

## Inhaltsverzeichnis

<b>AN007 - Application examples of the HMI2 device</b> .....	3
Wait for a KeyPress or more keys for some time .....	3
Create a recursive view .....	4
Create a text view .....	4
Create multiple recursive views mixed with text displays .....	5
Create a simple data input .....	5
Create a complex data introduction .....	6
Create a mixed non-recursive visualization .....	7
Diagnostic Inputs .....	8



## AN007 - Application examples of the HMI2 device

In this chapter we will analyze some programming examples useful to be able to perform basic functionality with the help of the HMI2 device. How will only use an D221 hardware platform, but the applicability of such examples, with any small changes, is extended to all microQMove hardware. It's a good idea, before using this device, define a constant value series (in the CONST section of the configuration unit of the Qcl application) to be inserted in the module configuration to improve readability and maintenance of the application developed.

```

*****
;Definition of values associated with the keys
*****
CONST
KEY_ENTER 1 ; enter key
KEY_CLEAR 8 ; clear key
KEY_PLUS 4 ; + key
KEY_MINUS 32 ; - key
KEY_F 16 ; F key

*****
;Definition of values associated with the leds
*****
LED_L1 2 ; L1 led
LED_L2 4 ; L2 led
LED_L3 8 ; L3 led
LED_L4 16 ; L4 led
LED_F 512 ; F key led
LED_AL 1 ; ALARM led

*****
;Definition of values associated with the display characters
*****
CHAR_ 35 ; display code for the <space> character
CHAR_0 0 ; display code for the 0 character
CHAR_1 1 ; display code for the 1 character
CHAR_2 2 ; display code for the 2 character
CHAR_3 3 ; display code for the 3 character
CHAR_4 4 ; display code for the 4 character
CHAR_5 5 ; display code for the 5 character
CHAR_6 6 ; display code for the 6 character
CHAR_7 7 ; display code for the 7 character
CHAR_8 8 ; display code for the 8 character
CHAR_9 9 ; display code for the 9 character
CHAR_A 10 ; display code for the a character
CHAR_B 11 ; display code for the b character
CHAR_C 12 ; display code for the c character
CHAR_D 13 ; display code for the d character
CHAR_E 14 ; display code for the e character
CHAR_F 15 ; display code for the f character
CHAR_G 16 ; display code for the g character
CHAR_H 17 ; display code for the h character
CHAR_I 18 ; display code for the i character
CHAR_J 28 ; display code for the j character
CHAR_K 40 ; display code for the k character
CHAR_L 19 ; display code for the l character
CHAR_M 43 ; display code for the m character
CHAR_N 20 ; display code for the n character
CHAR_O 21 ; display code for the o character
CHAR_P 22 ; display code for the p character
CHAR_Q 23 ; display code for the q character
CHAR_R 24 ; display code for the r character
CHAR_S 5 ; display code for the s character
CHAR_T 25 ; display code for the t character
CHAR_U 26 ; display code for the u character
CHAR_V 34 ; display code for the v character
CHAR_W 28 ; display code for the w character
CHAR_Y 27 ; display code for the y character
CHAR_UP 40 ; display code for the upper segment character
CHAR_CENTER 33 ; display code for the central segment character
CHAR_LOWER 36 ; display code for the lower segment character
CHAR_UPCEN 39 ; display code for the upper & middle segments character
CHAR_LOWCEN 37 ; display code for the lower & middle segments character
CHAR_LOWUP 42 ; display code for the lower & upper segments character
CHAR_LOWUPCE 38 ; display code for the lower & upper & middle segments character
CHAR_NONE 0 ; display code for the NONE (none char showed) character
CHAR_POINT &H80 ; bits that enable the decimal point

```

This methodology is important to apply it to all parameters formats from bit fields such as *scflags* or *deflags*; in this case we define for example:

```

SCRA_ENABLE 1 ; Bit screenA enabling viewing
SCRB_ENABLE 2 ; Bit screenB enabling viewing
SCRC_ENABLE 4 ; Bit screenC enabling viewing
SCRA_DISSIGN 8 ; Bit screenA disable sign
SCRB_DISSIGN 16 ; Bit screenB disable sign
SCRC_DISSIGN 32 ; Bit screenC disable sign
SCRA_DISLZB 64 ; Bit Leading zero blank (LZB) screenA disable
SCRB_DISLZB 128 ; Bit Leading zero blank (LZB) screenB disable
SCRC_DISLZB 256 ; Bit Leading zero blank (LZB) screenC disable

DE_ENABLE 1 ; Bit dataentry enable
DE_DISSIGN 4 ; Bit sign disable in data entry
DE_ENALIM 16 ; Bit enabling control limits

```

Insert then the HMI2 device with sampling time of 5ms in the specific section:

```

INTDEVICE
dvHMI      HMI2      5

```

In the following examples, the device will always be dvHMI.

### Wait for a KeyPress or more keys for some time

You want to write a Qcl program wait for the keystroke F for executing a subroutine. Simply verify that the key parameter having the bit for the F key active:

```
MAIN:
IF ( dvHMI:key ANDB KEY_F )
CALL MyFUNC
ENDIF
WAIT 1
JUMP MAIN

SUB MyFUNC
;-----
;subroutine code
;-----
ENDSUB
```

This code does not ensure that it is only pressed the F button: the MyFUNC function may also be called if they were pressed together with F key also other keys. To ensure the exclusivity of the pressure of F the code becomes:

```
IF ( dvHMI:key EQ KEY_F )
CALL MyFUNC
ENDIF
```

You want now write code that listens for the both pressure of the CLEAR and ENTER keys for at least 2 seconds:

```
IF ( dvHMI:key EQ (KEY_ENTER+KEY_CLEAR) )
IF tm01:remain EQ 0 ;check expired timer
CALL MyFUNC
ENDIF
ELSE
tm01=2000 ;timer is reloaded
ENDIF
```

## Create a recursive view

You want to write a program that enables a Qcl recursive view on the leftmost 4 display with sign and 2 decimal places. We decide for ease to use screenA. We must first set the number of characters you want to shown bearing in mind that the sign is a character; we can therefore say that the number of characters is the number of digits of the display that are occupied and manipulated by the view. The maximum and minimum values that will allow us to shown are 9999 and -999. If the data to be showed is less than this minimum value or greater than this maximum value, the display shows the out of range characters \$\$\$\$.

We'll set:

```
dvHMI:ncharA = 4
```

We will put our view on the leftmost display setting the offset value to:

```
dvHMI:offsA = dvHMI:numdis - 4
```

We set decimal point position to 2:

```
dvHMI:decptA = 2
```

Enable recursive view screenA by setting the corresponding enable of the scflags variable:

```
dvHMI:scflags = SCRA_ENABLE
```

Executing the above statement We automatically disabled the other two recursive views and we have enabled the display of the sign on screenA. In case we wanted to preserve the States of other screenB and screenC views we should have written:

```
dvHMI:scflags = dvHMI:scflags ORB SCRA_ENABLE ;screenA enable
dvHMI:scflags = dvHMI:scflags ANDB SCRA_DISSIGN ;screenA sign enable
```

Finally, you can simply update the screenA variable with the value you want to shown and normally contained in another variable of our program (in the example, suppose we use a variable with the *count* name):

```
dvHMI:screenA = count
```

The update operation of screenA must be continuously performed by our program with the refresh rate more appropriate for reasons of functionality that the programmer has planned for that variable.

## Create a text view

You want to write a Qcl program that writes on display „HELLO“ right-aligned. To do this, just set the variables associated with the digit of the display the code of the character that you want to shown. We will therefore have:

```
;Print "HELLO"
dvHMI:dis6 = CHAR_
dvHMI:dis5 = CHAR_
dvHMI:dis4 = CHAR_H
dvHMI:dis3 = CHAR_E
dvHMI:dis2 = CHAR_L
```

```
dvHMI:dis1 = CHAR_L
dvHMI:dis0 = CHAR_O
```

**Note:**

In order to work properly, must not be active recursive views that overwrite all or part of interested digit by our „HELLO“. Check that in the *scflags* parameter the 0,1 and 2 bits are to 0 or force them to that value.

## Create multiple recursive views mixed with text displays

You want to create a view consists of two fixed texts and two recursive values. As an example, you shown a time in seconds and a program number. The desired show might be: „t51 Pr2“ where „t“ indicates the time, „51“ is the time value, „Pr“ it's a text that indicates the program, „2“ indicates the program number.

First we print the texts:

```
dvHMI:dis6 = CHAR_T
dvHMI:dis3 = CHAR_
dvHMI:dis2 = CHAR_P
dvHMI:dis1 = CHAR_R
```

Then we set the data for the numerical display of the time through the screenA.

```
dvHMI:ncharA = 2
dvHMI:offsA = 4
dvHMI:decptA = 1
dvHMI:scflags = dvHMI:scflags ORB SCRA_DISSIGN
```

We then the data for the numerical display of the program using the screenB.

```
dvHMI:ncharB = 1
dvHMI:offsB = 0
dvHMI:decptB = 0
dvHMI:scflags = dvHMI:scflags ORB SCRB_DISSIGN
```

We enable the two views:

```
dvHMI:scflags = dvHMI:scflags ORB SCRA_ENABLE ORB SCRB_ENABLE
```

Then recursively we will update the view data:

```
dvHMI:screenA = glTime
dvHMI:screenB = glProgram
```

## Create a simple data input

You want to write a Qcl program that allows the user to input a value to a variable , for example, one used to store a pieces counting. First we will declare that variable, for example *cntPieces* in the section of the configuration unit. Suppose you want to view the „CP“ message on the left side of the display to indicate the introduction of pieces counting, and that the value to be introduced is 4 characters and positioned on the far right of the display. The data entry will occupy the dis0, dis1, dis2, dis3 display while the message is written in dis5 and dis6.

```
dvHMI:dis6 = CHAR_C
dvHMI:dis5 = CHAR_P
dvHMI:deoffs = 0
dvHMI:denchar = 4
```

The position of the decimal point will be placed to 0 and we will copy the value of the current pieces in the *devalue* parameter count to ensure that data appears at the entrance of the introduction that value on the display.

```
dvHMI:dedecpt = 0
dvHMI:devalue = cntPieces
```

Finally we will enable the data input using the appropriate flag, we will disable the sign (a pieces counter cannot be negative) and activate the introduction with the *DATAENTRY* command:

```
dvHMI:deflags = DE_ENABLE ORB DE_DISSIGN
DATAENTRY dvHMI
```

At this point the most significant digit on the display will start flashing the value of *cntPieces* and you will have to wait for the user to enter the data and confirm with the ENTER button. Then you must read the introduced data (in the *devalue* parameter) and copy it into our variable *cntPieces* of pieces counting. The *st\_dentry* state lets us know if data entry is active o expect this go to 0 before copying:

```
WHILE (dvHMI:st_dentry)
  WAIT 1
ENDWHILE
cntPieces = dvHMI:devalue
```

At this point the *cntPieces* variable is updated with the value entered by the user.

## Create a complex data introduction

You want to write a Qcl program that allows the user to input a value to a variable, as in the previous example, but with the following additional features:

- check that the figure is between 1 to 1000 and otherwise show „Error“ for 1 second and repeat the data entry
- if the F key is pressed you step out of the data input without storing the data introduced and may be printed for a second the „Exit F“ message
- If the CLEAR key is pressed you step out of the data input without storing the data introduced and may be printed for a second the „Exit C“ message
- print for a second the „MODiFY“ message if the introduced data has been modified

### Control data limits

To enable bounds checking of the introduced data you must enable this feature putting to 1 the relevant bits of the *deflags* parameter and set in *deuplim* and *delowlim* parameters the values of the upper and lower limits. Compared to the previous example code we will add, before the *DATAENTRY* command, the following Qcl instructions:

```
dvHMI:deuplim = 1000
dvHMI:delowlim = 1
```

and replace the setting instruction of the *deflags* parameter:

```
dvHMI:deflags = DE_ENABLE ORB DE_DISSIGN ORB DE_ENALIM
```

### Configure one or more keys to exit from data entry

To enable the output from data entry with a key You must set the *deExKeymask* parameter that is the form to exit buttons. To enable a button to function as data entry exit key, simply activate the corresponding bit of the above mentioned parameter. So if we want to ensure that you exit from data entry with the F and CLEAR keys you must insert the following *DATAENTRY* command instruction Qcl:

```
dvHMI:deExKeymask = KEY_CLEAR ORB KEY_F
```

### Check if the introduced data is within limits

When you exit from the data entry (then with the *st\_dentry* = 0 State), check the value of the *st\_uplim* and *st\_lowlim* states to know if the data introduced is in excess of the limits set. If *st\_uplim* vale 1 means that the input value is greater than the upper limit, while if *st\_lowlim* vale 1 means that the input value is less than the lower limit. Then we will check those states, and we will make a call to the ERROR subroutine (that will display the error message for 1 second) if the limits are exceeded.

```
;Data limits control
IF ( dvHMI:st_uplim OR dvHMI:st_lowlim )
    CALL ERROR ;print error message
    JUMP Dentry ;return datai introduction
ENDIF
```

### Check the output key from data entry

Checking the *deExitKey* parameter and the *st\_modified* and *st\_exitcmd* states, you can understand in what way you are signed out from data entry. The following table summarizes the possible conditions:

<i>deExitKey</i>	<i>st_exitcmd</i>	Description
0	0	Exit with confirmation by pressing the ENTER key or by <i>EXITDEC</i> command
0	1	Exit without confirmation by <i>EXITDE</i> command
!=0	X	Exit without confirmation by pressing the button identified by the value of the <i>deExitKey</i> parameter

### Check if the data has been modified

To check if the introduced data has changed, simply check the *st\_modified* status. It takes the 1 value If the input value is different from the previous value of the *devalue* parameter befor of the *DATAENTRY* command.

The full program will then:

```
LABEL0:
    dvHMI:dis6 = CHAR_C
```

```

dvHMI:dis5 = CHAR_P
dvHMI:dis4 = CHAR_
dvHMI:deoffs = 0
dvHMI:denchar = 4
dvHMI:decpt = 0
dvHMI:devalue = cntPieces
dvHMI:deuplim = 1000
dvHMI:delowlim = 1
dvHMI:deExKeymask = KEY_CLEAR ORB KEY_F
dvHMI:deFlags = DE_ENABLE ORB DE_DISSIGN ORB DE_ENALIM
DATAENTRY dvHMI

WHILE (dvHMI:st_dentry)
  WAIT 1
ENDWHILE

IF dvHMI:deExitKey
  ;--Output from data entry with output keys
  dvHMI:dis6 = CHAR_E
  dvHMI:dis5 = CHAR_H
  dvHMI:dis4 = CHAR_I
  dvHMI:dis3 = CHAR_T
  dvHMI:dis2 = CHAR_
  dvHMI:dis1 = CHAR_
  IF dvHMI:deExitKey EQ KEY_F
    dvHMI:dis0 = CHAR_F ;F key press
  ELSE
    IF dvHMI:deExitKey EQ KEY_CLEAR
      dvHMI:dis0 = CHAR_C ;CLEAR key press
    ENDIF
  ENDIF
ELSE
  ;--Output from data entry with confirm
  ;--Limits control
  IF ( dvHMI:st uplim OR dvHMI:st lowlim )
    CALL ERROR ;print error message
    JUMP LABEL0 ;return to data entry
  ENDIF
  ;--Checks if the data has changed
  IF dvHMI:st modified
    dvHMI:dis6 = CHAR_
    dvHMI:dis5 = CHAR_M ;print "MODIFY" message
    dvHMI:dis4 = CHAR_O
    dvHMI:dis3 = CHAR_D
    dvHMI:dis2 = CHAR_I
    dvHMI:dis1 = CHAR_F
    dvHMI:dis0 = CHAR_Y
    tm01 = 1000
    WAIT tm01
  ENDIF
  cntPieces = dvHMI:devalue ;stores entered value
ENDIF

MAIN:
  WAIT 1
  JUMP MAIN

SUB ERROR
  dvHMI:dis6 = CHAR_ ;print "ERROR" message
  dvHMI:dis5 = CHAR_E
  dvHMI:dis4 = CHAR_R
  dvHMI:dis3 = CHAR_R
  dvHMI:dis2 = CHAR_O
  dvHMI:dis1 = CHAR_R
  dvHMI:dis0 = CHAR_
  tm01 = 1000
  WAIT tm01
ENDSUB

END

```

## Create a mixed non-recursive visualization

You want to create a view of a message consisting of the „Error“ string and an identification number of the error that appears when occurs an error, while normally appears, recursively the counter value. To achieve this, we exploit the functioning can only be displayed by a numerical value present in *DATAENTRY* command functionality and enabled by setting to 0 the *DE\_ENABLE* bits of the *deFlags* parameter. For simplicity, we'll create a fictitious error condition by the end of a timer uploaded to 5 sec. As will see, It will be important to remember to disable recursive view before showing the error message, otherwise the result will not be what you expect.

The code is:

```

MAIN:
  ;stampa "C"
  dvHMI:dis6 = CHAR_C

  ;Configure and enable screenA
  dvHMI:ncharA = 6
  dvHMI:offsA = 0
  dvHMI:decptA = 0
  dvHMI:scFlags = dvHMI:scFlags ORB SCRA_ENABLE

  tm01 = 5000 ;how to use the timer to cause an error

LOOP:
  dvHMI:screenA = (dvHMI:screenA + 1) % 999999

  ;Errore?
  IF tm01
    ;disable screenA
    dvHMI:scFlags = dvHMI:scFlags ANDB ( NOT SCRA_ENABLE )
    CALL ERROR
    errNum = errNum + 1
    JUMP MAIN
  ENDIF

  WAIT 1
  JUMP LOOP

SUB ERROR
  ;print "ERROR"
  dvHMI:dis6 = CHAR_E
  dvHMI:dis5 = CHAR_R

```

```

dvHMI:dis4 = CHAR_R
dvHMI:dis3 = CHAR_0
dvHMI:dis2 = CHAR_R

;printing error with the ID
dvHMI:deoffs = 0
dvHMI:denchar = 2
dvHMI:ddecpt = 0
dvHMI:devalue = errNum
dvHMI:deflags = DE_DISSIGN
DATAENTRY dvHMI

;wait 2 seconds
tm01 = 2000
WAIT tm01

```

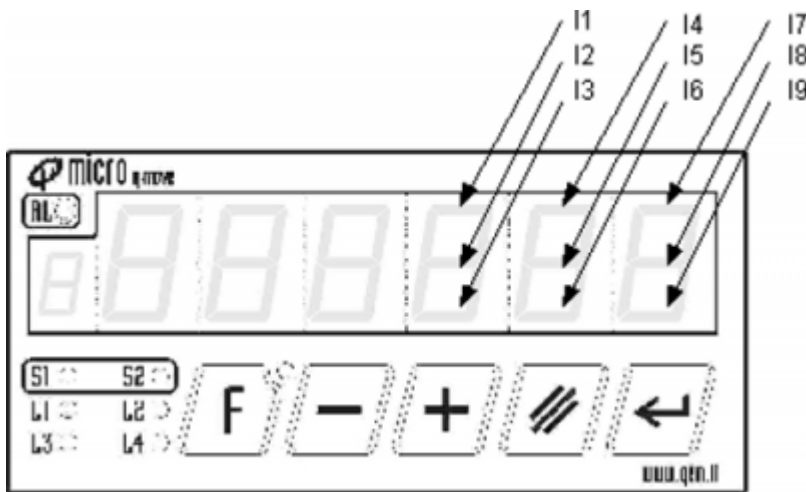
ENDSUB

END

## Diagnostic Inputs

You want to create a view that represents the State of 9 digital inputs. The same example can be used for the representation of digital outputs. We will assign to each input, one of the segments of each of the three rightmost digit and we will activate when the corresponding input will be active.

The figure shows the assignment chose for the inputs and segments of the digits of the display:



First we will declare, in the configuration unit, 9 variables of FLAG dimensions whose value will simulate the condition of 9 digital inputs.

```

GLOBAL
gfInp01      F
gfInp02      F
gfInp03      F
gfInp04      F
gfInp05      F
gfInp06      F
gfInp07      F
gfInp08      F
gfInp09      F

```

We will declare a global array to 8 items that will serve to hold character codes to print for each combination of inputs.

```

ARRGBL
diagnTab      B      8      ;character table for Diagnostics

```

In fact, for each group of three inputs associated with one of the three digits on the display we will have 8 possible combinations. For example, the table summarizes the possible States of the digit associated with, the combination of inputs I7, I8 and I9:

I7	I8	I9	Display
0	0	0	
0	0	1	
0	1	0	-
0	1	1	=
1	0	0	
1	0	1	~
1	1	0	*
1	1	1	\$

Will finally also defining some constants to be used as a mask for generic bits of a byte:

```

CONST
;-- Generic bit field mask -----

```



```

B_00      &H01      ; value for bit 00
B_01      &H02      ; value for bit 01
B_02      &H04      ; value for bit 02
B_03      &H08      ; value for bit 03
B_04      &H10      ; value for bit 04
B_05      &H20      ; value for bit 05
B_06      &H40      ; value for bit 06
B_07      &H80      ; value for bit 07

```

The complete code to obtain the diagnostic function is:

```

;Initializes table
diagnTab[1] = CHAR_
diagnTab[2] = CHAR_UP
diagnTab[3] = CHAR_CENTER
diagnTab[4] = CHAR_UPCEN
diagnTab[5] = CHAR_LOWER
diagnTab[6] = CHAR_LOWUP
diagnTab[7] = CHAR_LOWCE
diagnTab[8] = CHAR_LOWUPCE

;print "INP." message
hmi:dis6 = CHAR_I
hmi:dis5 = CHAR_N
hmi:dis4 = CHAR_P ORB CHAR_POINT

MAIN:
hmi:dis2 = diagnTab[(gfInp01 * B_00 + gfInp02 * B_01 + gfInp03 * B_02) + 1]
hmi:dis1 = diagnTab[(gfInp04 * B_00 + gfInp05 * B_01 + gfInp06 * B_02) + 1]
hmi:dis0 = diagnTab[(gfInp07 * B_00 + gfInp08 * B_01 + gfInp09 * B_02) + 1]

WAIT 1
JUMP MAIN
END

```

Documento generato automaticamente da **Qem Wiki** - <https://wiki.qem.it/>

Il contenuto wiki è costantemente aggiornato dal team di sviluppo, è quindi possibile che la versione online contenga informazioni più recenti di questo documento.