Embedding it better...

µTasker Document

µTasker – Keypad User's Guide, including Port Simulation

Table of Contents

1. Introduction

Many projects require that a user can control a device via a keypad. This can be a simple push button, a small number of separate such buttons or a keyboard type with a large quantity of keys.

The µTasker project supports a key scan module which allows a number of processor ports to be combined to create a matrix keypad which is scanned at a defined rate. A matrix keypad allows an increased quantity of individual keys compared to the number of processor ports used (eg. 8 port lines can scan up to 64 individual keys).

Alternatively, the module can be used to operate without a matrix, where each key is connected to its own input port.

The defined keypad/front-panel is displayed when running the project in the µTasker simulator, allowing the hardware configuration and keypad entry to be fully tested.

This document also explains the way that ports are simulated in the µTasker project:
- how to monitor the state and use of pins
- how to defined that input pins are connected to front-panel keys
- how to define that outputs are driving front-panel LEDs
- how to use and set up a three-state analogue based switch
- how to use scripts to simulate user input sequences
- notes about interrupt driven inputs
- notes about inputs triggering DMA transfers

2. Matrix Keypad Scanning Operation

Keyboard scanning is a typical application to multiplex a higher quantity of keys to a smaller number of ports. The following diagram illustrates a typical 4 x 4 scanning matrix, with 4 rows and 4 columns.

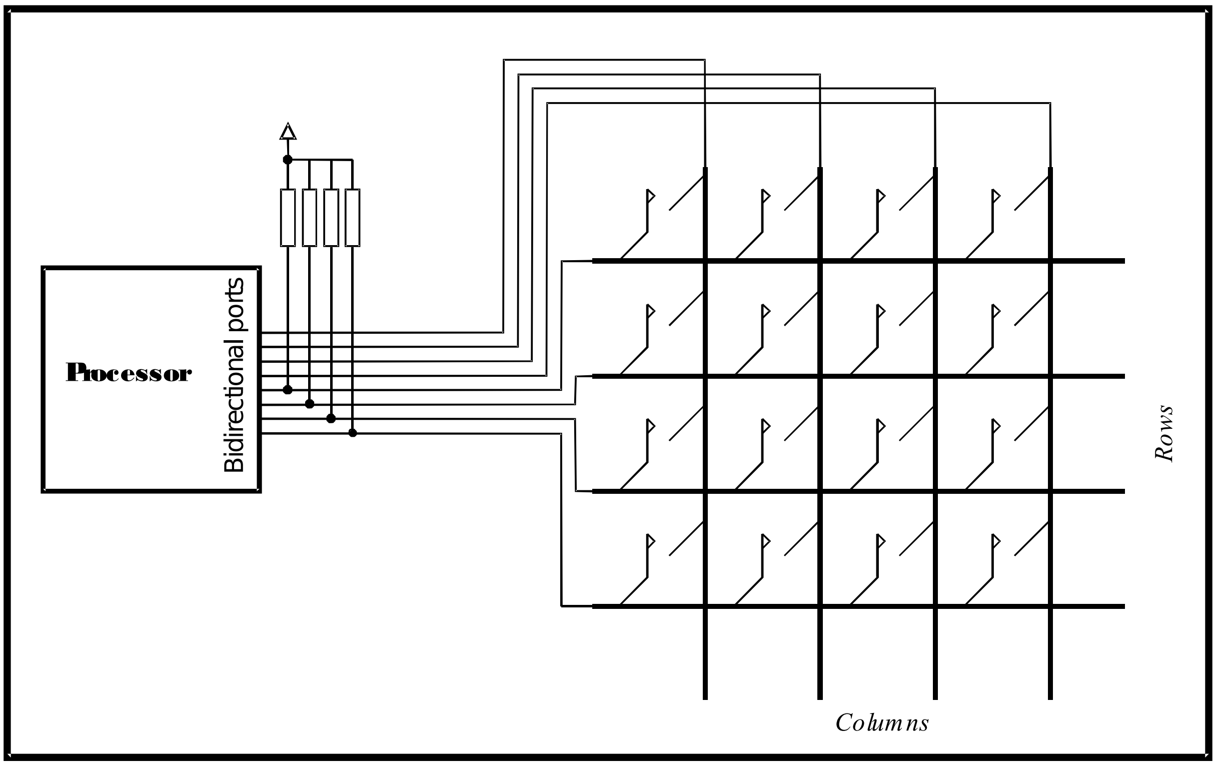


Figure 1. Typical 4 x 4 matrix keyboard connected to a processor port

The operation requires that the scanning software drives one output at a time (typically driving a '0' to each column and using pull-up resistors to define the '1' state of an open line) and the keys of each row belonging to the column are read. This simplest strategy allows a limited amount of keys to be pressed at the same time before a leakage path is created. In figure 2 it is shown that pressing three particular keys causes two rows to be detected when a single column is activated, which is false since only one key actually belonging to the driven column is in the pressed state.

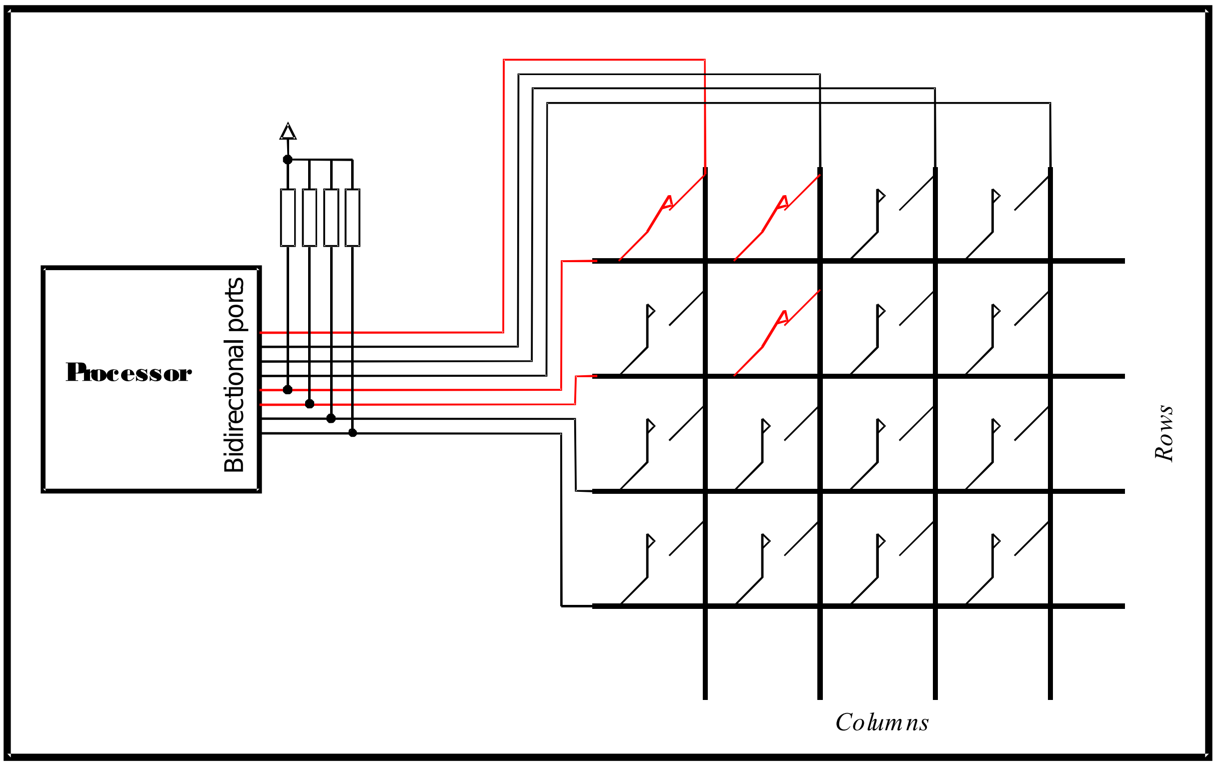


Figure 2. Leakage path when multiple keys are pressed, causing false key detection

By adding diodes in series with each key leakage paths are excluded, allowing the keypad scanning to recognise any number of simultaneous key presses. *Other strategies may also be possible to achieve the required solution*.

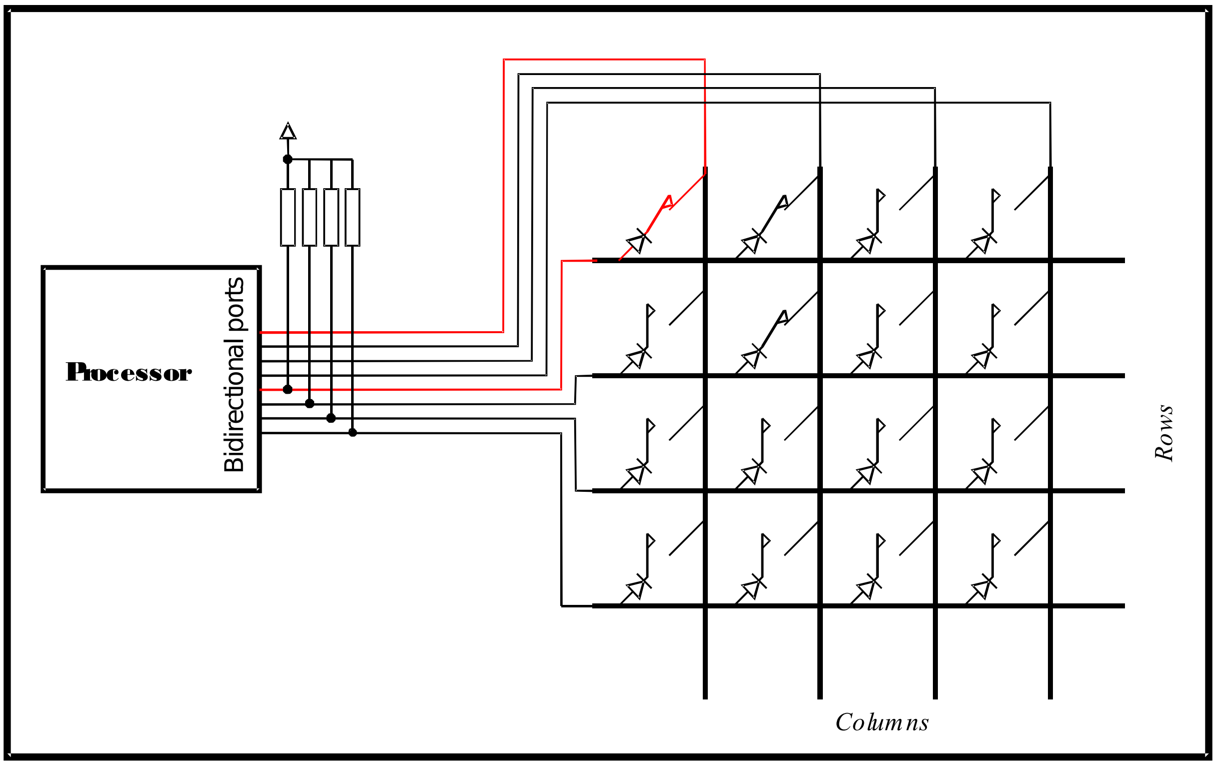


Figure 3. The use of series diodes in the keys to avoid leakage paths and allow any number of simultaneously pressed keys to be recognised correctly

3. Matrix Keypad Scanning Task

The µTasker project define `SUPPORT_KEY_SCAN` in `config.h` enables the key scan operation. The number of rows and columns is then specified by

```
    #define KEY_ROWS          4                        // 4 rows by
    #define KEY_COLUMNS       4                        // 4 columns

    #define KEYPAD_PARTNER_TASK    TASK_APPLICATION  // key change events sent to this task
```

whereby a maximum of 8 of each is supported (up to 64 keys).

When the key scan support is enabled a key scan task is added to the system (see `TaskConfig.h`) - `extern void fnKey(TTASKTABLE *);` which polls with a defined rate (eg. 100ms). The task performs the scanning sequence based on hardware macros as configured in `app_hw_xxxx.h` (where `xxxx` is the target processor used). For each key change an event is generated which can be handled by the application task. There is a separate event for a key press and for a key release, which allows the application to detect any combination of keys being pressed, held and released.

The key scan task performs the following tasks:

```
// This task is assumed to be polling at an adequate rate
//
void fnKey(TTASKTABLE *ptrTaskTable)                        // key scan task
{
       fnUpdateInputs();                                    // update the present input states
       fnCheckKeyChanges();                                 // generate events on changes
}
```

`fnUpdateInputs()` performs the scanning function and collects the complete instantaneous key pad state (which keys are presently pressed and which are presently released).

`fnCheckKeyChanges()` dispatches events for each change that occurs.

The following illustrates the scanning of a single column:

```
static void fnUpdateInputs(void)
{
    DRIVE_COLUMN_1();                    SIM_MATRIX_KB(); // set as output to drive column low
    fnGetCol(0);
    RELEASE_COLUMN_1();              // set column to high impedance (with active drive high)
}
```

First the output column is driven to logic '0'. The row information is read in (see `fnGetCol(0)` below) while the single column is being driven. Finally the column is released, meaning that it is set back to a high impedance state; *note that the output is however momentarily set to logic '1' before being set to an input to ensure that the row states are quickly forced a logic '1' state before the line is floated.* If this is not performed, the key capacitance can cause the row to be incorrectly read as '0' (pressed) on the next column scan since it is still in the process of being changed to the '1' state.

An input row of width 4 is shown being read below:

```
// A column is being driven, collect the corresponding row inputs
//
static void fnGetCol(int iRow)
{
    unsigned char ucBits = 0x01;
    ucCols[iRow]    = 0x00;

    if ((KEY_ROW_IN_PORT_1 & KEY_ROW_IN_1) == 0) {
        ucCols[iRow] |= ucBits;
    }
    ucBits <<= 1;
    if ((KEY_ROW_IN_PORT_2 & KEY_ROW_IN_2) == 0) {
        ucCols[iRow] |= ucBits;
    }
    ucBits <<= 1;
    if (KEY_ROW_IN_PORT_3 & KEY_ROW_IN_3) == 0) {
        ucCols[iRow] |= ucBits;
    }
    ucBits <<= 1;
    if ((KEY_ROW_IN_PORT_4 & KEY_ROW_IN_4) == 0) {
        ucCols[iRow] |= ucBits;
    }
}
```

The use of macros to read each row input allows the ports to be positioned anywhere on the processor and the code to be made portable between various processor types; *the same is also true for the column driving*.

4. Configuring the Hardware

The following example shows how to configure a processor for key scan operation.

The Luminary Micro LM3S8962 will be used as example, whereby its ports D-4..7 will be used as column control outputs and its ports E-0..4 as row inputs.

The individual pin defines are set in `app_hw_xxxx.h` (in this case `app_hw_lm3sxxxx.h`)

```
#define KEY_COL_OUT_1                 PORTD_BIT4
#define KEY_COL_OUT_2                 PORTD_BIT5
#define KEY_COL_OUT_3                 PORTD_BIT6
#define KEY_COL_OUT_4                 PORTD_BIT7

#define KEY_ROW_IN_1                  PORTE_BIT0
#define KEY_ROW_IN_2                  PORTE_BIT1
#define KEY_ROW_IN_3                  PORTE_BIT2
#define KEY_ROW_IN_4                  PORTE_BIT3
```

All used ports are configured as inputs with the macro

```
#define INIT_KEY_SCAN()       _CONFIG_PORT_INPUT(E, (KEY_ROW_IN_1 | KEY_ROW_IN_2 | KEY_ROW_IN_3 | KEY_ROW_IN_4)); \
                              GPIOPUR_E |= (KEY_ROW_IN_1 | KEY_ROW_IN_2 | KEY_ROW_IN_3 | KEY_ROW_IN_4); \
                              _CONFIG_PORT_INPUT(D, (KEY_COL_OUT_1 | KEY_COL_OUT_2 | KEY_COL_OUT_3 | KEY_COL_OUT_4));
```

Note that the row inputs have also activated internal pull-up resistors to avoid the need for connecting external ones (compare with figure 1).

For the column scan sequence two output macros are defined for each column (just column 0 is shown here for clarity).

```
#define DRIVE_COLUMN_1()     _DRIVE_PORT_OUTPUT_VALUE(D, KEY_COL_OUT_1, 0) // drive output low
                                                                            (column 1)
#define RELEASE_COLUMN_1()   _SETBITS(D, KEY_COL_OUT_1); _FLOAT_PORT(D, KEY_COL_OUT_1)
```

When driving a column the port is driven with a '0'. The port macro used is compatible between all µTasker projects and so other targets can be easily adapted from this example.

The release shows that the column output is momentarily set to a logic '1' to remove any charge from the line (as explained previously) before being set to the floating state. The `_FLOAT_PORT()` port macro simply sets the port to an input.

To allow the µTasker simulator to simulate the particular matrix keypad connection the following defines are also setup:

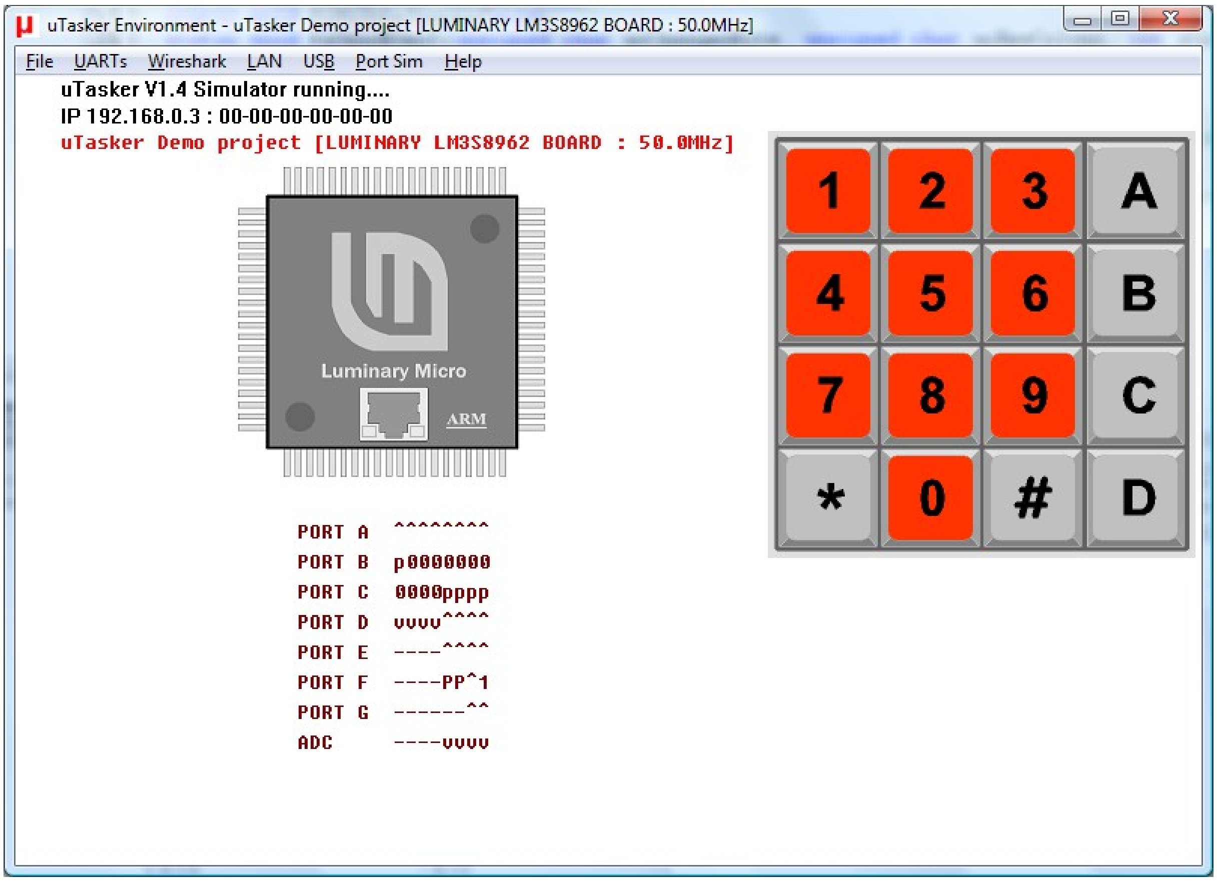
```
    #define KEY_ROW_IN_PORT_1_REF      E
    #define KEY_ROW_IN_PORT_2_REF      E
    #define KEY_ROW_IN_PORT_3_REF      E
    #define KEY_ROW_IN_PORT_4_REF      E

    #define KEY_COL_OUT_PORT_1         GPIODATA_D
    #define KEY_COL_OUT_DDR_1          GPIODIR_D
    #define KEY_COL_OUT_PORT_2         GPIODATA_D
    #define KEY_COL_OUT_DDR_2          GPIODIR_D
    #define KEY_COL_OUT_PORT_3         GPIODATA_D
    #define KEY_COL_OUT_DDR_3          GPIODIR_D
    #define KEY_COL_OUT_PORT_4         GPIODATA_D
    #define KEY_COL_OUT_DDR_4          GPIODIR_D

    #define KEY_ROW_IN_PORT_1          GPIODATA_E
    #define KEY_ROW_IN_PORT_2          GPIODATA_E
    #define KEY_ROW_IN_PORT_3          GPIODATA_E
    #define KEY_ROW_IN_PORT_4          GPIODATA_E
```

Based on this additional information, the simulator can accurately exercise the individual row inputs based on the state of the column scan outputs. See the next section for full details of working with the matrix keypad simulator in the µTasker project.

5. Keypad Simulation


uTasker Environment - uTasker Demo project [LUMINARY LM3S8962 BOARD : 50.0MHz]
File UARTs Wireshark LAN USB Port Sim Help
Luminary Micro
ARM
1
2
3
A
4
5
6
B
7
8
9
C
*
0
#
D

```
uTasker V1.4 Simulator running....
IP 192.168.0.3 : 00-00-00-00-00-00
uTasker Demo project [LUMINARY LM3S8962 BOARD : 50.0MHz]

PORT A  ^^^^^^^^
PORT B  p0000000
PORT C  0000pppp
PORT D  vvvv^^^^
PORT E  ----^^^^
PORT F  ----PP^1
PORT G  ------^^
ADC     ----vvvv
```

Figure 4. The μTasker simulator showing a 4 x 4 matrix key pad being simulated

When the key scan support is enabled a keypad will be automatically displayed when the μTasker simulator runs. The image is a bit map taken from the simulation folder called `keypad.bmp`. This bitmap can be replaced by any project specific image.

Depending on the number of rows and columns, the image is equally divided into the same number of rows and columns to create clicking areas used to identify individual keys presses using mouse clicks. This means that images used for the simulation should also be correspondingly dimensioned – possibly not exactly representing the real keypad which will be finally used in every detail.

The matrix key pad simulator built into the μTasker simulator will automatically recognise key presses and releases based on the state of the keys and the ports that they are virtually connected to.

Keys can be held down by clicking with the CTRL key pressed. Multiple keys can this be held down at the same time. A second click on the key releases the individual key and a release of the CTRL key will cause all presently pressed keys to be released. The keys that are pressed at a particular point in time is seen by a click frame on the keypad image.

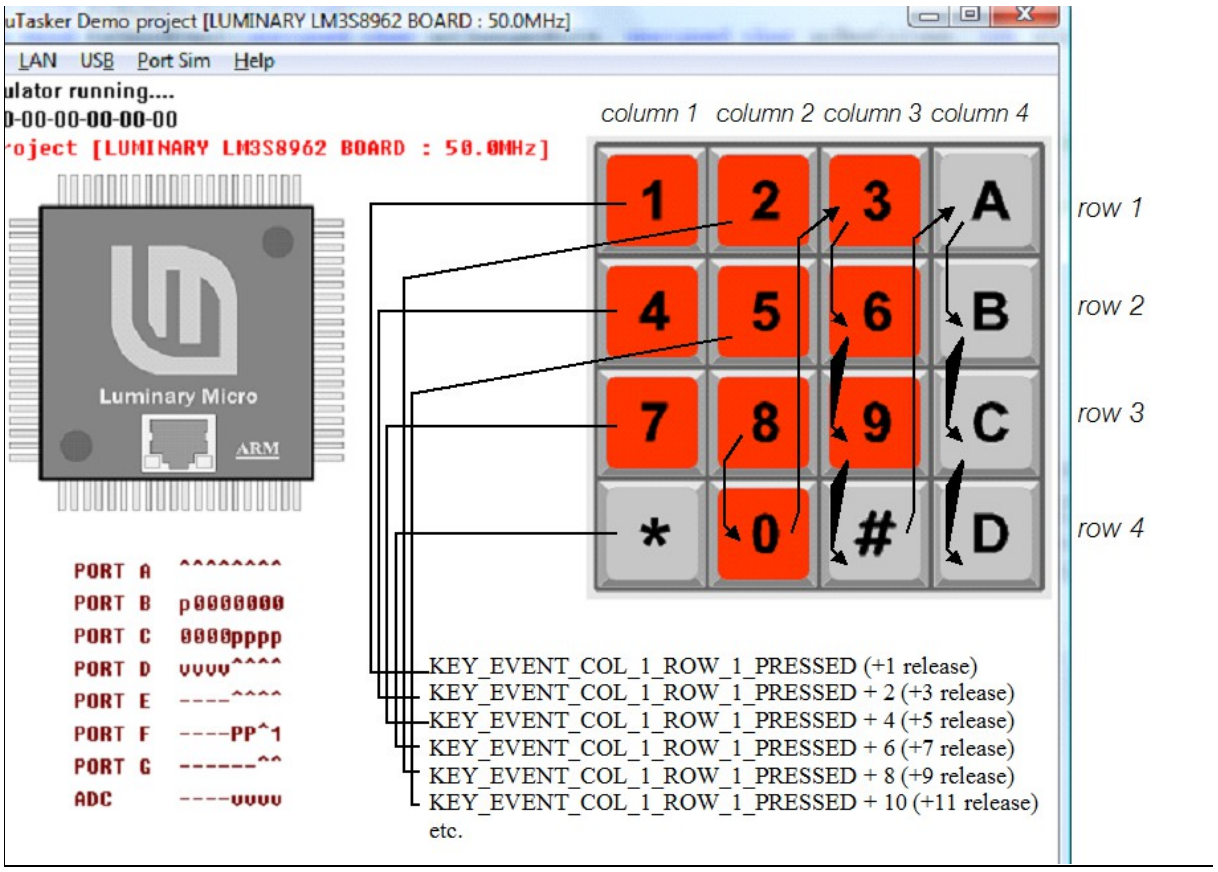
Note that mouse click releases are only recognised when the mouse is also on the released key. Beware that the mouse doesn't leave the key press field before the mouse button is released to be sure that the key release is also correctly recognised!

6. Handling Keypad Events

When the key scan task recognised a change in key state it will generate an interrupt event for the defined handling task (see `KEYPAD_PARTNER_TASK` in section 3). The event number starts with `KEY_EVENT_COL_1_ROW_1_PRESSED`, as defined in `application.h`. This first event is sent when the first key is pressed (the key connected between column 1 and row 1). When the key is pressed the next higher event is sent (`KEY_EVENT_COL_1_ROW_1_PRESSED + 1`).

The event is incremented by two for each subsequent key (one for a key press and one for a key release).

The ordering is from column 1 – row 1 to column 1 – row max, before the second column starts. The event numbering in the example 4 x 4 matrix can this be illustrated as in figure 5.



```
KEY_EVENT_COL_1_ROW_1_PRESSED (+1 release)
KEY_EVENT_COL_1_ROW_1_PRESSED + 2 (+3 release)
KEY_EVENT_COL_1_ROW_1_PRESSED + 4 (+5 release)
KEY_EVENT_COL_1_ROW_1_PRESSED + 6 (+7 release)
KEY_EVENT_COL_1_ROW_1_PRESSED + 8 (+9 release)
KEY_EVENT_COL_1_ROW_1_PRESSED + 10 (+11 release)
etc.
```

Figure 5. The µTasker simulator showing a the event numbering of the matrix keys

The demo project (see `application.c`) sends the key pressed/released to the debug output as follows when handling interrupt events.

```
if ((KEY_EVENT_COL_1_ROW_1_PRESSED <= ucInputMessage[MSG_INTERRUPT_EVENT]) &&
    (KEY_EVENT_COL_4_ROW_4_RELEASED >= ucInputMessage[MSG_INTERRUPT_EVENT])) {
    fnDebugMsg((char *)cKey[ucInputMessage[MSG_INTERRUPT_EVENT] -
                              KEY_EVENT_COL_1_ROW_1_PRESSED]); // key press or release
    break;
}
```

Where there is a string associated to each event/key in `cKey[]`. The application can use these events as the input to a state-event machine for practical user interface control. It can also measure the time that each key is pressed (for example, to enter a menu only when keys are pressed for a certain duration) or react to key presses or releases based on multiple keys being held down at the same time.

7. Configuration and work with Non-Multiplexed keypads

Not all applications require a matrix keypad and so the project can also be configured for each available key being directly connected to a dedicated port input.

The following illustrates how the previously described keypad can be modified to operate with just 5 keys connected directly to ports – for the example the LM3S8962 Evaluation board is used. This has 5 push buttons connected to ports as follows:

Button	Port
UP	PE0
LEFT	PE2
RIGHT	PE3
DOWN	PE1
SELECT	PF1

It would be possible to replace the keypad simulator bit map with a different image representing the board's keys but the original 4 x 4 keypad will be used for simplicity, in which case the push buttons (and subsequently the processor ports) are assigned to the following keypad numbers. In the fourth column the event key (from the previous section, whereby the first row/column is 1, the next button is 2, the following is 3, up to 16) has been entered to aid in assigning the simulated key to ports input during the subsequent configuration.

Button	Port	Keypad button	Event Key
UP	PE0	1	1
LEFT	PE2	2	5
RIGHT	PE3	3	9
DOWN	PE1	4	2
SELECT	PF1	5	6

In order to convert the project to using a non-matrix keypad the define `KEY_COLUMNS` should be set to 0. This implies that it is one-dimensional and so has only inputs and no scan outputs.

Furthermore, the simulator can still use a matrix type layout by assigning 'virtual' rows and columns:

```
#define VIRTUAL_KEY_ROWS           4  // virtual rows and columns when KEY_COLUMNS is zero
#define VIRTUAL_KEY_COLUMNS        4
```

The hardware configuration in `app_hw_lm3sxxxx.h` is shown below.

```
#define KEY_IN_1                    PORTE_BIT0
#define KEY_IN_2                    PORTE_BIT2
#define KEY_IN_3                    PORTE_BIT3
#define KEY_IN_4                    PORTE_BIT1
#define KEY_IN_5                    PORTF_BIT1

#define INIT_KEY_STATE              (0x1f)  // all keys are expected to be pulled up at start

#define INIT_KEY_SCAN()   _CONFIG_PORT_INPUT(E, (KEY_IN_1 | KEY_IN_2 | KEY_IN_3 | KEY_IN_4)); \
                         GPIOPUR_E |= (KEY_IN_1 | KEY_IN_2 | KEY_IN_3 | KEY_IN_4); \
                         _CONFIG_PORT_INPUT(F, (KEY_IN_5)); GPIOPUR_F |= (KEY_IN_5)

#define READ_KEY_INPUTS()   _READ_PORT_MASK(E, (KEY_IN_1 | KEY_IN_2 | KEY_IN_3 | KEY_IN_4)) |
                           (_READ_PORT_MASK(F, KEY_IN_5) << 3)

#define KEY_1_PORT_REF              E           // to allow simulator to map key pad to inputs
#define KEY_1_PORT_PIN              KEY_IN_1
#define KEY_5_PORT_REF              E
#define KEY_5_PORT_PIN              KEY_IN_2
#define KEY_9_PORT_REF              E
#define KEY_9_PORT_PIN              KEY_IN_3
#define KEY_2_PORT_REF              E
#define KEY_2_PORT_PIN              KEY_IN_4
#define KEY_6_PORT_REF              F
#define KEY_6_PORT_PIN              KEY_IN_5
```

The new define `INIT_KEY_STATE` is the expected start state of the inputs when no keys are pressed. If there is a difference key press events will immediately be sent after the software starts, allowing an already pressed key to be identified. Since there are 5 inputs the value is equivalent to 5 bits being set high.

Note however that the inputs are not all on one port but are spread over two different ports. These are ports E and port F, which are initialised as inputs with pull-ups in the macro `INIT_KEY_SCAN()`.

Rather key scan task simply reads all inputs using the macro `READ_KEY_INPUTS()`. This example shows that two masked port reads are performed and the result is 5 bits wide, with the F-port read result being shifted into the MSB bit position.

Note that the defines for mapping the simulated 4 x 4 keypad to the inputs are used exclusively by the simulator. The interrupt events sent by the key scan task to the `KEYPAD_PARTNER_TASK` are defined by the ordering of the inputs:

- `KEY_IN_1` **`0x01`** (LSB)
  event (`KEY_EVENT_COL_1_ROW_1_PRESSED`) when pressed and (`KEY_EVENT_COL_1_ROW_1_PRESSED + 1`) when released.
- `KEY_IN_4` **`0x02`**
  event (`KEY_EVENT_COL_1_ROW_1_PRESSED + 2`) when pressed and (`KEY_EVENT_COL_1_ROW_1_PRESSED + 3`) when released.

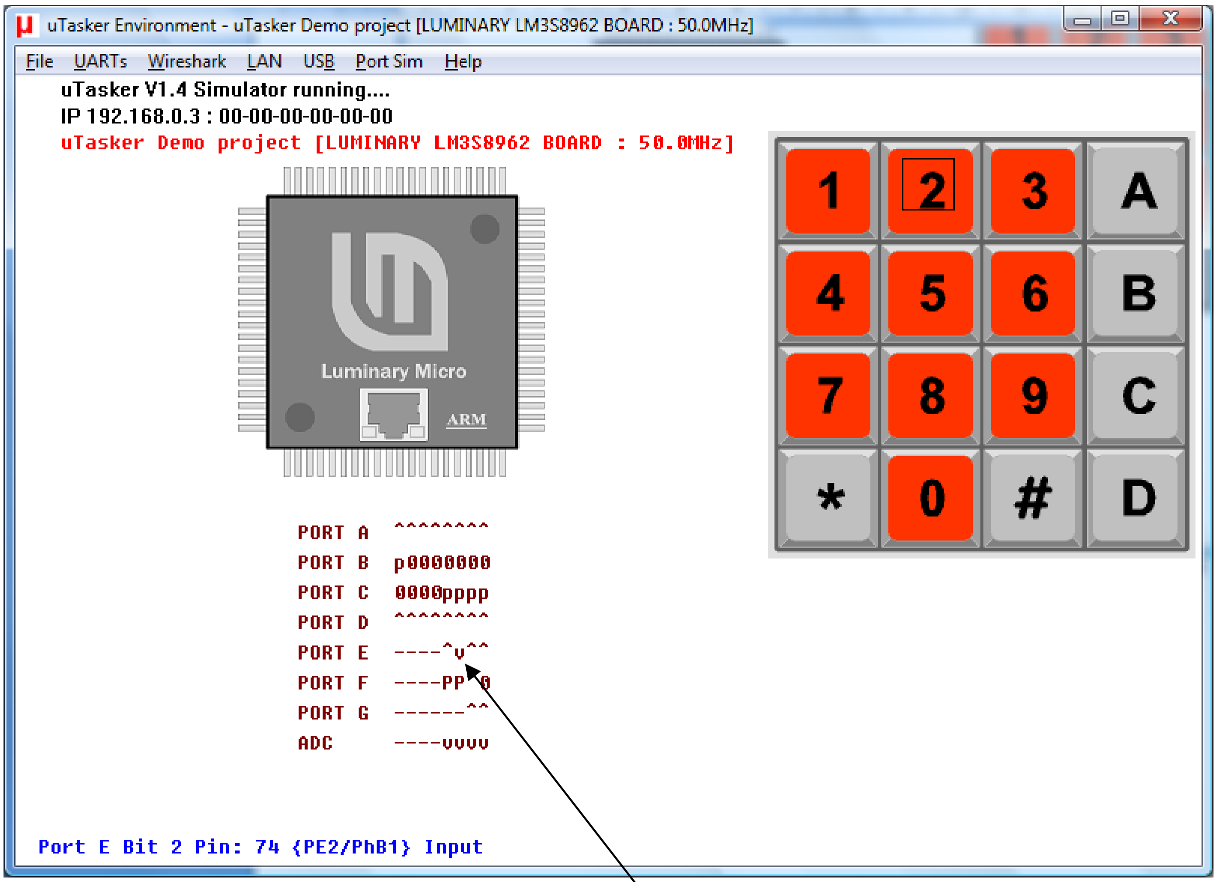
through to

- `KEY_IN_5` **`0x10`*** - event (`KEY_EVENT_COL_1_ROW_1_PRESSED + 8`) when pressed and (`KEY_EVENT_COL_1_ROW_1_PRESSED + 9`) when released.

*Note that it was shifted to this position in the read input value!

The processing at the application interface is, apart from project/configuration specific event numbering, identical to when a matrix key pad is used.

Figure 6 shows the simulator when the key '2' is pressed.


uTasker Environment - uTasker Demo project [LUMINARY LM3S8962 BOARD : 50.0MHz]
File UARTs Wireshark LAN USB Port Sim Help
Luminary Micro
ARM
1 2 3 A
4 5 6 B
7 8 9 C
* 0 # D

```
uTasker V1.4 Simulator running....
IP 192.168.0.3 : 00-00-00-00-00-00
uTasker Demo project [LUMINARY LM3S8962 BOARD : 50.0MHz]

PORT A   ^^^^^^^^
PORT B   p0000000
PORT C   0000pppp
PORT D   ^^^^^^^^
PORT E   ----^v^^
PORT F   ----PP^0
PORT G   ------^^
ADC      ----vvvv

Port E Bit 2 Pin: 74 {PE2/PhB1} Input
```

Figure 6. The μTasker simulator showing directly connected key being pressed

Notice that the port E-2 is held low, as is required since this key is indeed connected to this port input.

8. Conclusion

This document has discussed the key scan support integrated in the µTasker project. Using examples based on a 4 x4 matrix keyboard and also on 5 directly connected buttons the project configuration and simulation with the µTasker simulator have been illustrated.

The project can easily be configured to suit different processor types and also matrix key pads up to 8 x 8, or up to 32 directly connected buttons.

Modifications:

- V0.01 13.8.2009: First preliminary version
- V0.02 21.8.2009: Diode direction in figure 3 corrected
- V1.01 08.1.2018: Added appendix with notes about debouncing, press-length and double-click detection

9. Appendix A – Input Debounce, Duration and Double-Click Detection

This section discusses method of debouncing an input, detecting when it is pressed for a short or long duration and also whether it has been pressed as a double-click.

a) Polling

Polling an input means that it is sampled periodically to identify whether it is at a button pressed or button released state. It has the advantage that button debouncing becomes unnecessary as soon as the sampling period is longer than the duration of any mechanical button bouncing (typically a few ms) but has the potential disadvantage that the polling needs to be performed also when there is never any input activity. Polling requires work by the processor (including exiting from a low power state in certain designs) and so can represent an inefficiency compared to interrupt based methods. Polling however generally allows the simplest design.

b) Interrupt Driven

If the input (or all inputs) can be configured to generate inputs on state changes no polling is necessary. This allows a design to be achieved that only requires processor activity when an input state change actually requires attention. Interrupt operation can however be more complicated to achieve, especially due to the necessity to be careful due to mechanical input bouncing causing multiple interrupt events to occur.

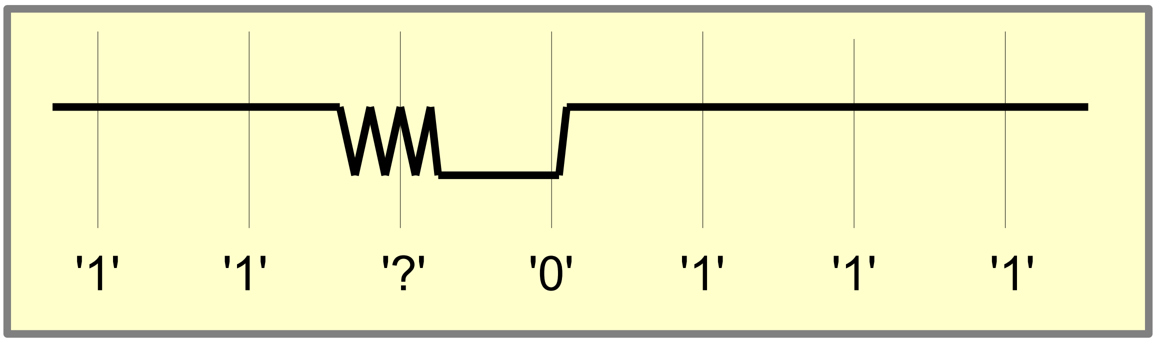
c) Interrupt Triggered Polling

A compromise between interrupt driven and polling operation can be achieved by using interrupts to only trigger the start of polling operation. Once the input remains in a stable inactive state for a certain period the polling can then be stopped and the interrupt be used to restart it at some point in the future when input changes are detected again. The advantage can be a reduction of complication but still achieve efficient (low power) operation during periods of inactivity.

Idle state: The idle state exists when the input is in it non-pressed button state, which will usually be the majority of the time. *In the following examples this is represented as a logic '1' input.*

Pressed state: When the input button is pressed it is considered in the following examples that it is detected as *a logic '0' input*.

Debouncing: When a mechanical button is pressed it is usual that the input "bounces", which causes a period where the input may change rapidly between the Idle and Pressed state before finally becoming stable in the Pressed state. When the button is released there is typically no such bouncing effect. The effect on the sampled input is shown in the following diagram:

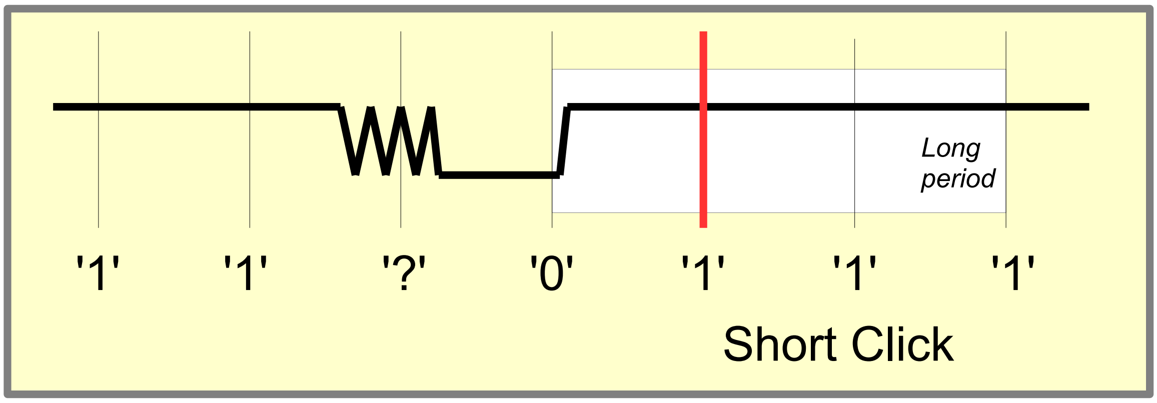


Short input press showing potential input bouncing on input button press

If the polling rate is longer than the maximum debounce time no additional debouncing logic is required in the firmware since it makes no big difference whether the initial sample is detected as a '1' or a '0'; in both cases a short press will be detected.

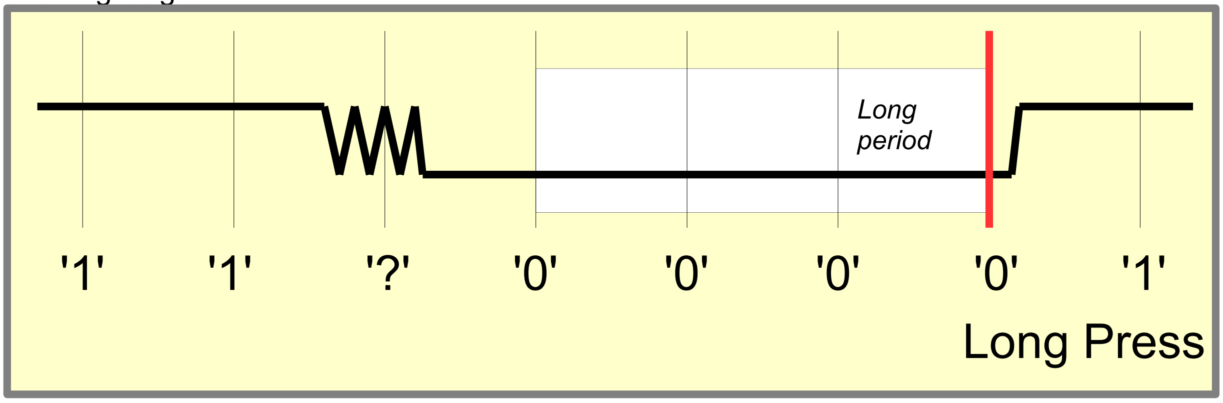
If the polling rate is fast enough that transitions are detected during the button press it becomes necessary to require the input to be detected in the Pressed state during a *defined number of successive samples*; only when this number of samples is detected in a stable Pressed state will the detection of a press be declared.

Short click (*without double-click detection*): A short click is valid when the Pressed state is detected for a period *shorter* than a certain duration, meaning that it is detected at the *release* of the key as shown in the following diagram:



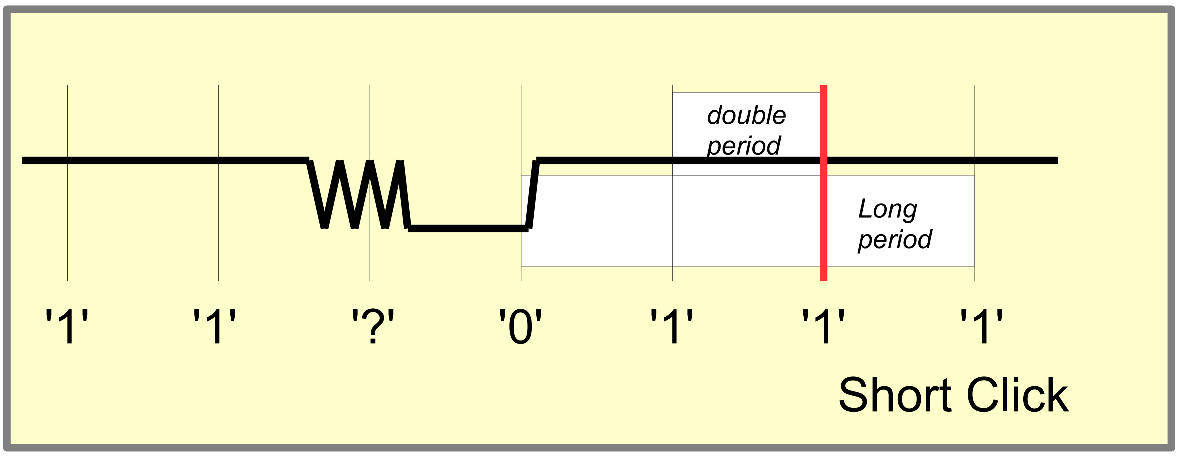
Short press detection (without double click detection)

This shows that the short click is true as long as the duration of the Pressed state is less than a certain long period duration and can be compared to the long press case as shown in the following diagram:



We notice here that the long press is detected while the input is still in the Pressed state since it becomes valid after the long period duration, irrespective of whether it will be subsequently released or not.

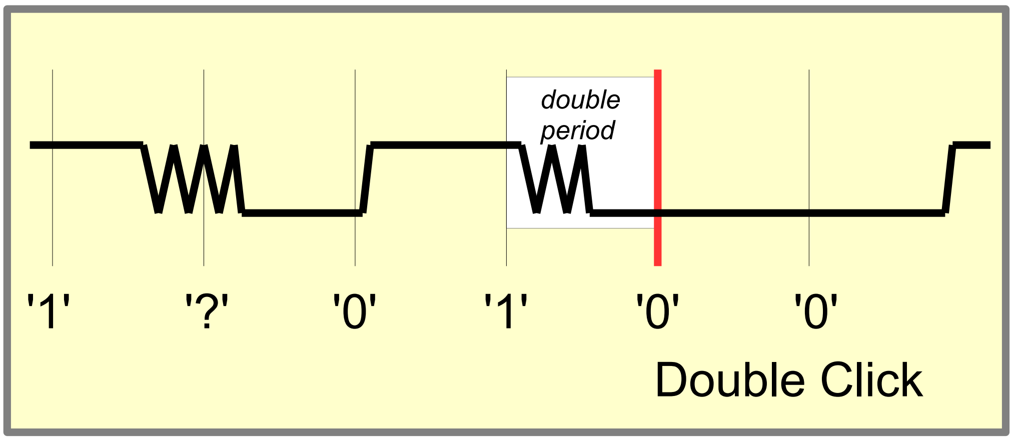
Double-click Detection:
When double click detection is required it has an influence on the short click behaviour since a single click is only valid if not followed (immediately) by a second button press short after the first input button release. This is shown in the following diagram, noting the additional period for the double-period detection:



Short press detection (with double click detection)

We note that the double period value is the number of sample periods between a short click (as would be detected without this period) and a second button press; a short press is thus only detected after this period has expired and a subsequent button press is not yet detected. This additional period results in a delay between the initial single click detected and it being reported as such.

The double-click detection case is finally shown in the next diagram, where it is seen that a double-click is a short click followed by a second click *before* the double period expires. The actual duration of the second press doesn't make any difference in this case.



Double click detection

In a design with double-click detection it is interesting to note that if the double-period were to be set to 0 it could be used to disable the double-click operation – this is because it would effectively cause the short press detection to occur at the same point in time as in the case with no double-click support.