This is a simple illustration of the machinery which is used for more complex purposes within TinyMUX. Converting an ASCII digit to a value is simple enough that there are many other, better ways to do it. But, for the sake of illustration, let's say that we want to map U+0030 through U+0039 to their corresponding decimal value using the TinyMUX machinery for UTF-8. First, we need the mapping, so we create a tr_digits.txt file with the following:

```
0030;0;DIGIT ZERO;
0031;1;DIGIT ONE;
0032;2;DIGIT TWO;
0033;3;DIGIT THREE;
0034;4;DIGIT FOUR;
0035;5;DIGIT FIVE;
0036;6;DIGIT SIX;
0037;7;DIGIT SEVEN;
0038;8;DIGIT EIGHT;
0039;9;DIGIT NINE;
```

The first field of each line is the Unicode code point in the usual hexadecimal format. The second field is the integer value, the third and fourth fields are optional. Then, we use the following command to build the tables for a state machine that will recognize and perform this mapping:

```
./integers -d 10 tr_digits tr_digits.txt
```

This produces the following tables:

```
// utf/tr_digits.txt
//
// 10 code points.
// 1 states, 11 columns, 269 bytes
//
#define TR_DIGITS_START_STATE (0)
#define TR_DIGITS_ACCEPTING_STATES_START (1)
extern const unsigned char tr_digits_itt[256];
extern const unsigned char tr_digits_sot[1];
extern const unsigned char tr_digits_sbt[12];
```

```
// utf/tr_digits.txt
//
// 10 code points.
// 1 states, 11 columns, 269 bytes
//
const unsigned char tr_digits_itt[256] =
{
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       1,   2,   3,   4,   5,   6,   7,   8,    9,  10,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,

       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0

};
```

```
const unsigned char tr_digits_sot[1] =
{
        0
};

const unsigned char tr_digits_sbt[12] =
{
     245,  11,   1,   2,   3,   4,   5,   6,    7,   8,   9,  10
};
```

The input translation table maps input bytes to column numbers:

Otherwise	**0**
‘0’ (0x30)	**1**
‘1’ (0x31)	**2**
‘2’ (0x32)	**3**
‘3’ (0x33)	**4**
‘4’ (0x34)	**5**
‘5’ (0x35)	**6**
‘6’ (0x36)	**7**
‘7’ (0x37)	**8**
‘8’ (0x38)	**9**
‘9’ (0x39)	**10**

The state transition table contains one state and eleven columns:

	Column										
Previous State	0	1	2	3	4	5	6	7	8	9	10
State 0	**11**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Any state beyond State 0 ends the machine and gives the mapping. The mapping is determined by looking at how far the next state is beyond the end of the state transition table. For example, State 1 implies `(1-TR_DIGITS_ACCEPTING_STATES_START)` or a value of zero. That is, ‘0’ maps to a value of zero – which is what we wanted. Likewise, State 10 implies `(10-TR_DIGITS_ACCEPTING_STATES_START)` or a value of nine. Finally, State 11 implies a value of ten which implies that it did not match anything in the list (controlled by the `-d 10` command-line option).

It’s not hard to write a helper function based upon these tables:

```
int DigitValue(const UTF8 *pDigit)
{
    int iState = TR_ASCII_START_STATE;
    do
    {
        unsigned char ch = *p++;
        unsigned char iColumn = tr_digits_itt[(unsigned char)ch];
        unsigned char iOffset = tr_digits_sot[iState];
        for (;;)
        {
            int y = tr_digits_sbt[iOffset];
```

```
            if (y < 128)
            {
                // RUN phrase.
                //
                if (iColumn < y)
                {
                    iState = tr_digits_sbt[iOffset+1];
                    break;
                }
                else
                {
                    iColumn = static_cast<unsigned char>(iColumn - y);
                    iOffset += 2;
                }
            }
            else
            {
                // COPY phrase.
                //
                y = 256-y;
                if (iColumn < y)
                {
                    iState = tr_digits_sbt[iOffset+iColumn+1];
                    break;
                }
                else
                {
                    iColumn = static_cast<unsigned char>(iColumn - y);
                    iOffset = static_cast<unsigned char>(iOffset + y + 1);
                }
            }
        }
    } while (iState < TR_ASCII_ACCEPTING_STATES_START);
    return iState – TR_DIGITS_ACCEPTING_STATES_START;
}
```

With these pieces in place, adding other Unicode characters only requires adding a line to the tr_digits.txt file. No other changes are necessary. Let's try it. In Unicode, there are 370 non-ornamental digits. Due to space constraints, let's just add the next range (ARABIC-INDIC DIGIT ZERO through NINE).

```
0030;0;DIGIT ZERO;
0031;1;DIGIT ONE;
0032;2;DIGIT TWO;
0033;3;DIGIT THREE;
0034;4;DIGIT FOUR;
0035;5;DIGIT FIVE;
0036;6;DIGIT SIX;
0037;7;DIGIT SEVEN;
0038;8;DIGIT EIGHT;
0039;9;DIGIT NINE;
0660;0;ARABIC-INDIC DIGIT ZERO;
0661;1;ARABIC-INDIC DIGIT ONE;
0662;2;ARABIC-INDIC DIGIT TWO;
0663;3;ARABIC-INDIC DIGIT THREE;
0664;4;ARABIC-INDIC DIGIT FOUR;
0665;5;ARABIC-INDIC DIGIT FIVE;
0666;6;ARABIC-INDIC DIGIT SIX;
0667;7;ARABIC-INDIC DIGIT SEVEN;
0668;8;ARABIC-INDIC DIGIT EIGHT;
0669;9;ARABIC-INDIC DIGIT NINE;
```

We regenerate the tables with the following:

```
./integers -d 20 tr_digits tr_digits.txt
```

This produces the following tables:

```
// utf/tr_digits.txt
//
// 20 code points.
// 2 states, 22 columns, 288 bytes
//
#define TR_DIGITS_START_STATE (0)
#define TR_DIGITS_ACCEPTING_STATES_START (2)
extern const unsigned char tr_digits_itt[256];
extern const unsigned char tr_digits_sot[2];
extern const unsigned char tr_digits_sbt[30];
```

```
// utf/tr_digits.txt
//
// 20 code points.
// 2 states, 22 columns, 288 bytes
//
const unsigned char tr_digits_itt[256] =
{
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       1,   2,   3,   4,   5,   6,   7,   8,    9,  10,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,

       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
      11,  12,  13,  14,  15,  16,  17,  18,   19,  20,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,  21,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,
       0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0

};

const unsigned char tr_digits_sot[2] =
{
       0,   16
};

const unsigned char tr_digits_sbt[30] =
{
     245,  22,   2,   3,   4,   5,   6,   7,    8,   9,  10,  11,  10,  22, 255,   1,
      11,  22, 245,   2,   3,   4,   5,   6,    7,   8,   9,  10,  11,  22
};
```

The input translation table maps input bytes to column numbers:

Otherwise	**0**
‘0’ (0x30)	**1**
‘1’ (0x31)	**2**
‘2’ (0x32)	**3**
‘3’ (0x33)	**4**
‘4’ (0x34)	**5**
‘5’ (0x35)	**6**
‘6’ (0x36)	**7**
‘7’ (0x37)	**8**
‘8’ (0x38)	**9**

‘9’ (0x39)	**10**
0xA0	**11**
0xA1	**12**
0xA2	**13**
0xA3	**14**
0xA4	**15**
0xA5	**16**
0xA6	**17**
0xA7	**18**
0xA8	**19**
0xA9	**20**
0xD9	**21**

The state transition table contains two states and twenty-two columns:

	Column																					
Previous	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
State 0	**22**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**1**
State 1	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**22**

The ten ASCII digits are still mapped to their corresponding value in State 0 after one lookup. For most other sequences, the decoding also stops after one lookup in State 0.

The UTF-8 encoding of U+0660 is 0xD9 0xA0. 0xD9 corresponds to column 21 which takes us to State 1. In State 1, values 0xA0 through 0xA9 are mapped to values 0 through 9 as expected.

Notice that the state machine does not necessarily advance through an entire UTF-8 sequence. So, you cannot use this to determine the length of a sequence. That has to be handled separately.