Embedding it better...

µTasker Document

µTasker – LM3SXXXX GPIO and Simulator

Table of Contents

1. Introduction 3
2. Checking whether a specific Luminary Device is supported 3
3. Adding a new Luminary Device 5
4. Conclusion 9

1. Introduction

The Luminary Micro devices have 8-bit ports, although not all bits in a port are always physically connected to a pin.

The port bits can often be shared with a peripheral, whereby the newer TEMPEST CLASS devices (4th generation / 2009, for example the LM3S9B90) have extended port multiplexing to allow up to a maximum 16 alternative peripheral functions to be configured (the LM3S9B90 actually uses up to 11 peripherals on a single pin)!

The range of devices in the Luminary portfolio is very impressive and more family members are being introduced all of the time. This means that the µTasker project and its simulator don't necessarily support the exact type which has been chosen for a particular project; this document explains the procedure for adding a new specific part to the project so that it can be used accurately in the µTasker simulator.

2. Checking whether a specific Luminary Device is supported

The Luminary board type is selected in `config.h`, where there is a limited set of devices already fully configured:

```
...
  //#define EK_LM3S6965     // LUMINARY EVAL board with Ethernet
    #define EK_LM3S8962     // LUMINARY EVAL board with Ethernet and CAN
  //#define EK_LM3S9B95     // LUMINARY EVAL board with Ethernet, USB-OTG, CAN
```

The first thing to do is to add the new board type (which can be a custom board with a specific Luminary Micro device on it). Once the new board has been added to the list, its major settings can be defined as shown in the following example of a new board type using the LM3S2139 [with CAN interface] in 100 pin package. The fictive board is called "`MY_BOARD_1`".

```
  #define MY_BOARD_1       // board type used in the project
...
  #elif defined MY_BOARD_1 // specifics of this board and chip used
      #define DEVICE_WITHOUT_ETHERNET // disable Ethernet demo since not supported
      #define DEVICE_WITHOUT_USB      // disable USB demo since not supported
      #define TARGET_HW       "LM3S2139 CAN DEVICE"
```

The project define `MY_BOARD_1` is now used to cause the board time to be displayed when simulating and also to disable major parts of the µTasker demo project which cannot be run on it due to the fact that it doesn't have an Ethernet of USB interface.

The same project-define can now be used to control further configurations; the first being the chip type and characteristics in app_hw_lm3sxxxx.h:

```
#elif defined MY_BOARD_1
    #define _LM3S2139
    #define CRYSTAL_FREQ        8000000   // 8MHz crystal used
    #define PLL_OUTPUT_FREQ     25000000  // highest speed possible for this chip
    #define PART_CODE           CODE_LM3S2139
    #define PIN_COUNT           PIN_COUNT_100_PIN
    #define PACKAGE_TYPE        PACKAGE_LQFP
```

Here the exact Luminary Micro device is selected, the oscillator used to drive it, the CPU frequency to be used (obtained by the internal Phase-Lock-Loop) as well as package to be mounted on the board.

The define can be further used in this file and others to configure anything which is board specific, for example the port output used by the RUNLED if it is different to other board configurations which the same code may be run on.

The chip type _LM3S2139 with its chip code CODE_LM3S2139 inform the hardware part of the project, and also the µTasker simulator which parts is involved so that it can adapt itself accordingly.

If the hardware part of the µTasker project already contains the configuration of the specific device type this should be adequate for the project configuration at the hardware level and application coding and simulation test can already begin. If the hardware type is not yet available, this needs to be added as explained in the next chapter.

3. Adding a new Luminary Device

Since the related device LM3S2110 (also an early SAND-STORM device) is already available it can be used as a reference to adding the new LM3S2139. A search for the define `_LM3S2110` shows that it is used in `LM3SXXXX.h`, `LM3SXXXX_ports.c` and `WinSimMain.cpp`:

LM3SXXXX.h: This file is used to add some more specific details about the hardware capabilities of the part. First it declares the device code, whereby the following are the values for use by the new part that we are defining. Some may already exist but the rest needs to be added:

```
    #define CODE_LM3S2139           (0x84 << 16)
```

The device part number is a hard coded value which can be read from the Device Identification 1 (`DID1`) register in the chip. This value should match with the value given in the `PARTNO` field of this register according to its data sheet.

The part code is then used to specify the ports and peripherals available in the device

```
#elif PART_CODE == CODE_LM3S2139
    #define PART_DC0            (0x003f001f) // 64k FLASH, 16k SRAM
    #define PART_DC1            (0x010171bf)
    #define PART_DC2            (0x07071013)
    #define PART_DC3            (0xbf0f37c0)
    #define PART_DC4            (0x000000ff)
    #define PART_DC5            (0x00000040)
    #define PART_DC6            (0x00000000)
    #define PART_DC7            (0x00000000)
    #define PART_DC8            (0x00000000)
```

This list corresponds to the hard coded device capability registers in the chip (DC0 up to DC8). Earlier chips do not have a full set of device capability registers (for example the part in question has in fact DC0..DC4) but further register values should still be set with zeros.

The values are simply copied from the register contents according to the device's data sheet, whereby each bit, or bit combination informs about the device's capabilities (eg. DC4 is indicating that the chip has ports A to H present and bit 24 of DC1 is indicating that a CAN module is build into the chip).

This is generally adequate information of the project to configure the peripherals to suit the part.

Some GPIOs default to JTAG use but the exact details may change between devices. The default configuration after reset should be checked in the JTAG section of the part's data sheet. Because the GPIO PB7 does default to JTAG use in the LM3S2139 it is added to teh list of parts:

```
// JTAG debug
//
#if defined _LM3S6965 || defined _LM3S8962 || defined _LM3S1968 ||
          defined _LM3S10X || defined _LM3S2110 || defined _LM3S2139
    #define DEVICE_HAS_JTAG_PB7
#endif
```

LM3SXXXX_ports.c: This file is responsible for displaying the GPIO pins and peripheral functions in the simulator. Each parts needs to have a list of pins and functions (strings) but some can share settings when they are identical of very similar (with a few specific defines to tune the model). In the LM3S2139 the differences to the existing LM3S2110 are quite large and so it was decided to add a new port configuration header specifically for the new chip.

```
#elif defined _LM3S2110
    #include "LM3S2110_port.h"
#elif defined _LM3S2139
    #include "LM3S2139_port.h"
```

The content of the new `LM3S2139_port.h` (which can also be added to the simulator project if desired) can be copied from an existing similar device. The parts with only two peripheral functions are quite simple to configure and consist or two parts:

The first part is a list of the GPIO pin numbers and peripheral functions:

```
static const char *cPinNumber[PORTS_AVAILABLE][PORT_WIDTH] = {
    // PORT A (0..7)
    { "26 {PA0/U0Rx}", "27 {PA1/U0Tx}", "28 {PA2/SSI0Clk}", "29 {PA3/SSI0Fss}", "30 {PA4/SSI0Rx}", "31
{PA5/SSI0Tx}", "34 {PA6/CCP1}", "35 {PA7/CCP4}"},
    // PORT B (0..7)
    { "66 {PB0/CCP0}", "67 {PB1/CCP2}", "70 {PB2/I2C0SCL}", "71 {PB3/I2C0SDA}", "92 {PB4/C0-}", "91
{PB5/C1-}", "90 {PB6/C0+}", "89 {PB7/TRST}"},
    // PORT C (0..7)
    { "80 {PC0/TCK/SWCLK}",  "79 {PC1/TMS/SWDIO}", "78 {PC2/TDI}", "77 {PC3/TDO/SWO}", "25 {PC4/CCP5}", "24
{PC5/C1+}", "23 {PC6/C2+}", "22 {PC7/C2-}"},
    // PORT D (0..7)
    { "10 {PD0/CAN0Rx}", "11 {PD1/CAN0Tx}", "12 {PD2/U1Rx}", "13 {PD3/U1Tx}", "95 {PD4/CCP3}", "96 {PD5}",
"99 {PD6}", "100 {PD7/C0o}"},
    // PORT E (0..7)
    { "72 {PE0}", "73 {PE1}", "74 {PE2}", "75 {PE3}", "NA", "NA", "NA", "NA"},
    // PORT F (0..7)
    { "47 {PF0}", "61 {PF1}", "60 {PF2}", "59 {PF3}", "58 {PF4}", "46 {PF5}", "43 {PF6}", "42 {PF7}"},
    // PORT G (0..7)
    { "19 {PG0}", "18 {PG1}", "17 {PG2}", "16 {PG3}", "41 {PG4}", "40 {PG5}", "37 {PG6}", "36 {PG7}"},
    // PORT H (0..7)
    { "86 {PH0}", "85 {PH1}", "84 {PH2}", "83 {PH3}", "NA", "NA", "NA", "NA"},
    // ADC (0..3)
    { "1 {AIN0}", "2 {AIN1}", "5 {AIN2}", "6 {AIN3}", "NA", "NA", "NA", "NA"},
};
```

When the user hovers the mouse over the port in the simulator the corresponding string will be displayed.

The second part is a list of specific strings to be displayed when peripheral functions are actually programmed.

```
static const char *cPer[PORTS_AVAILABLE][PORT_WIDTH][2] = {
    {
        //  Q    Periph.
        {  "Q", "U0Rx"     },                                          // PORT A
        {  "Q", "U0Tx"     },
        {  "Q", "SSI0Clk"  },
        {  "Q", "SSI0Fss"  },
        {  "Q", "SSI0Rx"   },
        {  "Q", "SSI0Tx"   },
        {  "Q", "CCP1"     },
        {  "Q", "CCP4"     }
    },
...
    {
        //  Q    Periph.
        {  "Q", "CAN0Rx"   },                                          // PORT D
        {  "Q", "CAN0Tx"   },
        {  "Q", "U1Rx"     },
        {  "Q", "U1Tx"     },
        {  "Q", "CCP3"     },
        {  "Q", "invalid"  },
        {  "Q", "invalid"  },
        {  "Q", "C0o"      }
    },
...
```

Although only the entries for ports A and D are shown, there is an entry for each port in the device. Notice that any GPIOs without peripheral functions are entered as "`invalid`" so that this can be displayed in case the embedded software actually sets them incorrectly.

The details about the GPIO, its pin number and its multiplexed function(s) can be taken from the corresponding "*GPIO Pins and Alternative Functions*" take in the part's data sheet.

WinSimMain.cpp: This code is responsible for displaying ports only where they exist. By masking out ports that are not available, the port simulator only displays existing ports. This is a bit trickier to program, involving manipulating the value of `ulPortMask` for each port being displayed in the routine static void `fnDisplayPorts(HDC hdc)`. Usually only an extra define or two are required to tune it up for the device, although some debugging may be required to get it exactly right – *in this case orientation on an existing chip is useful*.

In the case of the LM3S2139, ports A, B, C, E, F and G are full width (8 bit) and so need no masking. Ports E and H are half-width (0..3 available) and so are masked with 0xf0 when they are drawn.

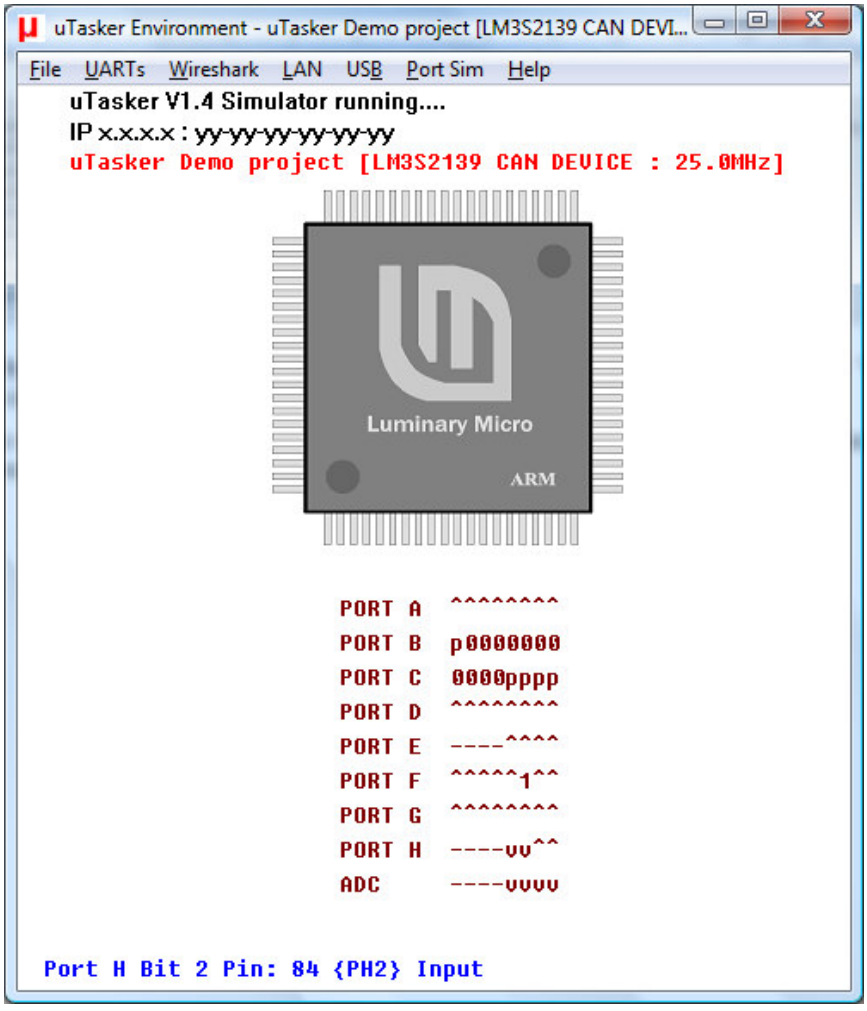
4. Configuring Port Initial Input States

In some hardware designs external pull-up or pull-down resistors are connected to the device. In order to allow the simulator to set the correct input state out of reset, respecting these external voltages or components the values can be modified in `app_hw_lm3sxxxx.h`

```
#define PORTA_DEFAULT_INPUT        0xffffffff     // initial port input states for simulator
#define PORTB_DEFAULT_INPUT        0xffffffff
#define PORTC_DEFAULT_INPUT        0xffffffff
#define PORTD_DEFAULT_INPUT        0xffffffff
#define PORTE_DEFAULT_INPUT        0xffffffff
#define PORTF_DEFAULT_INPUT        0xffffffff
#define PORTG_DEFAULT_INPUT        0xffffffff
#define PORTH_DEFAULT_INPUT        0xfffffff3
#define PORTJ_DEFAULT_INPUT        0xffffffff
```

In this example all GPIOs have pull-ups (or input voltages) equivalent to a logic '1' apart from two inputs at port H (PH2 and PH3).

The simulator finally running the newly added device, with these two inputs pulled down out of reset now looks like this:



5. Conclusion

This document is in progress and has not yet been officially released.

Modifications:

V0.01 3.8.2009: - Initial draft – work in progress. Not officially released.