LUMINARY MICRO®

Using a Stellaris® Microcontroller as an I/O Processor

APPLICATION NOTE

AN01241-02

Legal Disclaimers and Trademark Information

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Table of Contents

Introduction

Despite best efforts to bring all I/O functions on-chip, high-end embedded microprocessors often need help when interfacing to peripheral circuits. These additional interface circuits have required programmable logic devices (PLDs), discrete logic, dedicated function integrated circuits (ICs), and 8-bit microcontrollers. Stellaris® microcontrollers offer several significant advantages over these I/O solutions.

This application note covers background information on I/O processors, architectural considerations, and a practical implementation example. The design example uses a UART interface to add a PS/2 keyboard interface, I/O lines, and an audio beeper.

The Case for an I/O Processor

In an ideal world, the high-end microprocessor used in a design would have the perfect mix of on-chip peripherals. The peripherals would be available at the desired pins and there would be no system latency issues. In reality, there are many factors to complicate I/O implementation. Table 1 lists the most common reasons to consider adding an I/O processor to a system.

Table 1. I/O Processor Considerations

I/O Interfacing Problem	Description	Solved by I/O Processor?
Pin-multiplexing conflict	Two conflicting functions are needed from a single physical pin.	Yes
Low-power considerations	System has peripheral functions that must remain active while the microprocessor is in power-saving sleep mode.	Yes
Pin-count constraints	Even with high pin-count BGA packaging, I/O pins may be in short supply.	Yes
Operating-system latency	Either operating-system latency exceeds the real-time requirements of the peripheral, or responding to thousands of real-time interrupts each second places an unacceptable load on performance.	Yes
Electrical isolation	It is often impractical to individually isolate each I/O channel, especially where analog signals are involved.	Yes
Wiring constraints	I/O processors can significantly reduce inter-PCB wiring complexity.	Yes
Logic levels	Most microprocessors in this class do not have 5 V-tolerant pins.	Yes
Electrical noise	On-chip analog circuit performance can be compromised by high-speed digital switching.	Yes

Using a Stellaris microcontroller as an I/O processor can address all common I/O system problems.

I/O System Options

Table 2 examines a range of different solutions. Microcontrollers in general are the most versatile solution, with the Stellaris microcontroller providing both an economical solution and a common tool chain with the host CPU.

Table 2. Comparison of Available Solutions

Possible Solution	Digital I/O	Analog I/O	ARM Architecture	Intelligence	Cost
CPLD	Yes	No	No	No	Low
FPGA	Yes	No	No	No	Medium/High
I^2C/SPI Peripherals	Yes	Yes	No	No	Medium
MCU 8-bit	Yes	Yes	No	Yes	Low
MCU Cortex-M3	Yes	Yes	Yes	Yes	Low

Host CPU Interfaces

An important consideration is the type of interface between the I/O processor and the host microprocessor. Stellaris microcontrollers offer three types of serial interface to the host microcontroller. Table 3 lists the attributes of each serial bus.

Table 3. Serial Bus Attributes

Serial Bus	Wire Count	Typical Maximum Speed	Typical Distance	Easy to Isolate
I^2C	2	100/400 kbps	< 1M	No
SSI (SPI)	4	100 kHz – 10 MHz	< 1M	Yes
UART	2	460.8 kbps	Depends on drivers	Yes

Figure 1 shows a Stellaris microcontroller interfaced to a host microprocessor using an I^2C serial interface. I^2C has the advantage of supporting multiple slave devices with only two wires.

The Stellaris Advantage

Development and Debugging

A significant benefit of using a Stellaris microcontroller in a system containing other ARM devices is the ability to use common development tools. All microcontroller targets in a system can use the same Integrated Development Environment (IDE) and debugger hardware, which reduces development time and budget.

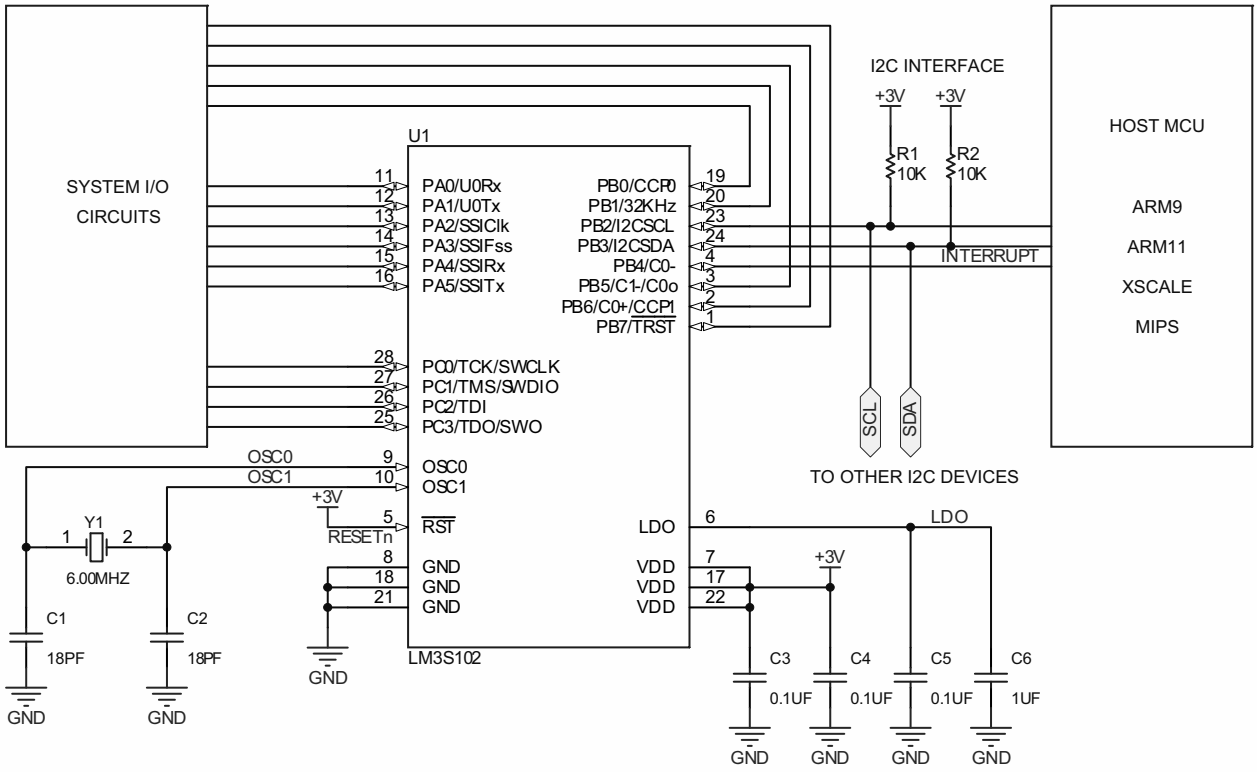
Architecture

Stellaris microcontrollers use ARM's Cortex-M3 processors—part of the ARMv7 family. Thumb-2 technology combines both 16-bit and 32-bit instructions for high-performance processing.

Compared to earlier ARM generations, Cortex-M3 provides improved interrupt-handling capabilities, which are essential in time-critical, embedded-control applications. The Cortex-M3's Nested Vectored Interrupt Controller (NVIC) reduces the number of clock cycles needed to enter an interrupt by up to 70%. I/O processing code can move quickly and efficiently between multiple prioritized interrupt sources.

For total flexibility, Stellaris microcontrollers also allow any GPIO pin to be configured as an edge- or level-sensitive interrupt.

Figure 1. Stellaris Microcontroller Interfaced to a Host Microprocessor Using an I²C Serial Interface



I/O Processor Design

The following design adds the following three interfaces to a low-cost, high-end embedded microprocessor system:

- PS/2 keyboard interface
- Eight general-purpose output pins
- Audio beeper

Interfacing a PS/2 port directly to an embedded microprocessor presents several challenges:

- The host microprocessor does not have a PS/2 interface on-chip.
- The PS/2 is a 5 V interface.
- The PS/2 keyboard clocks out data at more than 10 kHz. The format is not compatible with SPI or I^2C, so receiving this data stream either requires specialized hardware or an interrupt on each clock. This is either not achievable with most embedded operating systems, or an inefficient use of microprocessor bandwidth.

These three challenges are easily resolved by selecting a Stellaris microcontroller as an I/O processor.

A Luminary Micro Stellaris LM3S101 microcontroller can perform all three functions for about $1.00 with resources to spare for future expansion. This example communicates to the host using a UART interface and a simple ASCII-based protocol. Replacing the microcontroller with an LM3S102 device would enable I^2C communication to the host CPU.

Functional Description

The keyboard generates synchronous PS/2 clock and data signals to the LM3S101 microcontroller at 10-15 kHz. The LM3S101 microcontroller monitors these signals, clocks in the data stream, and verifies parity. The PS/2 interface is actually a bi-directional interface, but only keyboard transmit is demonstrated in this example.

Once a byte has been received and verified, the LM3S101 microcontroller software writes the data to the UART for transmission to the host microprocessor. The software could be expanded to convert the PS/2 scan codes to ASCII equivalents before they are relayed.

The entire I/O interface circuit is shown in Figure 2.

The software listing in the I/O Processor Example Source Code on page 8 uses the Stellaris family driver library, DriverLib. to simplify Stellaris peripheral accesses.

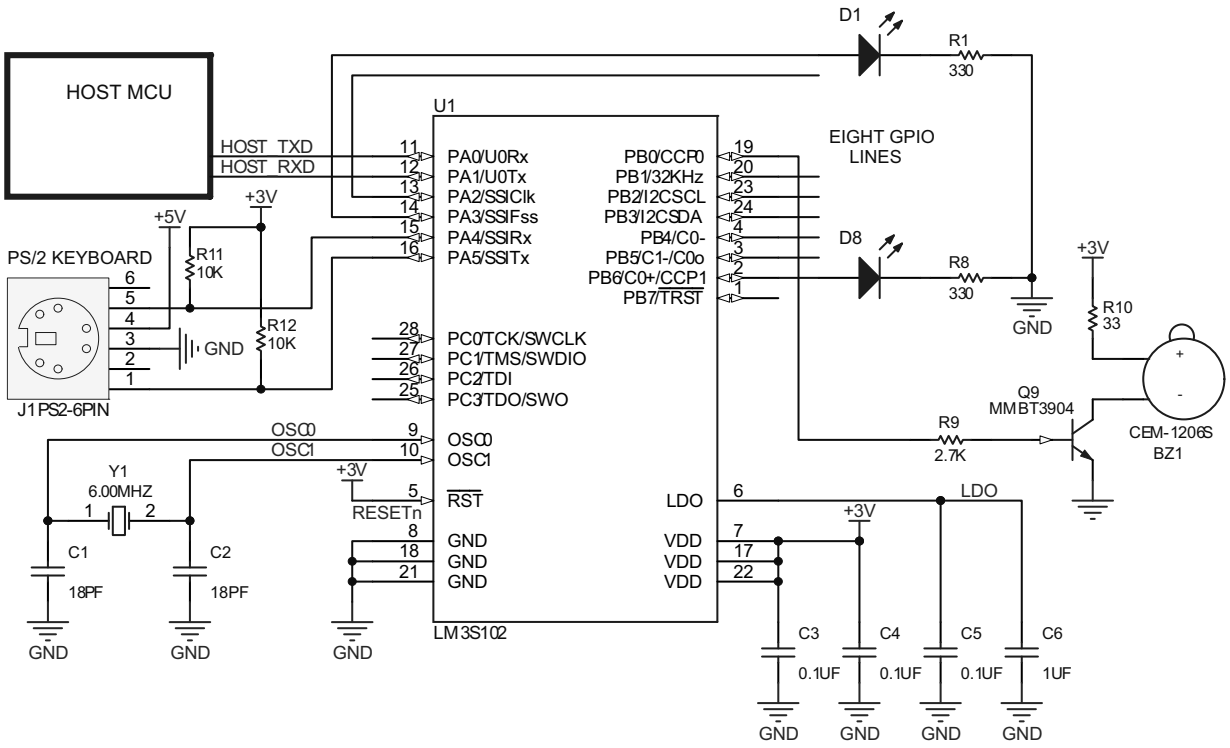
Protocol

This example uses a very simple one-byte ASCII protocol for commands from the host microcontroller:

- `0..7` control digital outputs 0..7
- `b` and `m` control the beeper

Communication to the host microprocessor consists entirely of scan code data—in this case a series of codes known as Set 2. Each physical key in the keyboard generates unique byte sequences for `make` (key down) and `break` (key released). For example, the `A` key generates 0x1C for `make`, and 0xF0 0x1C for `break`.

Figure 2. I/O Interface Circuit



I/O Processor Example Source Code

```
//*****************************************************************************
//
// AN3_main.c - Example Program for Luminary Micro Application Note 3
//              "Using Stellaris as an I/O Processor"
//
// Manages PS/2 keyboard, beeper and GP output functions for a host microprocessor
//
// Copyright (c) 2006 Luminary Micro, Inc.  All rights reserved.
//
// Software License Agreement
//
// Luminary Micro, Inc. (LMI) is supplying this software for use solely and
// exclusively on LMI's Stellaris Family of microcontroller products.
//
// The software is owned by LMI and/or its suppliers, and is protected under
// applicable copyright laws.  All rights are reserved.  Any use in violation
// of the foregoing restrictions may subject the user to criminal sanctions
// under applicable laws, as well as to civil liability for the breach of the
// terms and conditions of this license.
//
// THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
// OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
// LMI SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR
```

```
// CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
//
//*****************************************************************************

#include "../../StellarisWare/hw_memmap.h"
#include "../../StellarisWare/hw_types.h"
#include "../../StellarisWare/hw_ints.h"
#include "../../StellarisWare/driverlib/sysctl.h"
#include "../../StellarisWare/driverlib/uart.h"
#include "../../StellarisWare/driverlib/gpio.h"
#include "../../StellarisWare/driverlib/timer.h"
#include "../../StellarisWare/driverlib/interrupt.h"

//*****************************************************************************
//
// States for PS/2 receive state-machine
//
//*****************************************************************************
enum
{
     PS2_STATE_IDLE,
     PS2_STATE_DATA,
     PS2_STATE_PARITY,
     PS2_STATE_STOP,
     PS2_STATE_DONE
};

//*****************************************************************************
//
// The value of PS/2 receive state-machine
//
//*****************************************************************************
volatile unsigned char g_ucPS2State;

//*****************************************************************************
//
// PS/2 receive data value
//
//*****************************************************************************
volatile unsigned long g_ulScanCode;

//*****************************************************************************
//
// PS/2 receive parity count value
//
//*****************************************************************************
unsigned char g_ucParity;

//*****************************************************************************
//
// PS/2 data bit count value
//
//*****************************************************************************
unsigned char g_ucDataBitCount;

//*****************************************************************************
//
```

```
// Read and return the logic level of PS/2 Dat signal on PA5.
//
//*****************************************************************************
char
Ps2DatIn(void)
{
    if(GPIOPinRead(GPIO_PORTA_BASE, 0x20) == 0x20)
    {
        return(1);
    }
    else
    {
        return(0);
    }
}

//*****************************************************************************
//
// The NVIC calls this ISR every time there's a falling edge on the PS/2
// clock input.
//
//*****************************************************************************
void
PORTaISR(void)
{
    //
    // Clear the interrupt
    //
    GPIOPinIntClear(GPIO_PORTA_BASE, 0x10);

    //
    // Determine current receiver state
    //
    switch(g_ucPS2State)
    {
        case PS2_STATE_IDLE:
        {
            //
            // We were Idle, so check that the start bit is valid.
            // If it is then move to Data receive state
            //
            if(Ps2DatIn() == 0)
            {
                g_ucPS2State = PS2_STATE_DATA;
                g_ulScanCode = 0;
                g_ucParity = 0;
                g_ucDataBitCount = 0;
            }
            else
            {
                g_ucPS2State = PS2_STATE_IDLE;
            }
            break;
        }

        case PS2_STATE_DATA:
        {
```

```
            //
            // Read in a data bit, LSB first, and
            // increment parity count if it is a '1'
            //
            g_ulScanCode >>= 1;
            if(Ps2DatIn())
            {
                g_ulScanCode |= 0x80;
                g_ucParity++;
            }
            if(++g_ucDataBitCount == 8)
            {
                g_ucPS2State = PS2_STATE_PARITY;
            }
            break;
        }

        case PS2_STATE_PARITY:
        {
            //
            // If theParity bit matches move to the Stop bit state
            //
            if((g_ucParity & 0x01) == Ps2DatIn())
            {
                g_ucPS2State = PS2_STATE_IDLE;
            }
            else
            {
                g_ucPS2State = PS2_STATE_STOP;
            }
            break;
        }

        case PS2_STATE_STOP:
        {
            //
            // If the stop bit is not a '1', then fail
            //
            if(Ps2DatIn()==0)
            {
                g_ucPS2State = PS2_STATE_IDLE;
            }
            else
            {
                g_ucPS2State = PS2_STATE_DONE;
            }
            break;
        }
    }
}

//*****************************************************************************
//
// Main function for the I/O processor loop
//
//*****************************************************************************
int
```

```
main(void)
{
    unsigned char ucLedState;
    int iCode;

    //
    // Enable the peripherals used by this application
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

    //
    // Set up GPIO B[6:1] as outputs
    // Don't use B7, it is for JTAG!!
    //
    GPIODirModeSet(GPIO_PORTB_BASE, 0x7e, GPIO_DIR_MODE_OUT);

    //
    // Set up GPIO A[3:2] as outputs, and A[5:4] as PS/2 port inputs
    //
    GPIODirModeSet(GPIO_PORTA_BASE, 0x0c, GPIO_DIR_MODE_OUT);
    GPIODirModeSet(GPIO_PORTA_BASE, 0x30, GPIO_DIR_MODE_IN);

    //
    // Set MCU clock to 20 MHz
    //
    SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_6MHZ);

    //
    // Setup UART for serial communications to host MCU (115,200 baud, 8-N-1)
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTConfigSet(UART0_BASE, 115200, (UART_CONFIG_WLEN_8 |
                                       UART_CONFIG_STOP_ONE |
                                       UART_CONFIG_PAR_NONE));
    //
    // Configure Timer 0 for 50% PWM output at 600 Hz, ready to generate beep
    //
    TimerConfigure(TIMER0_BASE, TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_PWM);
    TimerLoadSet(TIMER0_BASE, TIMER_A, 0x8000);
    GPIODirModeSet(GPIO_PORTB_BASE, 0x01, GPIO_DIR_MODE_HW);
    TimerMatchSet(TIMER0_BASE, TIMER_A, 0x4000);

    //
    // Enable interrupts to the processor.
    //
    IntMasterEnable();

    //
    // Set up to interrupt on falling edge of PS/2 clock signal
    //
    IntPrioritySet(INT_GPIOA, 0x00);
    GPIOIntTypeSet(GPIO_PORTA_BASE, 0x10, GPIO_FALLING_EDGE);
    GPIOPinIntEnable(GPIO_PORTA_BASE, 0x10);
```

```
    IntEnable(INT_GPIOA);

    //
    // Start main processing loop
    //
    ucLedState=0;
    while (1)
    {
        //
        // Check for receive character from Host MCU
        //
        if(UARTCharsAvail(UART0_BASE))
        {
            iCode = UARTCharGet(UART0_BASE);

            //
            // If the received char is '0'..'7', toggle the corresponding LED
            //
            if((iCode >= '0') && (iCode <= '7'))
            {
                ucLedState ^= (1 << (iCode - 0x30));
                GPIOPinWrite(GPIO_PORTB_BASE, 0x7e, ucLedState << 1);
                GPIOPinWrite(GPIO_PORTA_BASE, 0x0c, ucLedState >> 4);
            }

            //
            // 'b' starts a beep. 'm' mutes it
            //
            if(iCode == 'b')
            {
                TimerEnable(TIMER0_BASE, TIMER_A);
            }
            if(iCode == 'm')
            {
                TimerDisable(TIMER0_BASE, TIMER_A);
            }
        }

        //
        // Check for scan code ready
        //
        if(g_ucPS2State == PS2_STATE_DONE)
        {
            UARTCharPut(UART0_BASE, g_ulScanCode);
            g_ucPS2State = PS2_STATE_IDLE;
        }
    }
    return(0);
}
```

Conclusion

A Stellaris microcontroller programmed as an I/O processor can solve difficult interfacing issues when working with high-end embedded microprocessors. A simple serial interface can support a rich set of I/O types, and preprocessing by the I/O microcontroller further reduces host microprocessor

overhead. With Stellaris, developers gain the advantage of a common tool chain, while providing an economical and effective system solution.

References

The following documents are available for download at www.luminarymicro.com:

- *LM3S101 Microcontroller Data Sheet*, Publication Number DS-LM3S101
- *Stellaris® Peripheral Driver Library User's Guide*, Document Order Number SW-DRL-UG

In addition, the following document may be useful:

- *The PS/2 Mouse/Keyboard Protocol* by Adam Chapweske, www.Computer-Engineering.org

Company Information

Luminary Micro, Inc. designs, markets, and sells ARM Cortex-M3-based microcontrollers (MCUs). Austin, Texas-based Luminary Micro is the lead partner for the Cortex-M3 processor, delivering the world's first silicon implementation of the Cortex-M3 processor. Luminary Micro's introduction of the Stellaris® family of products provides 32-bit performance for the same price as current 8- and 16-bit microcontroller designs. With entry-level pricing at $1.00 for an ARM technology-based MCU, Luminary Micro's Stellaris product line allows for standardization that eliminates future architectural upgrades or software tool changes.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext. 3