ...     D0=D0+  5
...     ...     AD0EX
...     ...     B=A     A
...     ...     D0=D0+  5
...     ...     A=DAT0  A
...     ...     D0=D0+  5
...     ...     PC=(A)
```

Rather than placing all the code for `DOCOL` immediately after `CON(5) =PRLG`, we could simply have a `GOVLNG =execcol` call and all the machine code to execute the `DOCOL` object can be stored at the label `=execcol`. As you can see, we really only need a total of $3 * 5 + 7 = 22$ nibbles somewhere safe in RAM where we can inject our own code:

```
ASSEMBLE
        CON(5)  spanaddr        5 nibbles
        CON(5)  skipaddr        5 nibbles
        CON(5)  =PRLG           5 nibbles
        GOVNLG  execprolog      7 nibbles
RPL
```

When we inject this snippet of machine code into RAM, the labels spanaddr, skipaddr, and execprolog must exist somewhere else that can easily be accessed by operating system. The best place is in a library stored in Port 0. We could theoretically store it in covered memory and, instead of the `GOVLNG` call, add code to uncover/cover memory banks. However, that requires more overhead AND objects in covered memory are not stable. In fact, each time an object is stored into a port, the system rescans the objects stored there. During the scan,

it is possible that the system moves objects around within a port to allocate memory prior to storing. This is done to minimize the number of writes to the flash chip (for Port 2) and hence maximize the lifespan of the flash memory. Lastly, Port 0 is uncovered SRAM, which is stable (we do not want spanaddr, skipaddr, and execprolog to point to correct memory locations upon installation and then point to invalid memory locations after being moved around).

The implementation I came up with uses a library and the config object of the library to insert code into RAM. The library itself contains the main code for handling our new prologue. To insert our prologue into memory, I chose a place that is seemingly never used after being set up: `TolVar`s and `TopicVar`s land! The config program of the library modifies `FIRSTPROC` to include the code object which will inject our custom prologue into reserved RAM blocks.

Anyway, that's enough writing for now. Feel free to send questions to `hduong{\textunderscore}nospam@ju.edu` (remove the _nospam). Feel free to use the code however you wish; I just ask that you give me credit for whatever part of my work you use.