



AMICUS18®

www.protonbasic.co.uk



Powered by Proton Development Suite® Compiler by Les Johnson

Driving 3 to 8 seven segment displays & Keypad

PIC®, MPLAB®, PICkit3® and ICD3® are registered trademarks of Microchip Technology Inc®.
Proton Development Suite® or PDS® are a registered trademark of the author Les Johnson.

The project has been developed and written by Alberto Freixanet.

The document has been edited by John Drew.

07/23/2018: Manual version 1.1 The project has been compiled with PDS version 3.6.1.7.

INTRODUCTION:

I have recovered from the forum background a code to drive 4 seven segment displays. The author is unknown; it could be a Les Johnson code. I have renewed the code to adapt it to the new versions of the compiler (V3.6.1.7). I have tried to make the code as efficient and fast as possible. Now I have been able to extend the code to scan 8 digits.

Writing such a code in the main program is a very difficult task almost impossible without imposing a very strict timing for its application, restricting its possibilities. It is for this reason it is better to use interrupts to scan the digits.

This new project is a variant of the projects already published. The particularity of scanning is used to read a keyboard by adding 4 resistors and occupying only 4 pins in the PIC®.

The code has been tested with Proteus V8.2 SP0.

Multiplexing:

The devices that use this type of display are commonly found in digital clocks, electronic meters, counters, signaling and other equipment to show only numerical data. The display segments are usually referred to by the letters "a" through "g". To multiplex the 7 segment displays with the microcontroller, you must determine or look for the pattern corresponding to the digit to be shown. This can be something like a table to interpret a number into the corresponding segments for display. This interpreted number is sent to the port where the display is connected.

The speed of scanning:

With more connected digits more speed is required in the scan to avoid blinking.

Brightness of the segments:

For example in the case of 4 digits, the segments will be activated for 25% of the time. So, for this type of application it is absolutely necessary to use a high brightness display and adjust the current sufficiently.

Code speed:

The writing of the code is also important to avoid blinking. Parameters must be calculated and transferred without interrupts causing value errors. In my opinion it would be highly recommended to use a PIC® with the maximum speed e.g. 40 MHz, 48 MHz or 64 MHz depending on the family. We will see an example in the code.

The variables used depends on the number of digits, for example.

A Word will be used to value up to 999 (3 digits).

A Word will be used for a value up to 9999 (4 digits).

A Dword will be used for a value up to 99999 (5 digits).

A Dword will be used for value up to 999999 (6 digits).

A Dword will be used for value up to 9999999 (7 digits).

A Dword will be used for value up to 99999999 (8 digits).

You can intuit that to calculate the values of the 8 digits with a variable Dword the number of lines of code will be much more important. It is another argument to operate the clock of the PIC® at maximum frequency.

The hardware:

The demonstration program is prepared to use common cathode or common anode displays. If each segment could, on peak, consume 15mA, then the consumption of one digit could reach $15 \times 8 = 120\text{mA}$. Consequently, to enable a display it is necessary to use a transistor that will support this current and high performance and high brightness displays with low consumption must be used.

In the case of using the keyboard, you must minimize the number of pins used. That is why a demultiplexer of type 74HCT238 is used for the digits. The transistors are replaced by a ULN2803 that occupies less space on the PCB.

I present 2 types of schematics. When a PIC to +5V is chosen, an 8-bit PORT could be used to send the information to the 8 segments. When using the internal AD converter, I recommend current in the PIC® is minimised in order to reduce the noise in the power supply. To reduce the current, it is necessary to use a buffer such as model UDN2981A. In addition this configuration will also allow the use of +3.3V PICs® and a display powered with +5V, as drawn in the diagram. Only the resistors for the keyboard will be connected to +3.3V and all other hardware will be connected to the display voltage (+5V).

For 28-pin PICs® the configuration is slightly different. The configuration of the library allows connection of the keyboard (4 bits) in a Low Nibble or High Nibble, but not for the DIGITS PORT (4 bits) that must always be connected in a Low Nibble PORT. See the example with the PIC18F25K20 of the Amicus18 board.

Parameters of the test program:

You can configure all possible versions: 3, 4, 5, 6, 7 or 8 displays, Common cathode, common anode with transistors. Although, for the keyboard project I have chosen the common cathode model. Likewise the decimal point is configured automatically in the position defined by the variable float.

The code for the keyboard is considered as an option and can be removed from the compiled code.

The code has been tested with Proteus V8.20. It can be adapted to any CPU, and interrupt timing of 10mS (100Hz) must be generated by TMR0 or TMR2. You could use another Timer if you choose. The scanning and writing on the PORT is done by interrupt. The code of the Timer0 and Timer2 has been written as simply as possible for the tests. But the possibility of refining the value of timing is more complicated. Great precision is not very important either. It is the visual flicker test that will matter.

Configuration of Timer:

I have written 2 versions of Timer (Timer0 and Timer2), being the most common in all the PICs®. To reduce the number of code lines in the interrupt routine, I have not reloaded the Timer0 and the Timer2 as this is not required. This way you can save 6 execution cycles.

Code:

Context Save

If DisplayInterruptFlag = 1 **Then**

Code...

Code...

DisplayInterruptFlag = 0 *'/ Reset Timer interrupt flag*

EndIf

Context Restore

'/ Exit the interrupt routine

"DisplayInterruptFlag" is generic for the timer interrupt flag. This way you do not need to change the code when changing the timer. The Timer2 is easier to adjust by PR2 SFR but only uses one byte for the counter. It does not reach 10mS for a clock of 40, 48 and 64 Mhz (4 or 6 mS)

As the PICs® have many variations the PDS user should review the PIC® datasheet that will be used and modify the timer configuration accordingly.

Configuration of the DisplayInit Sub:

This routine contains all the PIC® and hardware parameters. The code will depend on each PIC® and it will be the responsibility of the user PDS to adapt it. In particular putting the pins to digital, not always a simple task for some PICs®.

Configuration of the PORTs:

The configuration of the ports must be done at the beginning of the program.

Code: (For the PIC18F25K20)

```
'// Definition of the PORTS for Display segments.
Symbol SEGMENTS = LATB '// PORT for Display segments = 8Bits

'// Definition of the pins to enable displays to write.
Symbol PORT_Digits = LATA '// PORT to enable the digits
'// Pins location in the PORT: Always = LowNibble
Symbol AddressDA = LATA.0 '// Address 0 of the Digits.
Symbol AddressDB = LATA.1 '// Address 1 of the Digits.
Symbol AddressDC = LATA.2 '// Address 2 of the Digits.
Symbol LE2 = LATA.3 '// Enable/Disable all Digits.

$ifdef _Keypad
'// Declare the position of the PORT Keypad LowNibble/HighNibble
Symbol PORT_Keypad = PORTC '// PORT to read the Keypad
'// Pins location in the PORT. (LowNibble/HighNibble)
$define _PORTKeypad LowNibble
Symbol KA = PORTC.0 '// Input A Keypad
Symbol KB = PORTC.1 '// Input B Keypad
Symbol KC = PORTC.2 '// Input C Keypad
Symbol KD = PORTC.3 '// Input D Keypad
$endif

'// Declare the pin of the Buzzer.
$ifdef _BuzzerOption
Symbol BUZZER = LATA.5 '// Output Buzzer
$endif
'//-----
'// Configurations of PORTS (automatic)
$ifdef _Keypad
    $if _PORTKeypad = LowNibble
        Symbol DipsKeypadMask = %00001111 '// Enable Low nibble.
        Symbol NOKEYpressed = 15
    $else
        Symbol DipsKeypadMask = %11110000 '// Enable High nibble.
        Symbol NOKEYpressed = 240
    $endif
$endif
'//-----
```

Data Table for Segments:

To perform the conversion of numbers to segments, a data table is needed.

In the original code, this table was written in ROM with CData. But I found out that by placing it in the eeprom, access to the data could be faster and provided that the table starts at address 0 of the eeprom. For beginners, it is necessary to know when the entry or exit of a routine is a byte, the compiler almost always places the result to the WREG SFR. In this code, one line could be deleted. Here is the conversion of the original code into new code to read the EEPROM.

```
$if TableData = 0
    '// Data from the ROM memory.
    WREG = dRsOutByteIn - 48 '// Convert ASCIIdec to BCD.
    INDF1 = CRead8 SegmentsData[WREG] '// Data for segments.
$else
```



```

        '/ Data from the Eeprom memory.
        EEADR = dRsOutByteIn - 48      '/ Convert ASCIIdec to BCD.
        EECON1bits_RD = 1             '/ Set the eeprom to read.
        INDF1 = EEDATA                '/ Data for segments.
    $endif

```

As the input is an ASCII character sent by the **RsOut** command, this code transform the data received to segments:

Data Table for Keypad:

To perform the conversion of **KEYS** to **Numbers**, a data table is needed. Using a **LookDown** command, the returned values can be re-arranged to correspond with the legends printed on the keypad. In this way you will need a table for each type of keyboard.

```

$if _Keypad = 4X4
'/KEY
dMyKey = LookDown dKeyPressed, [%01110010,%10110001,%10110010,%10110100,_,
'/4      0      1      2      3
%11010001,%11010010,%11010100,%11100001,%11100010,%11100100,%11101000,_,
'/X 11    5      6      7      8      9      10
%11011000,%10111000,%01111000,%01110100,%01110001]
$define KeypadExit 15
$endif

$if _Keypad = 4X3
'/KEY
dMyKey = LookDown dKeyPressed, [%01110010,%11100001,%11100010,%11100100,_,
'/4      0      1      2      3
%11010001,%11010010,%11010100,%10110001,%10110010,%10110100,%01110001,
'/# 11
%01110100]
$define KeypadExit 11
$endif

```

With the help of the preprocessor you could incorporate many different keyboards.

Very important:

The user must define a special constant [**KeypadExit**] using the last key. This will serve to determine the exit key and the last key of the keyboard.

Interrupt routine:

To do the scan and write the PORT: I have modified the original code to improve the flicker, in the old case it could be some microseconds. The displays are off when the value of the PORT is changed.

Code:

```

LE2 = 1      '/ All digits OFF.
$ifdef _ProteusTest
DelayCS 2     '/ Problem of Proteus. Could be disabled for real wold.
$endif
$if DisplayMode = 0
SEGMENTS = ~PRINT_TENS '/ For Commom Cathode display
$else
DelayCS 1     '/ Problem of Proteus. Must be same execution time to write.

```



```

SEGMENTS = PRINT_TENS    '/ For Commom Anode display
$endif
WREG = PORT_Digits & DipsAddressMask    '/ Read + Clear all bits of Digits
                                           '/ address + Multiplexer On.
PORT_Digits = WREG | 1    '/ Write DIGIT number 2

```

Originally the table was written for displays with common anode. Using a very fast trick the code can be adapted to display with common cathode.

```

$if TableData = 0    '/ ROM Table.
'/-----
SegmentsData: CData Byte
'/0 1 2 3 4 5 6 7 8 9 blank blank blank blank blank
192,249,164,176,153,146,131,248,128,152,255,255,255,255,255,255,255,
'/A B C D E F G H I J K L M N O P Q R
136,131,198,161,134,142,144,137,207,225,138,199,200,171,163,140,148,175,
S T U V W X Y Z [
146,135,227,193,213,137,145,164,255 '/ 44 numbers
'/ ":" is the character for Space on the display.
'/-----
$else    '/ Eeprom Table.
'/-----
'/ For more efficiency the Table EData must be placed in address 0 of the
eeprom.
SegmentsData EData Byte
'/0 1 2 3 4 5 6 7 8 9 blank blank blank blank blank
192,249,164,176,153,146,131,248,128,152,255,255,255,255,255,255,255,
'/A B C D E F G H I J K L M N O P Q R
136,131,198,161,134,142,144,137,207,225,138,199,200,171,163,140,148,175,
S T U V W X Y Z [
146,135,227,193,213,137,145,164,255 '/ 44 numbers
'/ ":" is the character for Space on the display.
'/-----
$endif

```

The table is lineal from the character "0" until the character "Z". The characters "Colon", "Semi-Colon", "Less Than", "Equal", "Greater Than", "Question Mark" and "AT Symbol" are read as a Blank character. To write a "space" of blank character a ":" (Colon) could be used.

Write the Segments with Common Anode & Common Cathode configuration:

Code:

```

$if DisplayMode = 0
SEGMENTS = ~PRINT_ONES    '/ For Common Cathode display
$else
SEGMENTS = PRINT_ONES    '/ For Common Anode display
$endif

```

Avoid the blinking caused by interrupts:

If the interrupt occurs during the execution of the routine "RsOut" we could send the digits some old values for the high and new digits for the low digits at the same time. To solve this problem I have created some buffers whose load is independent of the interrupts in the following way. The interrupt generated by the Timer stops for a very short time.

Code:

' Make a copy to print the values for the interrupt routine.

```
DisplayInterruptEnable = 0      '/ Avoid blinking.  
PRINT_ONES = ONES  
PRINT_TENS = TENS  
PRINT_HUNDREDS = HUNDREDS  
PRINT_THOUSANDS = THOUSANDS  
PRINT_TENTHOUSANDS = TENTHOUSANDS  
PRINT_HUNDREDTHOUSANDS = HUNDREDTHOUSANDS  
DisplayInterruptEnable = 1
```

Configuration of the Library:

The test program is completely configurable to obtain the necessary code to drive 3 to 8 displays with Keypad options.

```
'/=====  
'/ Uncomment the next line to disable the Proteus test.  
$define _ProteusTest  
'/-----  
' Device = 18F23K20  
' Device = 18F24K20  
Device = 18F25K20  
' Device = 18F26K20  
' Device = 18F43K20  
' Device = 18F44K20  
' Device = 18F45K20  
' Device = 18F46K20  
  
$ifdef _ProteusTest  
Declare Xtal = 8      ' For Proteus testing  
$else  
Declare Xtal = 64     ' For real PIC (64Mhz)  
$endif  
  
Declare Optimiser_Level = 3  
Declare Dead_Code_Remove = On  
Declare Float_Display_Type = Fast  
Declare Show_System_Variables = On  
Declare Create_Coff = On  
Declare Watchdog = OFF  
Declare Bootloader = OFF  
Declare Eeprom_Address $F00000  
'/=====  
'/ Configuration of the Config Fuses.  
'/ DEFINE THE PLL: On / OFF  
$define PLL_ConfigFuses OFF    '/ Must be always OFF for Proteus.  
  
'/ Enable the to use the internal Oscillator.  
'/ Uncomment the next line to enable the configuration.  
$define _InternalOSC_  
  
'/ Uncomment the Line to use the CLOCK OUT FUNCTION:  
'/ If CLKOUT function is enabled, CLKOUT on RA6 & Port function On
```



```

    '/ RA7 otherwise Port function On RA6 & RA7.
    $Define _CLKOUT_Function_
'/-----
'/ Configuration of the displays hardware.
'/ Declare Display Mode (0 => Common Cathode) (1 => Common Anode)
    '/ with transistors
    $define DisplayMode 0

    '/ Declare Number of Digits (3,4,5,6,7,8)
    $define NumberDigits 8

    '/ Declare the Delay to see the Error in the Display. (MS)
    $define ErrorDelayMS 2000
'/-----
'/ Configuration of the Keypad hardware.
    '/ Declare the Keypad option.
    '/ Comment the next line to disable the Keypad option
    '/ Define the Keypad type. (4X4/4X3)
    $define _Keypad 4X4

    '/ Declare the Keypad TimeOut. (MS)
    $define KeypadTimeoutMS 2000

    '/ Declare number or digits for Password. (4 to 8)
    '/ Comment the next line to disable the Password option
    $define PassWordDigits 6
'/-----
'/ Declare the Timer used for scanning of the display. (one only)
'/ Comment one line to disable the option.
    ' $define ScanTimer0
    $define ScanTimer2
'/-----
'/ Declare the delay for buzzer when a key is pressed.
'/ This delay must to be adjusted with the Interrupt Timing. (MS)
'/ Comment the next line to disable this option.
    $define BuzzerDelayMS 250
'/=====

```

The Challenge:

The idea of further increasing the possibilities of the 7 segment displays has occurred to me, using all the wonderful options offered by the Proton Development Suite compiler. The trick is taking a command capable of extracting and sending each digit of a specific number. Then any of the serial communication commands could be used. Thanks to Les Johnson for leaving us the possibility of modifying the low level routines, we can achieve some small wonders.

Proton allows software functions to be modified. I have modified the **RsOut** command to continue to use modifiers, but instead of outputting to a serial pin the output goes to the pins driving the 7 segment displays. This has made it easier to send integers, floats, signed variables, strings etc to the pins connected to the displays.

What is the RsOut command?

According to the PDS manual.

Rsout

Syntax

Rsout Item {, Item... }

Overview:

Send one or more Items to a predetermined pin at a predetermined baud rate in standard asynchronous format using 8 data bits, no parity and 1 stop bit (8N1). The pin is automatically made an output. Of course, we are not going to use any PIC® pin.

Parameters:

Item may be a constant, variable, expression, or string list. There are no operators as such, instead there are modifiers.

The modifiers allowed are listed below:

Modifier Operation

Bin{1..to number of display digits} Send binary digits

Dec{0..to number of display digits} Send decimal digits (amount of digits after decimal point with floating point)

Hex{1..to number of display digits} Send hexadecimal digits

Sbin{1..to number of display digits} Send signed binary digits

Sdec{0..to number of display digits} Send signed decimal digits

Shex{1..to number of display digits} Send signed hexadecimal digits

Rep c\n Send character c repeated n times

Str array\n Send all or part of an array

Cstr cdata Send string data defined in a Cdata statement.

The numbers after the **Bin**, **Dec**, and **Hex** modifiers are optional. If they are omitted, then the default is all the digits that make up the value will be displayed, so always write the number of required digits (very important) The **RsOut** command must be terminated by "**CR**" (Carriage Return) to end the process of sending the bytes to the display. If a floating point variable is to be displayed, then the digits after the **Dec** modifier determine how many remainder digits are sent i.e. numbers after the decimal point.

Code:

Dim MyFloat as Float

MyFloat = 3.145

Rsout Dec2 MyFloat,CR ' *Send 2 values after the decimal point*

The above code will send 3.14. If the digit after the **Dec** modifier is omitted, then 3 values will be displayed after the decimal point.

Code:

' *Display a negative value on the display.*

Symbol Negative = -200

Rsout Sdec Negative,**CR**

MySWord = -200

Rsout Sdec MySWord,**CR**

For the display application it is not advisable to use other modifiers. Other modifiers could cause a crash of the PIC® program because you cannot override/modify these.

Operation mode:

It is preferable to use these forms only.

RsOut Dec MyWord,**CR**

RsOut Dec3 MyFloat,**CR**

RsOut Hex3 MyWord,**CR**

RsOut Bin4 MyWord,**CR**

If you prefer to have control of the variables to avoid errors; use my macros

PrintDisplnt(Dec, MyDword) *' Only for Integer.*

PrintDisplnt(Bin8, MyWord) *' Only for Integer.*

PrintDispFloat(Dec3, MyFloat) *' Only for Float.*

PrintDispSign(SDec, MySDword) *' Only for Signed.*

PrintDispSign(Bin7, MySDword) *' Only for Signed.*



Printing a Float variable.

At the last minute, I have thought about incorporating the possibility to write **ASCII** characters in the display. The code is built in such a way that characters must be entered in **upper case** only as the compiler sends upper case for HEX characters.

RsOut "COMPILER",CR



You can use the colon character to print a space (Blank) on the display.

RsOut "MY:PDS",CR



A B C D E F G H



I J K L M N O P



Q R S T U V W X



Y Z 0 1 2 3 4

You will always have to take into account the number of displays used. Again, the **RsOut** command must be terminated with "**CR**". Without these, the PICmicro™ will continue to transmit data until an **overflow error** is detected.

Error detection:

As typos are very common, I have planned a detection of the number of incoming bytes to avoid printing erroneous values on the display. Since I cannot know what your code will be like, the only way to perform the operation is to use interrupts. You can check that your

program will continue running while printing the text "**Error**" on the displays for a configurable time ().

Principle of operation:

The biggest problem is transforming the data sent by the **RsOut** command with the MSB first. To solve this problem, an array of (Number of Displays + 1) bytes is used. The MSB of the number is written first in the top of the array and it keeps recording until position zero. As the characters move within the array, a blank character is written in the free positions. In this way, digits without numbers will always be turned off. Then the incoming number becomes LSB first.

How the new RsOut command works:

The first step is to disable the RsOut command:

#disable RsOut

The subroutine must start with the same origin label:

```
Asm  
RsOut  
EndAsm
```

Not to be confused with the new command, the label is written in ASM.

And it ends with:

Return

This routine, in addition to recognizing **ASCII** characters, will transform them directly into information about the segments in the array.

The **RsOut** code begins by writing the character received in the array and the number of bytes received increases. In addition, the command is canceled when an error is written on the display. A **Clrwdt** is also placed if the Watchdog has been declared. The eeprom is set to read [**EECON1** = 0] if the eeprom option has been chosen by the user.

```
*/ New RsOut Routine.  
Asm  
messg "RSOUT ROUTINE" */ To find this routine in the asm file.  
EndAsm  
Asm  
RsOut  
EndAsm  
#ifdef watchdog_req  
Clrwdt  
#endif  
If F_PrintErrorDelay = 1 Then  
    Return */ If overflow error return to the  
            */ RsOut command.  
EndIf  
INDF1 = WREG */ Put the value into the array.  
dRsOutByteIn = WREG */ Save it.  
Inc dBytesReceived */ Increments Number of bytes received.  
  
$if TableData = 1
```



```

EECON1 = 0          '/ Eeprom to read. (only once is sufficient)
$endif
Code...

```

For any unauthorized character or if the number of characters exceeds the maximum allowed by the array, an error will be generated. This one will be printed on the display for a time configurable by the user. As this error is managed by an interrupt, it will be completely independent of the user's code. In addition, the **RsOut** command will be disabled during this time.

For the code to be compatible with the identifiers, special characters are detected.

Dot character (.):

```

Case 46          '/ Looking for Dot ("."). (Floating DP)
Inc FSR1        '/ Looking for the last valid digit received.
INDF1.7 = 0     '/ Set DP as common Anode in last digit received.
Dec FSR1        '/ Restore the FSR1 pointer for next byte.
Dec dBytesReceived '/ It is not a new value.

```

The trick is to go back to the previous character and include the decimal point.

Minus character (-):

```

Case 45          '/ Looking for Minus ("-").
INDF1 = MinusCharacter '/ Load directly the display value.
Dec FSR1         '/ Next array address to write.

```

The value (45) is (-) in ASCII. It is very simple to enter the minus character in the array.

Control of unwanted characters:

```

Case > 90        '/ Looking for disallowed character values.
                 '/ NO increment the pointer. Value is not valid. Goto Error.
GoTo SetAnErrorDisp

```

All received **ASCII** characters greater than (Z) cause an error.

End of data entry:

To terminate the input of **ASCII** characters you will need a **CR** (Carriage Return) as the standard command to avoid disorienting the user.

```

Case 13          '/ Test final bytes; looking For Carriage Return.
INDF1 = 255      '/ Clear the CR and blank the position.
Dec dBytesReceived '/ It is not a valid byte.
GoTo CheckErrorBytesReceived '/ All bytes received already.

```

Control of the number of characters:

Since the array contains the characters for the display used, the number of characters is controlled to generate an error if necessary.

```

If dBytesReceived > LenghtDISP_STRING - 1 Then

```

Error printing on the display:

Depending on the number of digits on the display, a different word will be generated.

```

$if NumberDigits = 8
PRINT_ONES = 175          '/ Character r (Digit 1)
PRINT_TENS = 163          '/ Character o (Digit 2)

```



```

PRINT_HUNDREDS = 175           */ Character r (Digit 3)
PRINT_THOUSANDS = 175          */ Character r (Digit 4)
PRINT_TENTHOUSANDS = 134       */ Character E (Digit 5)
PRINT_HUNDREDTHOUSANDS = 255    */ Blank (Digit 6)
PRINT_THOUSANDTHOUSANDS = 255   */ Blank (Digit 7)
PRINT_TENTHDTHOUSANDS = 255     */ Blank (Digit 8)
$endif

```



Put the characters in LSB position:

In order to correctly display the characters on the display, the first character must be in the zero position of the array. This code moves all the characters to the right.

```

*/ Move the bytes from high position to lower position. (LSB first)
Dec dBytesReceived           */ prepare FSR1
FSR1 = dHighPointerFSR1 */ Restore FSR1 to higher byte of the array.
FSR1 = FSR1 - dBytesReceived */ Point FSR1 at the last byte received.
FSR0 = VarPtr (DISP_STRING) */ Point FSR0 to lower byte of STRING.
Inc dBytesReceived           */ Restore the correct value of dBytesReceived.
Repeat
    #ifdef watchdog_req
    Clrwdt
    #endif
    INDF0 = INDF1             */ Copy all bytes
    INDF1 = 255               */ Blanking old bytes
    Inc FSR0
    Inc FSR1
    Dec dBytesReceived
Until dBytesReceived = 0      */ Shift all the digits already.

```

Print the new characters on the display:

All new characters are copied to other variables to be displayed. The interrupt routine will take care of this task.

```

$if NumberDigits = 8
PRINT_ONES = ONES
PRINT_TENS = TENS
PRINT_HUNDREDS = HUNDREDS
PRINT_THOUSANDS = THOUSANDS
PRINT_TENTHOUSANDS = TENTHOUSANDS
PRINT_HUNDREDTHOUSANDS = HUNDREDTHOUSANDS
PRINT_THOUSANDTHOUSANDS = THOUSANDTHOUSANDS
PRINT_TENTHDTHOUSANDS = TENTHDTHOUSANDS
$endif

```

Restore the array:

Once copied the characters are no longer useful, be sure to delete the array with blank characters.

```

*/ Clear DISP_STRING, blanking all the digits.

```



```

FSR0 = VarPtr (DISP_STRING)
PRODH = LenghtDISP_STRING
Repeat
    #ifdef watchdog_req
    Clrwdt
    #endif
    POSTINC0 = 255    '/ Blank character.
    Dec PRODH
Until PRODH = 0
INDFO = 0                                '/ Set Array as String again.

```

Restore the PORTS:

For security reasons, the configuration of all the PORTS used is restored. (Better immunity to noise)

```

Output SEGMENTS                        '/ Put the display PORT as Output.
'/ Every individual pin is set as output.
Output AddressDA
Output AddressDB
Output AddressDC
Output LE2
'/ Every individual pin is set as input.
#ifdef _Keypad
Input KA
Input KB
Input KC
Input KD
#endif

```

Restore the pointers:

It is very important to restore all pointers to receive a new command **RsOut**.

```

'/ Configure the array pointers To start a new RsOut command.
FSR0 = VarPtr (DISP_STRING)    '/ FSR0 At the beginning of the array.
FSR1 = FSR0 + (LenghtDISP_STRING - 1) '/ FSR1 at the end of the array
                                         '/ to start a new RsOut command.
dHighPointerFSR1 = FSR1        '/ Make a copy of FRS1.
Return                          '/ End of routine RsOut.

```

Special commands:

Reminder for macros:

Macros not used in the main program do not occupy lines of code in the compiled file. In addition I have found that the variables of the macros will be declared only if the macro is used, for example:

```

#ifdef _mInkeyDisp, _mEnterNumber, _mEnterValue, _mPasswordInput
Dim dKeyPadFlags As Byte System
Dim F_KeyPressed As dKeyPadFlags.0
Dim F_KeyRead As dKeyPadFlags.1
Dim F_KeypadTimeOut As dKeyPadFlags.2
Dim F_KeyBuzzerDelay As dKeyPadFlags.3
Dim dKeyPressed As Byte System
Dim dKeyPressedOld As Byte System
Dim dMyKey As Byte System
Dim KeypadTimeOut As Word System

```



```

Dim dKeyReleasedCounter As Byte System
$ifdef _BuzzerOption
Dim KeyBuzzerDelay As Byte System
$endif
#endifMacro-

```

Macro InkeyD():

MyKey = InkeyD()

This command works in the same way as the **Inkey** command of the PDS. It should be used in a loop that will not have any delay incorporated. This code has been very difficult to write because the keys come from the interrupt routine that cannot be controlled very well from the main program.

The first pressed key is considered valid. In the other direction, an anti-bounce system is needed to detect the raised key. Self-repeating is not allowed. If there is no key pressed the result of "**MyKey**" will be equal to 255.



An option of a Buzzer can be defined by the user. When a key is pressed, an output of one pin of the PIC is activated. The duration of the buzzer, which is configurable, is performed by the interrupt routine.

```

$ifdef _BuzzerOption
F_KeyBuzzerDelay = 1                                '/ Enable the Buzzer pulse.
'/ This delay must to be adjusted with the Interrupt Timing.
KeyBuzzerDelay = VKeyBuzzerDelay                     '/ Time of the Buzzer pulse.
High BUZZER                                           '/ Start Buzzer.
$endif

```

Macro EnterNumber(pValue, pExit):

This powerful macro, made by the preprocessor, allows entering a value in a Byte / Word or Dword [**pValue**] with the keyboard being displayed in real time on the display. This command will be used for all variables. If you need this macro only once in your program, use this command. The variable [**pExit**], also defined by the user, is a control byte. If pExit equals 0 then an error has occurred. If pExit is equal to 255, the result is correct.

Macro EnterValue(pValue, pExit):

If you need the macro several times in your program, better to use this command that uses a subroutine to save lines of code. A transition variable [**DISPVAL**] will be used, which can be a Word or Dword variable according to the number of digits (automatic). If the user wants to use only one type of variable, you will have to manually change the definition.

Macro PasswordInput(pValue, pExit):

A variant of these macros could be an original way to enter a password key for any control. The trick is to use a Dword variable only. The keys will be displayed by a minus on the display. The number of keys previously defined by the user is detected. If there is an error or Timeout the variable [pExit] will be equal to 0. If the value is correct (not the password), [pExit] will be equal to 255. The code will be available in the variable [dPassWordVal] and will be passed to the variable defined by user in (pValue).

Note: All these last 3 commands have a Timeout planned and configurable by the user. See an example of use in the main test program.

CONCLUSION

PDS users can now use this template [[RSOutDigitsKeypad_xxK20.Inc](#)] for other PICs® applying the suggestions described above.

I have written many comments in the code to understand the operation of the library.

I hope that my comments on my code are accurate enough so that PDS users can modify the test code to make it their own.

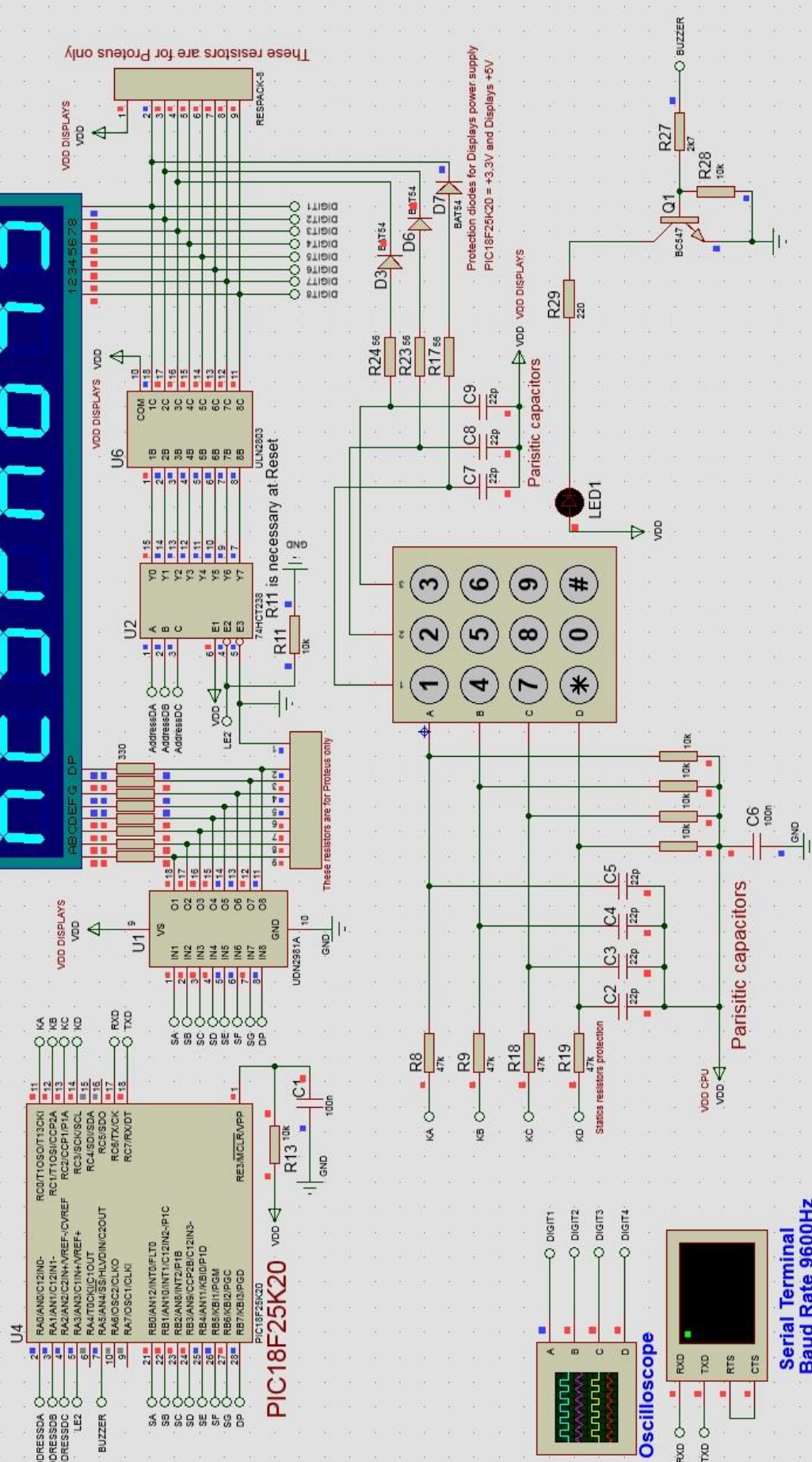
Good luck for your project.

8 Seven Segments Displays & Keypad V1.0. July 23 2018

Alberto Freixanet



8 X 7 Segment Displays COMMON CATHODE DISPLAY WITH DEMULTIPLEXER & 4x3 KEYPAD



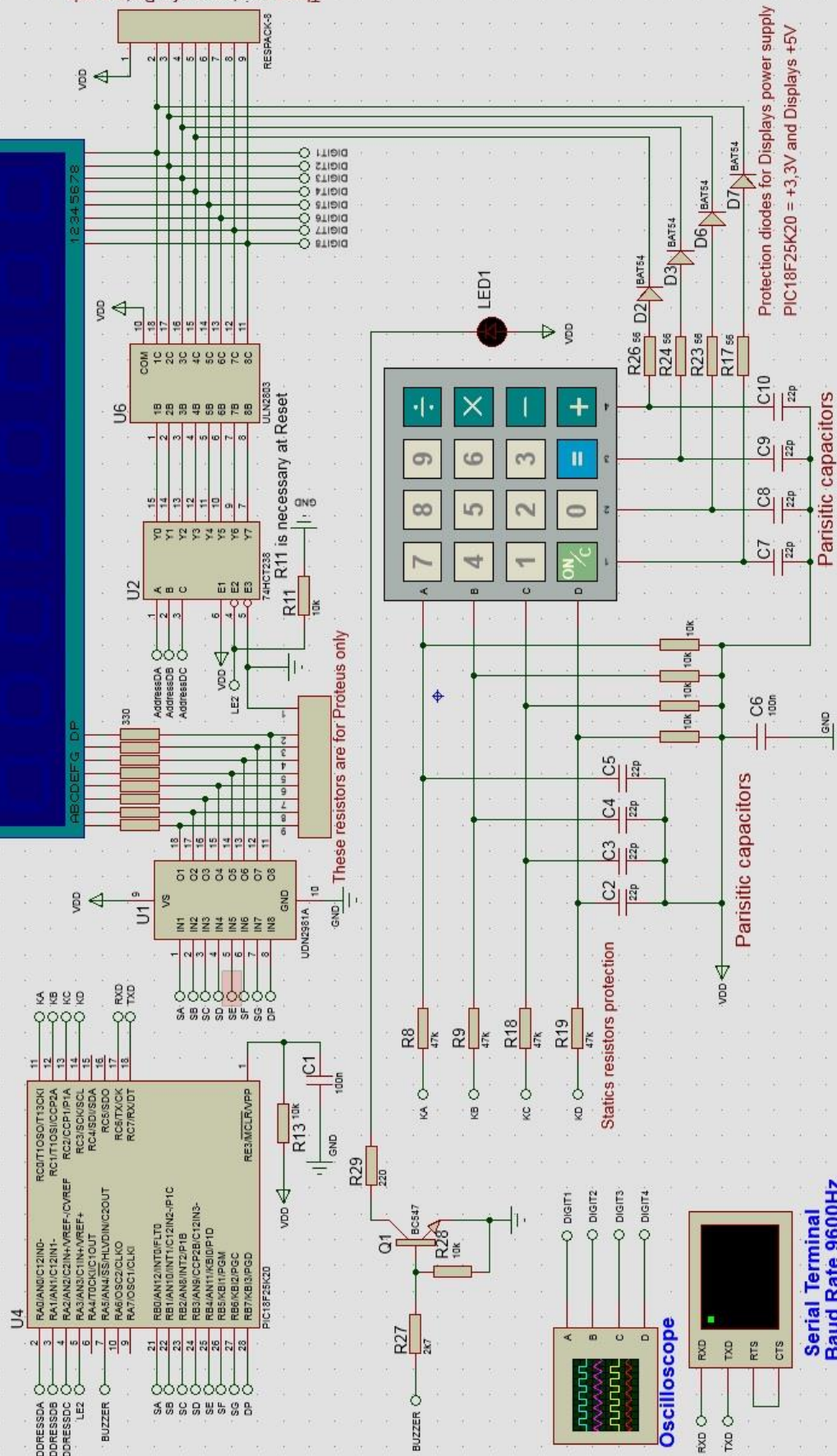
PDS PIC18 8 Displays & Keypad Evaluation Board

For a 18F25K20 PICmicro device

Please use the "R5OutDigitsKeypad_25K20.Inc" library



8 X 7 Segment Displays COMMON CATHODE DISPLAY WITH DEMULTIPLEXER & 4x4 KEYPAD



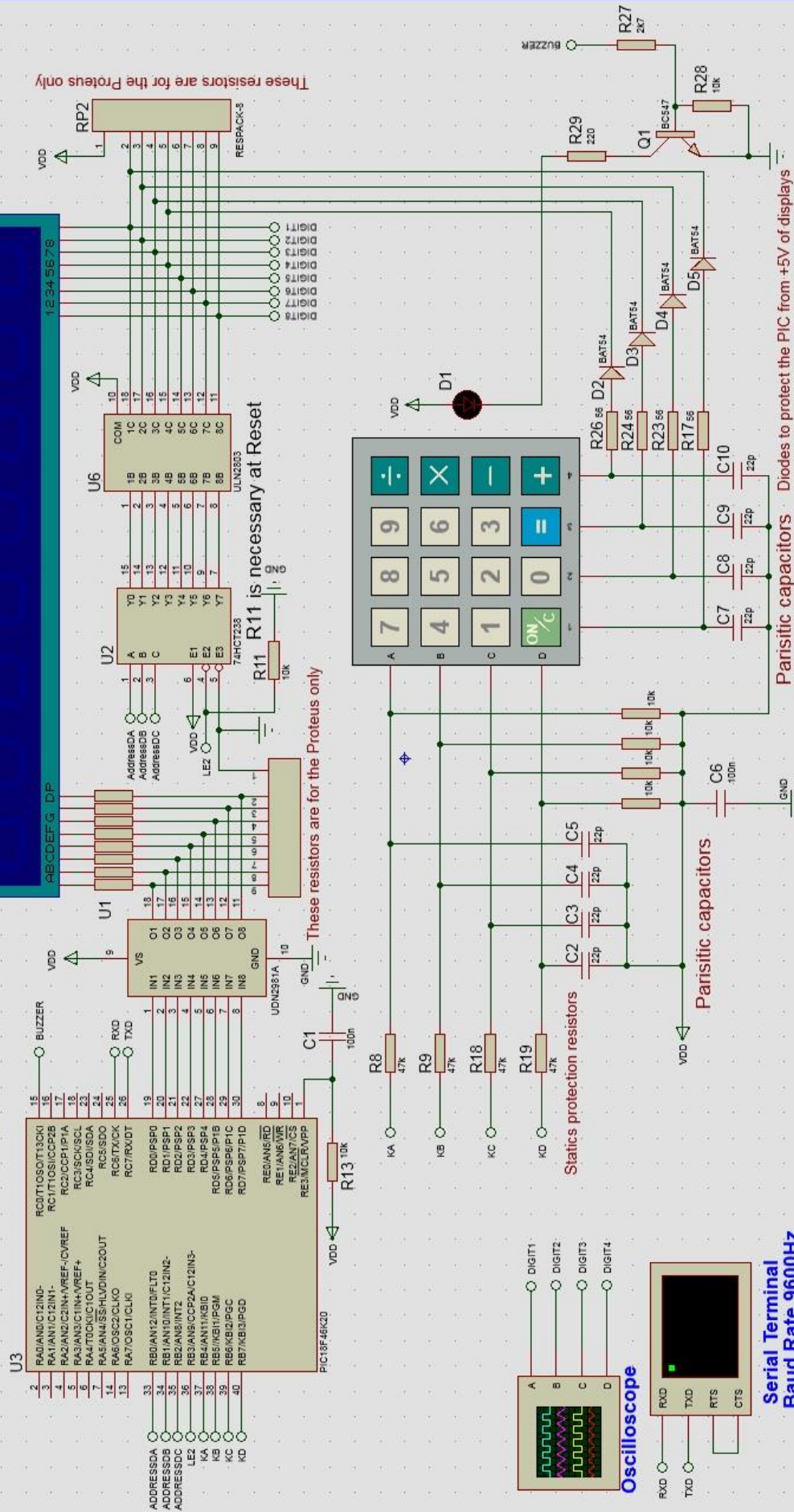
PDS PIC18 8 Displays & Keypad Evaluation Board

For a 18F25K20 PICmicro device

Please use the "R5OutDigitsKeypad_25K20.Inc" library

PROTEUS
The Complete Electronics Design System

8 X 7 Segment Displays COMMON CATHODE DISPLAY WITH DEMULTIPLEXER & 4x4 KEYPAD

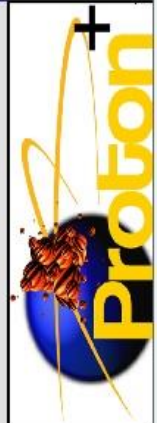
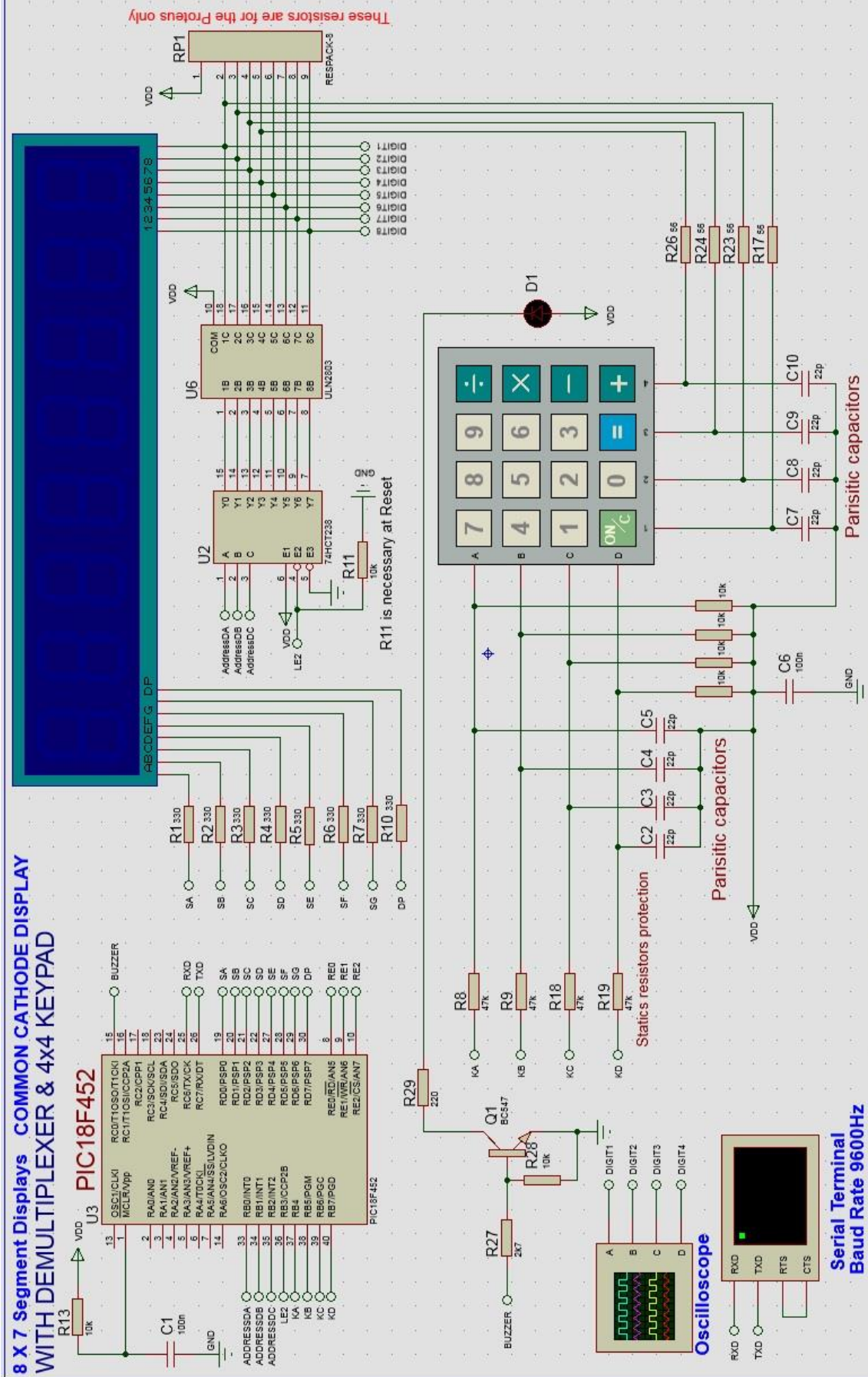


PROTON PIC18 8 Displays Evaluation Board

For a 18F46K20 PICmicro device

Please use the "R5OutDigitsKeypad_46K20.Inc" library

PROTEUS
The Complete Electronics Design System

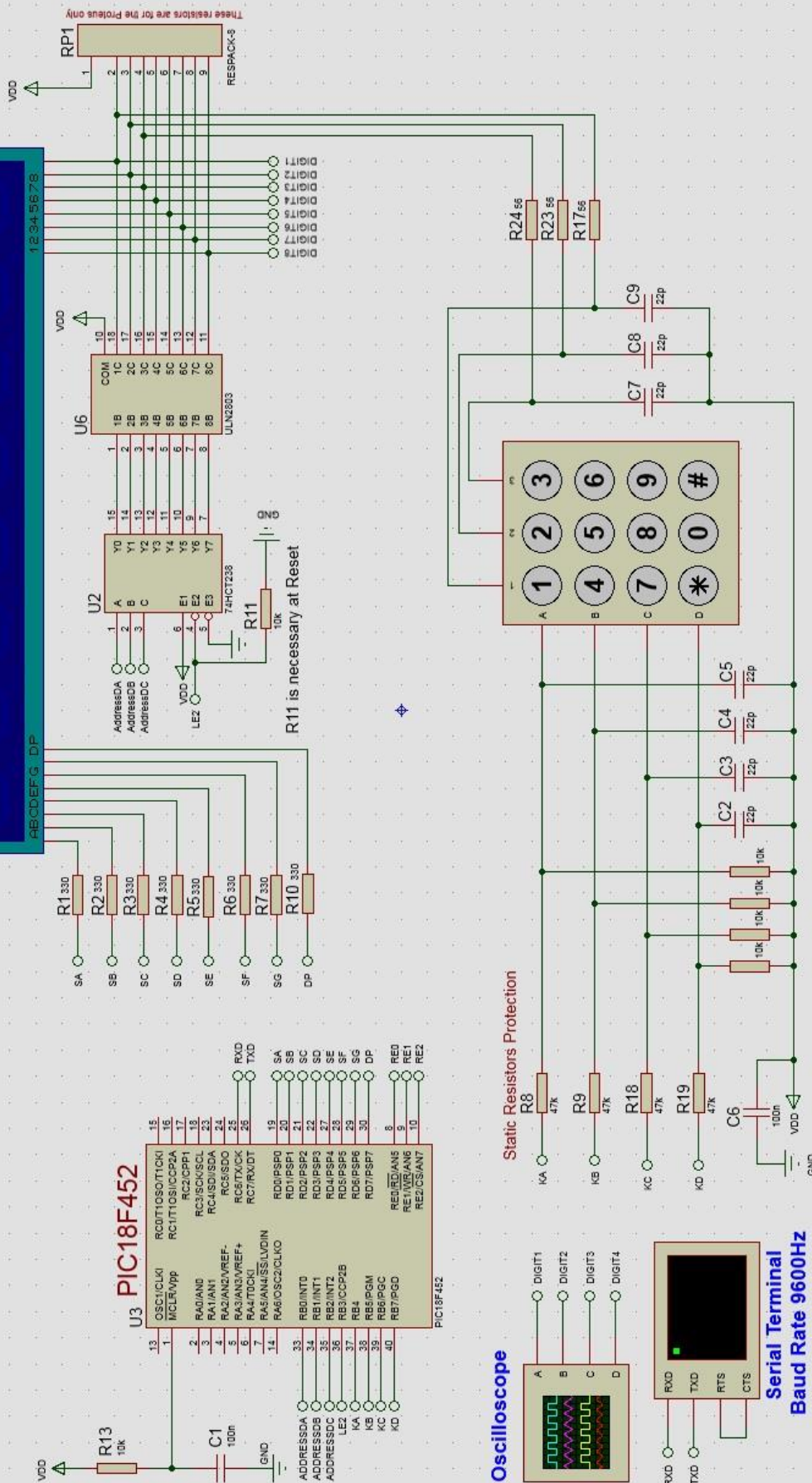


PROTON PIC18 8 Displays Evaluation Board

For a 18F452 PICmicro device
Please use the "R5OutDigitsKeypad_452.inc" library

PROTEUS

8 X 7 Segment Displays COMMON CATHODE DISPLAY WITH DEMULTIPLEXER & 4x3 KEYPAD



PROTON PIC18 8 Displays Evaluation Board

For a 18F452 PICmicro device
Please use the "R5OutDigitsKeypad_452.Inc" library

