Self Documenting Sequences

N.J. Nelson, K.B. Swiatlowski

Abstract

When creating automation code for mechanical handling equipment which is specially adapted for installation at a wide variety of different sites, it is common for numerous alterations in the code to be required on site, at the last minute. Under pressure, user documentation starts to diverge from code. How nice it would be, if clear, accurate and readable documentation could be regenerated automatically each time the code was recompiled - and translated, also automatically, into the customer's language!

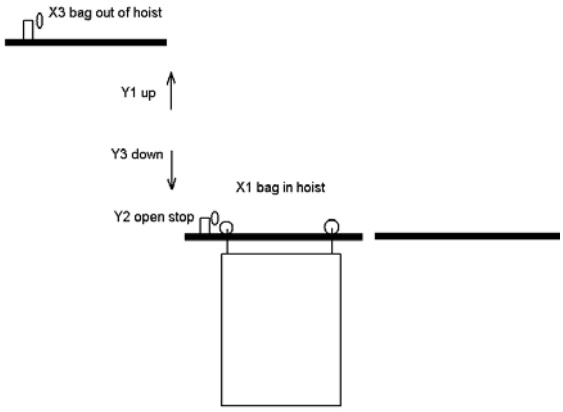
N.J. Nelson B.Sc., C.Eng., M.I.E.E.
K.B. Swiatlowski Mgr.
Micross Electronics Ltd.,
Units 4-5, Great Western Court,
Ross-on-Wye, Herefordshire.
HR9 7XP U.K.
Tel. +44 1989 768080
Fax. +44 1989 768163
Email. njn@micross.co.uk

1. Background - sequences

At the 1997 EuroForth conference, Jonathan Morrish won a prize for his paper, "*Rapid development of real time multi-sequence control programmes* ".
Jonathan is no longer with Micross, but the wordset he described was so successful that we're still using them today.

In a typical complex conveyor system, the overall control process is broken down into a set of "sequences", each of which controls the movement of an item from one position to another. The sequences is divided into a series of "steps" thus forming a kind of state machine.

A simple example Jonathan used looked like:



A container runs by gravity into a vertical conveyor, is elevated, and runs off to a storage position.

The sequence would be described in the documentation as follows:

Sequence 1 - Hoist

Step	Stepped on by	Outputs
	Bag runs into hoist	
0	Wait for container to arrive in hoist (X1) Hoist at bottom (X4)	None
	Bag settles in hoist	
1	Wait for 3s for container to settle	None
	Hoist goes up	
2	Hoist at top (X2)	Hoist UP (Y1)
	Bag runs out of hoist	
3	Container out of hoist (X3)	Open stop (Y2)
	Hoist goes down	
4	Hoist at bottom (X4)	Hoist DOWN (Y3)

Since 75% of our equipment is exported, the documentation is generally provided in at least two languages.
In Jonathan's wordset, the sequence is coded as follows:

```
: HOIST
  1 SS CASE                      \ SS - set up specified sequence
    0 OF 1 X 4 X AND ?NS ENDOF \ X - true if input on
    1 OF 3 SECS ?NS      ENDOF \ SECS - true if time
    2 OF 2 X ?NS         ENDOF \ ?NS - next step if true
    3 OF 3 X 20DNS       ENDOF \ 20DNS - set diagnostic after 20s
    4 OF 4 X ?0S         ENDOF \ 0S - reset sequence
  ENDCASE
  S@                             \ S@ - returns step number
    DUP     2 =   1 Y            \ Y - output on/off
    DUP     3 =   2 Y
            4 =   3 Y
;
```

Note how compact the code is.

2. The problem

a) All conveyor systems are different, so sequences need to be individually documented and coded for each installation.

b) Complete conveyor systems are very large and complex and are only completely assembled for the first time at the end customer's site.

c) At the time of installation it is always found that there differences between the original specification and the customer's actual requirements.

d) Code modifications are therefore made on the spot.

e) Under the intense pressure of commissioning a large system, the code begins to diverge from the documentation.

3. The Eureka Moment

This occurred during the third glass of wine after the second day of EuroForth 2004.

The documentation essentially contains the same information as the code itself. The only difference is that the documentation refers to names of sequences, steps and signals, as well as numbers. However, the documentation already contains a list of signals, therefore sequence and step names is the only additional data in the documentation.

Although each sequence is different, the same phrases e.g. "Hoist goes up" are frequently repeated from job to job. They already exist in French and German. Hence automatic translation is usually possible.

Therefore, the code can be used to generate, and even translate the code, provided only two new words are added to the wordset:

```
n SQ" Sequence name"     \ Document sequence name, then do SS CASE
n ST" Step name"         \ Document step name, then do OF
```

4. The specification

As we developed the specification, we realised that further advantages could be gained. Since the code had generated the documentation, the program "knew" about the documentation structure and thus could display the information dynamically, potentially making the debugging of sequences much easier. For example, the current step could be described, and each logical action could indicate its true or false state. To find a faulty signal, it was merely necessary to notice which one was coloured false. The visualisation therefore became an important feature of the new sequence system.

However, the overriding requirement was that, with the exception of the two words mentioned above, the code should be exactly the same as before - just as simple and compact.

5. The implementation

Data about a sequence is generated during compilation. The function assigned to the word differs for different stages of the compilation, so every word that needs to generate documentation has to be be declared as DEFER-ed.
Example:

```
DEFER X ( x---f )
DEFER ?NS ( f ---)

: ~?NS ( f--)           \ "~" sign added to the word's name
  IF 0T NS THEN ;

: ~X ( x---f )          \ True if input is ON
  INPUTIMAGE + C@  ;

: IMM-X
  GET-FROM-DP DROP      \ Gets compiled input number from dictionary
  COMPILE ~X ;

: IMM-?NS
  GET-FROM-DP DROP      \ Puts "next step" entry into data structure
  COMPILE ~?NS ;
```

Every new word which needs to generate a description is registered (added) in function GET-SQD-WORD, to permit automatic assignment of multiple words during invoking words: START-DESCRIPTION and END-DESCRIPTION. START-DESCRIPTION sets IMMEDIATE flag of deferred word, END-DESCRIPTION resets that flag and assigns controlling words instead of compiling words.

Word GET-FROM-DP takes the last entry from the dictionary and the function "name" that called it and passes it to the word SQD-VOCAB, which fills the step data structure to preserve parameters.

```
STRUCT STEP-ELEM            \ Data structure for one step instruction
  CELL FIELD .DP_NUMBER     \ Any number associated with data?
  CELL FIELD .PHRASE-FUN    \ Points a function generating description string
  CELL FIELD .TYPE          \ Extra parameter to distinguish ie input from output
  CELL FIELD .PHRASE-ADD    \ Anything to stick on the end, holds a sqphrase number?
  CELL FIELD .TICKNUM       \ Store the number to recall the name (and CFA)
  CELL FIELD .TICKTYPE      \ Store the type of the word – number of parameters.
END-STRUCT
```

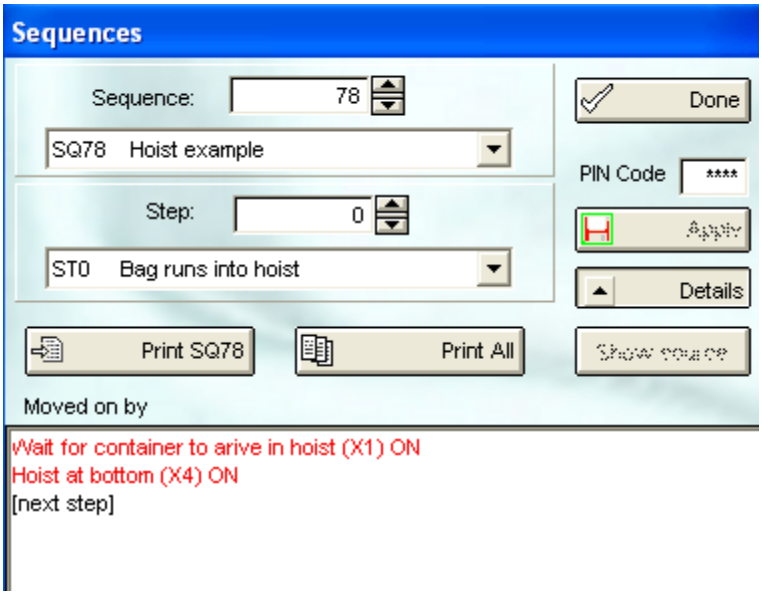
The list of instructions for each step of each sequence is generated, which index gives access to the instruction's parameters, description and FA.
A useful word SOURCE>INT returns an integer that comes after a word in the source stream. So having a piece of code like this:

```
3 PARAM@ 5 =
```

we can get both numbers 3 from the dictionary and 5 from the source. -1 is returned if a word is next instead of a number.

6. The result

```
START-DESCRIPTION
: SQ78
  78 SQ" Hoist example"
    0 ST" Bag runs into hoist"
          1 X 4 X AND ?NS          \ IO names taken from IO data base
    ENDOF
    1 ST" Bag settles in hoist"
      3 SECS ?NS
    ENDOF
    2 ST" Hoist goes up"
      2 X ?NS
    ENDOF
    3 ST" Bag runs out of hoist"
      3 X 20DNS
    ENDOF
    4 ST" Hoist goes down"
      4 X ?OS
    ENDOF
;
END-DESCRIPTION
```



7. Conclusion

The new system has already been used in three installations and has greatly decreased the time taken to write and debug sequences. It has also proved very popular with the plant maintenance technicians, who can identify faulty mechanical parts, sensors or actuators more quickly.