Cortex-M4/M0 Cooperating Blue River Parking Meter

Massimo Manca, February, 2012

The LPC43xx—NXP's newest microcontroller family—has so many interesting new peripherals that it's difficult for an engineer not to succumb to the temptation to try a project that uses all of them. Of course it is impossible to fully understand and utilize the LPC4350 in little more than a month—which is the time I had for this project—so I'm sure I've only scratched the surface of this NXP powerhouse. But I am very impressed by the LPC4350 and will continue to use it and learn.

I started gaining confidence with the development tools by trying some of the examples supplied with the starter kit, concentrating on the more interesting features that I will certainly use on future projects: dual-core cooperation and SPIFI. Also SCT and SGPIO are very interesting peripherals. So I decided to design a sample application to exercise the LPC4350 and its peripherals.

This sample application is a simple parking meter that works like those you find in Italy and nearby countries.

Configurations

The application is currently made to be loaded and executed from SPIFI and external flash both in debug and release mode; it isn't ready to be executed from internal ram at the moment. To automatically execute the application at boot up it is necessary to set the jumpers according to the mode you will be using.

Installation

1. Extract the zip archive into the LPC4350A_PDL directory, then you will find the Dparking subdirectory. You will probably need to edit the include path directories according to your Keil µVision 4 installation path; mine is C:\NXP\Keil, where I have the directories ARM and UV4.
2. Open M4_M0_ipc.uvmpw project group (located under Dparking\Parking\Keil) using µVision
3. Build the application in one of the supported configurations. First build the M0 application and then the M4 application. The example can be run on internal ram, external flash, or external SPIFI flash memory. After a successful build load the executable into the microcontroller's internal ram, external flash, or external SPIFI flash using the Flash submenu.

4. Run the application and observe:
    - 2 LEDs blinking: D9 is driven by the M4 core (at SysTick interrupt interval) and D10 by the M0 core every time it sends a heartbeat message to the M4.
    - The printer output through the serial port. To observe the output set the serial port to 115200 baud, 8 data bits, 1 start bit, and 1 stop bit.

Application logic

The idea is that a driver parks his car in a parking space then goes to the nearest parking meter, puts some money on it until it displays the desired expiry time; then he presses the print key, takes the ticket, and places it on the dashboard of his car.

System resources used and assignment to the cores

The Cortex-M4 core handles the application logic and manages the 4 touch keys, the four LEDs (via the PCA9502 I^2C 8-bit I7O expander), the LCD display, and the real-time clock.
The Cortex-M0 core is dedicated to printer management using the RS232 on-board interface (in the near future SCT and SGPIO will be used to interface a thermal printing head).
The cores communicates using interprocessor communication (IPC) mechanisms, minimizing the data passed from M4 to M0 and providing a simple security mechanism.

Maximizing Dual-Core performance

Having two CPUs running two different applications means that the memory resources have to be shared or divided between the two processors. To maximize the performance of each it is necessary to understand how the architecture of the LPC4350. According to the UM10503 LPC43xx preliminary user manual (Rev. 1):

> To optimize the CPU performance, the ARM Cortex-M4 has three buses for Instruction (code) (I) access, Data (D) access, and System (S) access. The I- and D-bus access memory space is located below 0x2000 0000, the S-bus accesses the memory space staring from 0x2000 0000. When instructions and data are kept in separate memories, then code and data accesses can be done in parallel in one cycle. When code and data are kept in the same memory, then instructions that load or store data may take two cycles.
>
> On the LPC43xx, the ARM Cortex-M4 host CPU is used as the top-level system controller. The LPC43xx also includes a second CPU, an ARM Cortex-M0. The ARM Cortex-M0 CPU is controlled by the host CPU. The communication between both CPUs makes use of shared memory space and interrupts.
>
> The multilayer AHB matrix enables all bus masters to access any embedded memory as well as external SPI flash memory connected to the SPIFI interface. When two or more bus masters try to access the same slave, a round robin arbitration scheme is used; each master takes turns accessing the slave in circular order. The access length is determined by the burst access length of the master. For the CPU, the burst size is 1, for GP-DMA, the burst size can be up to 8. To optimize CPU performance, low-latency code should be stored in a memory that is not accessed by other bus masters, especially masters that use a long burst size.

Unlike the M4 the M0 has a single bus to access program and data memory areas, so to maximize its performance it would access two dedicated memory regions. Of course it has to access memory regions common to the M4 to conduct interprocessor communications.
For these reasons I designed this application to use external flash connected to the LPC43xx through the SPIFI interface to contain the M4 application and the internal ram to contain and execute the M0 application. You could also try using external flash instead SPIFI flash to store the M4 application and compare the performance.

Security considerations

As a good master the M4 needs to know if the M0 is working correctly, and if not it has to execute some recovery action to resolve the problem.
To try to meet these requirements I designed this type of interprocessor communications between the M4 and M0 cores:

1. At reset the M4 starts and initializes itself and its controlled peripherals, then loads the M0 executable application in RAM memory and starts its execution;
2. When the M0 processor starts it initializes itself and its controlled peripherals and signals to the M4 that it is alive. It will signal to the M4 that is alive every 5,971 seconds as a simple sort of heartbeat;
3. The M4 monitors the M0 heartbeat, and if M0 doesn't send an heartbeat message to the M4 within 5,971 seconds the M4 will try to reset the M0;
4. The MASTER_MBX_ALIVE mailbox is used only by the M0 to send the heartbeat to the M4 so the direction is M0 → M4 (M0 is the producer, M4 the consumer).

Application organization

At reset the M4 initializes itself and the peripherals it controls. It then resets the M0 core, loads its application in the internal ram space, and waits to receive the first heartbeat message.
M4 Application Execution Flow

After the M0 is running the M4 send to it the ParkingParams_t data structure containing some parameters needed to print the tickets. The data structured is passed indirectly because the parameter passed in the IPC message is its memory address.
Then M4 initializes the MVP module and enters an infinite loop where it waits for the next interrupt and manages the associated events.

There are four sources that may interrupt M4 starting the MVP_Controller() function:

1. GPIO IRQ handler: This is fired when the user presses a touch key button. The associated action is the insertion in the event ring of the appropriate touch key code.
2. RTC 1 second interrupt: This is fired every second; it is used to refresh the actual time displayed on the LCD.
3. SysTick IRQ handler: This is fired every 1 millisecond; it is used to generate simple timings used by SysTimer module to implement simple timer and delay management. It is also used to flash the D9 led at a frequency of 1 Hz.
4. M0 IRQ handler: This is transparently handled by the IPC module included in the LPC4350_PDLA library.

All the main application logic is handled by the MvpHandler module with MVP_Handler() functions; basically it checks the event ring buffer and executes the appropriated actions.

M0 application execution flow

At reset M0 initialize the peripherals it will handle; sends the first heartbeat message to the M4; and then enters an infinite loop where it checks the received messages from the M4, checks the SLAVE_MBX_CMDS mailbox, and eventually executes the appropriate actions. It also manages the heartbeat signaling mechanism, sending the heartbeat message to the M4 using the MASTER_MBX_ALIVE mailbox.

In order to operate the M0 needs to receive two messages:

> PARAMS_TICKET_RCVD: M0 acquires the data of the ParkingParams_t data structure and makes a local copy of them.
>
> PRINT_TICKET_RCVD: M0 acquires the data sent by the M4 related to the ticket it has to print. M0 receives the memory address of a copy of the ParkingTicket_t data structure handled by M4, then uses it to print the ticket formatted by itself.

The M4 may also send a WAKEUP_SENT message to the M0 using the SLAVE_MBX_CMDS as a further security mechanism; the M0 then has to reply with the simple WAKEUP_RCVD command to signal it is working. Actually this feature isn't actively used.

Cortex-M4 and Cortex-M0 cooperation

One of the most interesting things demonstrated in this application is the cooperation between the two Cortex cores. NXP designed the LPC43xx family to have a Cortex-M4 core acting as the master CPU and a Cortex-M0 acting as a coprocessor to help the M4 in I/O and communications. The two cores communicate via a system of mailboxes realized in RAM and with the help of two interrupt vectors that are transparently managed by the IPC module included in LPC4350_PDL_A library.

The application uses two mailboxes for IPC:

> The M4 writes to the SLAVE_MBX_CMDS mailbox on the M0 side the messages related to the ticket to be printed; it also passes the data necessary to print the tickets. M0 answers to these messages by writing to the MASTER_MBX_CMDS mailbox.
>
> The M0 periodically writes to the MASTER_MBX_ALIVE mailbox a simple sort of heartbeat signal to inform M4 that it is alive. M4 monitors this message at regular intervals, and if it doesn't receive the message it resets the M0 core. M4 answers this message by writing to the SLAVE_MBX_ALIVE mailbox.

All the messages the mailbox follow the same format. The interesting thing is that the format provides a 32-bit message identifier and a 32-bit message data/parameter. So if the data sent is longer than 32 bits will be passed the memory address of the real data

User interface

I used the on-board LCD and two touch keys to build the user interface and the two remaining touch buttons to test the application: one to simulate the coin insertion and the other to simulate a car that leaves the parking area (presuming a sensor or a bar that detects it):

#	Button	Meaning	LED
1	T1	A car left the parking area	LED0
2	T2	The user inserted a coin	LED1
3	T3	The user cancelled the operation	LED2
4	T4	The user requested to print the ticket	LED3

- T4 and T3 are the keys used for the user interface; T2 and T1 are the keys used to simulate the insertion of a coin and a car that left the parking area.
- The four LEDs are used to signal with a short blink that the user pressed the corresponding button.

© 2012 NXP B.V.
All rights reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner. The information presented in this document does not form part of any quotation or contract, is believed to be accurate and reliable and may be changed without notice. No liability will be accepted by the publisher for any consequence of its use. Publication thereof does not convey nor imply any license under patent or other industrial or intellectual property rights.

Date of release: February 2012
Published in the USA