



MSP430 Launchpad 指导书

徐珺

2013.12

Contents

第一部分 第一个工程	3
第二部分 中断和计时器	10
第三部分 UART	16

第一部分 第一个工程

在这一部分中，我们将介绍 CCS 集成开发环境的一些基本使用方法，并试着编写一个简单的小程序，实现按键打开 LED

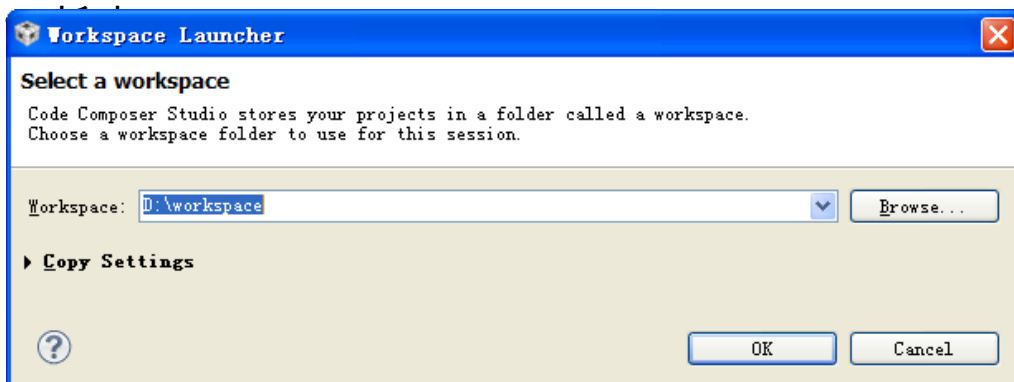


Figure 1-1 MSP430 外观

TI 的 Launchpad 板本身是完整的 MSP430 开发环境，我们所需要的仅仅是下载安装 CCS 集成开发环境（也可以用 IAR 等开发环境，这里我们选用 CCS），用 Mini-USB 线把 MSP430 连接到电脑，你就可以开始编写代码了。

现在我们就试着在 CCS 中新建一个新的工程。我们将写一个简单的小程序，实现按键打开 LED 灯的功能。

New Project



Workspace Launcher

Select a workspace

- **File > New > CCS Project**
- **Project name: ButtonLED**
- **Device>Family: MSP430**
- **Variant: MSP430G2553**
- **Project templates and examples : Empty Project (with main.c)**
- **Finish !**

New CCS Project

Create a new CCS Project.

Project name:

Output type:

☒ Use default location

Location:

Device

Family:

Variant:

Connection:

▶ Advanced settings

▼ Project templates and examples

type filter text

☐ Empty Projects

- ☐ Empty Project
- ☒ Empty Project (with main.c)
- ☐ Empty Assembly-only Project
- ☐ Empty RTSC Project
- ☐ Empty Grace (MSP430) Project

☐ Basic Examples

- ☐ Blink The LED

Creates an empty project fully initialized for the selected device. The project will contain an empty 'main.c' source-file.

3. 可以编写自己的程序了!

```
#include "msp430g2553.h" //Contains definitions
for registers and built-in functions
```

我们的第一行代码：

这个头文件包含了 MSP430G2553 的寄存器定义和内置函数，如果你使用的是其他型号的芯片，例如 MSP430G2231，你需要包含的头文件便是"msp430g2231.h"

接着，是主函数部分。

```
void main(void) // Main program
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P1DIR |= BIT0             // Set P1.0 to output
direction
    P1OUT &= ~BIT0; // set P1.0 to Off
    P1REN |= BIT3; // P1.3 Pull up Enabled
    P1IES |= BIT3; // P1.3 Interrupt at Falling Edge
    P1IFG &= ~BIT3; // P1.3 Interrupt Flag Clear
    P1IE |= BIT3; // P1.3 interrupt Enabled
    _EINT(); // Enable all interrupts
    while(1) //Loop forever, we'll do our job in the
interrupt routine...
    {}
}
```

这仅仅是一个初始化过程，我们还没有写有关开关 LED 灯的代码。

```
WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
```

这一行中，我们关闭了看门狗计时器——它的主要功能是在单片机运行了一段时间后重置单片机，以防止程序陷入无限死循环。

在大多数例程中，你会发现在初始化过程中都会关闭看门狗，以避免不必要的重置。

```
P1DIR |= BIT0;      // Set P1.0 to output direction
```

这一行中，我们将 P1.0 引脚设置为输出 (P1.0 与单片机上 Led1 相连)。在 MSP430 中，P1DIR 是一个控制 Port1 引脚方向的八位寄存器。如果你把某一位配置为 0，则它作输入用，反之是输出。BIT0 是一个系统定义的常量，即 0x01。

```
P1OUT &= ~BIT0; // set P1.0 to Off
```

这一行中，我们将 P1.0 引脚默认输出配置为低，这样在系统上电的时候 LED 灯一定是灭的。P1OUT 寄存器控制着 PORT1 上所有配置为输出的引脚，如果你把对应位配置为 1 (通过将 P1OUT 与上对应的常数如 BIT0，BIT1 等)，那么对应位的输出就是高电平，反之输出低电平。

```
P1REN |= BIT3;  // P1.3 Pull up Enabled
P1IES  |= BIT3;  // P1.3 Interrupt at Falling Edge
P1IFG &= ~BIT3; // P1.3 Interrupt Flag Clear
P1IE   |= BIT3;  // P1.3 interrupt Enabled
```

这几行中，我们配置了 P1.3 的中断功能，其中 P1.3 与板上的 S2 按键相连。什么是中断？我们希望单片机知道我们什么时候按下了按键，无论单片机当前所执行的任务是什么，这就需要使用中断功能。中断的具体内容后面会详细介绍，这里只简述上面所执行的内容：

```
P1REN |= BIT3  为P1.3配置了上拉电阻
P1IES  |= BIT3; 配置P1.3为下降沿有效
P1IFG &= ~BIT3 清除P1.3的中断标志位
P1IE   |= BIT3   P1.3中断使能
__EINT(); // Enable all interrupts
```

这一行是一个内置函数，它打开的所有中断功能。

```
while(1) //Loop forever, we'll do our job in the
interrupt routine...
{}
```

在进行完简单的配置后，我们终于进入了循环。注意到这是一个无限空循环，接下来的开关灯的工作就全交给中断函数了。

```
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= BIT0; //Toggle P1.0
    P1IFG &= ~BIT3; // P1.3 Interrupt Flag Clear
}
```

这是PORT1的中断处理函数，每当P1.3引脚从高电平变为低电平时（Launchpad上的按键是低电平有效的，这也是为什么配置时是下降沿有效触发中断），中断处理函数便会被调用。

```
P1OUT ^= BIT0; //Toggle P1.0
```


这一行用异或操作翻转 LED 灯的状态。


```
P1IFG &= ~BIT3; // P1.3 Interrupt Flag Clear
```

这一行用来清除中断标志位，从而下次触发中断依然有效。

如果一切顺利的话，现在我们可以把我们的程序下载到单片机运行了。

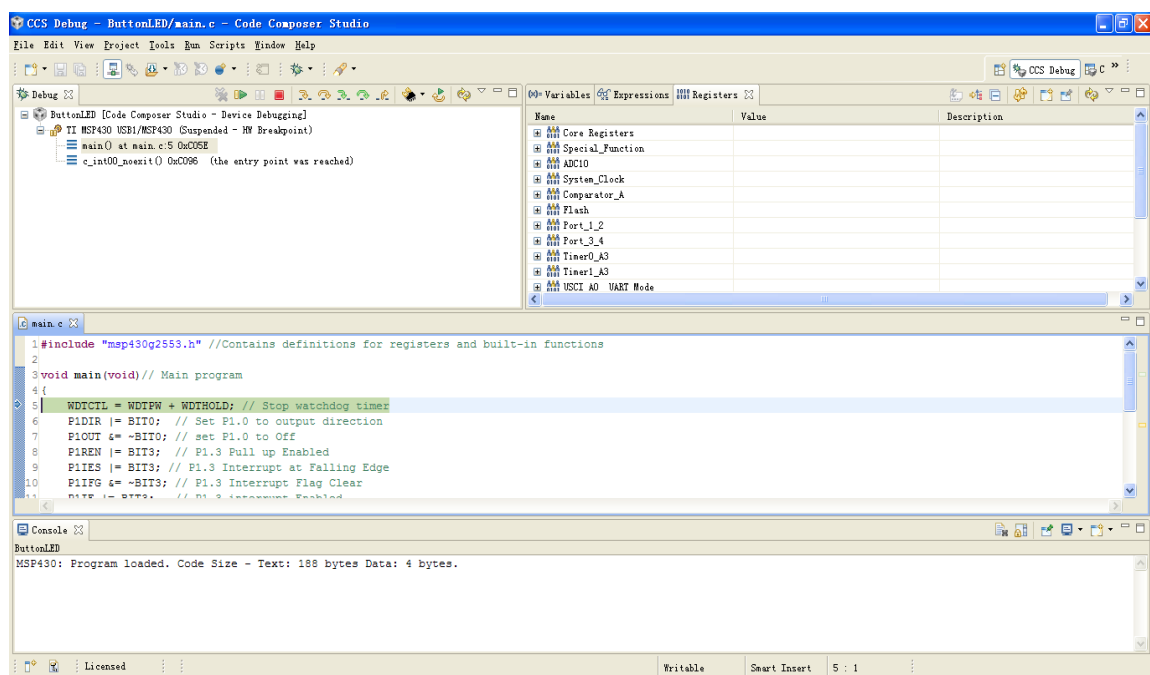
4. 将程序下载到单片机

 这个是编译按钮，如果编译顺利通过的话就可以下载到单片机了。

 这个是 Debug 按钮，编译通过之后，我们按这个按钮进入 DEBUG 模式。

 按红色按钮终止 DEBUG 模式。

至此，我们的开关 LED 程序已经下载到单片机中，如果你观察到红色的 LED 灯会随着按键亮灭，那么恭喜你，你已经完成了第一个 MSP430 的项目。



这是 *DEBUG* 模式的界面状态，我们可以单步调试程序以观察没运行一步单片机以及每一个寄存器、每一个变量的状态，以调试程序，检查错误等。

```
#include "msp430g2553.h" //Contains definitions
for registers and built-in functions

void main(void) // Main program
{
    WDTCTL = WDTPW + WDTCTL; // Stop watchdog timer
    P1DIR |= BIT0;           // Set P1.0 to output
    direction
    P1OUT &= ~BIT0; // set P1.0 to Off
    P1REN |= BIT3; // P1.3 Pull up Enabled
    P1IES |= BIT3; // P1.3 Interrupt at Falling Edge
    P1IFG &= ~BIT3; // P1.3 Interrupt Flag Clear
    P1IE |= BIT3; // P1.3 interrupt Enabled
    _EINT(); // Enable all interrupts
    while(1) //Loop forever, we'll do our job in the
    interrupt routine...
    {}
}

#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= BIT0; //Toggle P1.0
    P1IFG &= ~BIT3; // P1.3 Interrupt Flag Clear
}
```

以下是这一节的完整代码：

在这一部分中我们将会初步了解到中断的概念及其作用，我们会尝试使用计时器中断和 I/O 中断操作 LED 灯，让我们开

第二部分 中断和计时器

什么是中断？我们可以将它理解为一个约定的信号，来告知单片机特定的事件发生了，引起程序从正常运行的主函数中断开，转而执行中断处理程序，处理特定的事件。

中断是一个非常重要的概念，它可以让处理器免于执行冗余的轮询操作等待特定的外部事件的发生。在 MSP430 的架构中，有许多种类的中断：计时器中断，I/O 中断，ADC 中断等等。每一种中断在使用前都要使能和配置，每一种中断又分别有中断处理程序（Service Routine）。

下面就让我们尝试写一个小程序，实现使用计时器中断和 I/O 中断操作 LED 灯。

New Project

```
#include "msp430g2553.h"
void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;    // Stop WDT
```

按照惯例，首先包含 g2553 的头文件，关闭看门狗。每次写程序的时候你总会用到它们。

```
CCTL0 = CCIE;                // CCR0 interrupt enabled
TACTL = TASSEL_2 + MC_1 + ID_3; // SMCLK/8, upmode
TACCR0 = 10000;                // 12.5 Hz
```

这几行简单配置了计时器中断。

```
CCTL0 = CCIE; // CCR0 interrupt enabled
```

我们首先通过置 CCTL0 (Timer_A capture/compare control 0) 寄存器的 CCIE 位 (Capture/compare interrupt enable) 使能了计时器中断。

```
TACTL = TASSEL_2 + MC_1 + ID_3; // SMCLK/8, up mode
```

然后我们通过 TACTL (Timer_A control) 寄存器配置了计时器的时钟。如果查阅一下 MSP430 的手册，你会看到之后几位分别表示什么含义：

TASSEL_2 选择了 SMCLK 时钟 (由内部 DCO 支持，默认频率大约为 1MHz);

MC_1 选择了上升模式 (up mode)，即计时器计数的时候由小至大，计数上限由 TACCR0 (Timer_A capture/compare 0) 寄存器决定，由此就有

```
TACCR0 = 10000; // 12.5 Hz
```

你一定猜到了配置的结果 12.5Hz 是怎么得来的了吧，
 $1\text{M}/8/10000=12.5\text{Hz}$ ，这就是产生计时器中断的频率。

通过选择不同的时钟源，不同的时钟分频，不同的计数上限，你几乎可以配置出任何你想要的频率，需要注意的是，MSP430 的寄存器都是 16 位的，所以 TACCR0 的上限是 65535。

我们继续完善我们的程序。

```
P1OUT &= 0x00; // Shut down everything
P1DIR &= 0x00;
P1DIR |= BIT0 + BIT6; // P1.0 and P1.6 pins output
P1REN |= BIT3; // Enable internal pull-up/down
resistors
P1OUT |= BIT3; // Select pull-up mode for P1.3
```

这几行代码我们应该已经熟悉了。我们首先清空了 PORT1 的输出寄存器和方向寄存器，然后配置板上两个 LED 所对应引脚为输出，为按键对应的引脚配置上拉电阻。

```
P1IE |= BIT3;           // P1.3 interrupt enabled
P1IES |= BIT3;          // P1.3 Falling edge
P1IFG &= ~BIT3;         // P1.3 IFG cleared
```

这几行代码中，我们首先使能了 P1.3 引脚的中断功能，然后我们选择了下降沿触发中断（高电平到低电平触发），Launchpad 上的按键在不按下的时候连接着 VCC，而按下的时候连接的是 GND，因此我们选择下降沿触发。最后，我们要清除相应的中断标志位。中断标志位通知单片机一个中断的产生，因此在每次中断处理程序

```
_EINT();                // Enable all interrupts
while(1)                // Loop forever, we work with interrupts!
{ }
```

结束后，如果我们希望下次产生事件的时候依然有中断，我们应当清除中断标志位。

打开所有中断，轻松加愉快。接下来就是中断的事儿了。

```
// Timer A0 interrupt service routine
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0 (void)
{
    P1OUT ^= BIT0;           // Toggle P1.0
}
```

这是 TimerA 的中断处理程序。每当计数器溢出的时候，中断触发，程序便会执行这段代码，翻转 P1.0 的输出，对应地，LED1 会出现闪烁的效果。每次翻转之后，程序便会回到触发中断的地方，

```
// Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= BIT6;           // Toggle P1.6
    P1IFG &= ~BIT3;         // P1.3 IFG cleared
}
```

在本例中，回到 While (1)。

这是PORT1的中断处理程序，每当我们按下P1.3对应的按键时，中断触发，程序便会执行这段代码，效果如同在前一节中演示的一样。

烧代码看效果吧！

以下是这一节的完整代码：

```
#include "msp430g2553.h"

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // Stop WDT
    CCTL0 = CCIE;                       // CCR0 interrupt enabled
    TACTL = TASSEL_2 + MC_1 + ID_3;
    // SMCLK/8, upmode
    TACCR0 = 10000;                      // 12.5 Hz
    P1OUT &= 0x00;                       // Shut down everything
    P1DIR &= 0x00;
    P1DIR |= BIT0 + BIT6;
    // P1.0 and P1.6 pins output
    P1REN |= BIT3;
    // Enable internal pull-up/down resistors
    P1OUT |= BIT3;
    // Select pull-up mode for P1.3
    P1IE |= BIT3;                       // P1.3 interrupt enabled
    P1IES |= BIT3;                      // P1.3 Falling edge
    P1IFG &= ~BIT3;                     // P1.3 IFG cleared
    _EINT();                            // Enable all interrupts
    while(1)
        // Loop forever, we work with interrupts!
    {}
}

// Timer A0 interrupt service routine

#pragma vector=TIMER0_A0_VECTOR

__interrupt void Timer_A0 (void)
{
    P1OUT ^= BIT0;                     // Toggle P1.0
}
```

```
                                // Port 1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
    P1OUT ^= BIT6;                // Toggle P1.6
    P1IFG &= ~BIT3;              // P1.3 IFG cleared
}
```

这一节我们将会看到如何在 MSP430 Launchpad 上使用 UART 进行通信。我们的程序将会实现通过 UART 从 PC 读入一个字节的数据，然后发送相应的反馈字节给 PC。通信模式为 115200 波特率，全双工，8 位数据，无奇偶校验，1 位停止

第三部分 UART

MSP430 G2553 (Launchpad 使用的芯片) 是 MSP430 家族中比较给力的芯片，它集成了硬件 UART 模块。

UART 通信在处理/调试传感器的时候尤为有用，举一个简单的例子，我们可以用温度传感器采集数据，经过 AD 转换之后通过 UART 发送给 PC，传送的方式可以有有线，也可以使用无线例如蓝牙。

让我们直接进入正题吧。

New Project

首先，你应该已经熟悉了这样的开头：

```
#include "msp430g2553.h"

#define TXLED BIT0
#define RXLED BIT6
#define TXD BIT2
#define RXD BIT1

const char string[] = { "Hello World\n" };
unsigned int i; //Counter
```

像前两节中所做的那样，程序开头包含 MSP430G2553 的头文件，然后宏定义一些常数使得程序更具可读性。

接着我们定义了一个 char 类型数组（即一个 C String），存了我们将要反馈给 PC 的信息（经典的 Hello World）。最后定义一个计数变量，辅助反馈字符串的发送。

```
int main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop WDT
    DCOCTL = 0; // Select lowest DCOx and MODx
settings
    BCSC1L1 = CALBC1_1MHZ; // Set DCO
    DCOCTL = CALDCO_1MHZ;
```

这依然是大家熟悉的 main 函数的开始方法，首先关闭看门狗定时器，之后的三行将单片机的内部时钟设置为 1MHz。接下来

```
P2DIR |= 0xFF; // All P2.x outputs
P2OUT &= 0x00; // All P2.x reset
P1SEL |= RXD + TXD ; // P1.1 = RXD, P1.2=TXD
P1SEL2 |= RXD + TXD ; // P1.1 = RXD, P1.2=TXD
P1DIR |= RXLED + TXLED;
P1OUT &= 0x00;
```

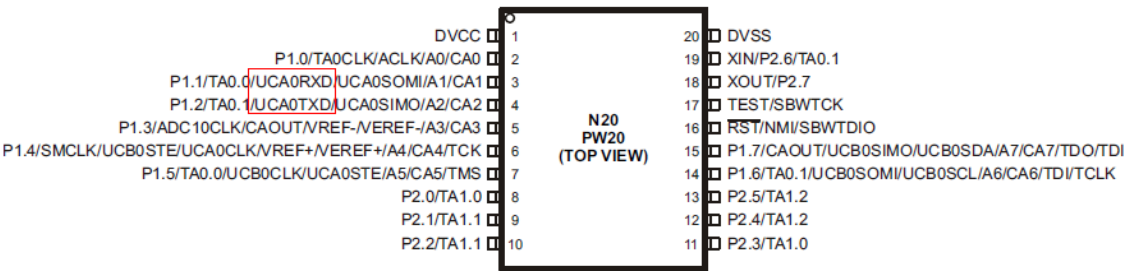
UART 和板上所有其他外设都将使用 SMCLK 时钟 (sub-main clock)。

这里是配置输入输出引脚：

前两行我们把 PORT2 的所有引脚关闭。对于用不到的引脚，关闭引脚是一个比较好的习惯，这样能有效地减少噪声和电流的消耗。

第 3 行和第 4 行将引脚 P1.1 和 P1.2 设置为 UART 模式。事实上，P1SEL 和 P1SEL2 两个寄存器是一个多路选择器，它们将 P1 的引脚连接至不同的板载的外设上。但是注意，TXD 和 RXD 是固定引脚的，我们可以通过查 G2553 的数据手册找到对应的引脚。

Device Pinout, MSP430G2x13 and MSP430G2x53, 20-Pin Devices, TSSOP and PDIP



最后两行是初始化板载 LED。

```
UCA0CTL1 |= UCSSEL_2; // SMCLK
UCA0BR0 = 0x08; // 1MHz 115200
UCA0BR1 = 0x00; // 1MHz 115200
UCA0MCTL = UCBRS2 + UCBRS0; // Modulation UCBRSx = 5
UCA0CTL1 &= ~UCSWRST;
// **Initialize USCI state machine**
UC0IE |= UCA0RXIE; // Enable USCI_A0 RX interrupt
EINT();
while (1)
{ }
}
```

这是 UART 的配置。第 1 行，如同前面所说的，我们选择 SMCLK 作为 UART 模块的时钟源，用来产生需要的波特率（当然，你也可以选择其他的时钟源）。

UCA0BR0 和 UCA0BR1 用来选择波特率：你可以将这两个寄存器存储的整数看做对时钟 SMCLK（1MHz）的分频。在我们的配置下，产生的频率是 $1\text{MHz}/8=125000\text{Hz}$ ，而实际上我们需要 115200 的波特率，所以在长时间的工作中会累积一定的误差。如果配置为 9 的话波特率又会低于 115200。这时候就需要用到 UCA0MCTL 寄存器。

这个寄存器是起调节作用的，它会选择 8 和 9 之间的分频因子，在通信中可以控制累积误差。如果分频因子为 8，有 $125000-115200=9600$ （+8.5%）的误差，如果分频因子为 9，有 $115200-111111=4089$ （-3.6%）的误差。

调节器差不多会这样工作：

位数	分频因子	误差（%）	累积误差（%）
1	8	+8.5	+8.5
2	9	-3.6	+4.9
3	9	-3.6	+1.3
4	9	-3.6	-2.3
5	8	+8.5	+6.2

.....

在 MSP430 的编程手册 (User Guide) 上可以找到一张表，列出了 SMCLK 典型频率和常用波特率下，UART 收发数据时期望的最小和最大的误差率。根据表格我们选择 5 作为调节值。

然后我们使能所有中断，进入循环。

```
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
    P1OUT |= RXLED;
    if (UCA0RXBUF == 'a') // 'a' received?
    {
        i = 0;
        UCA0IE |= UCA0TXIE;
        // Enable USCI_A0 TX interrupt
        UCA0TXBUF = string[i++];
    }
    P1OUT &= ~RXLED;
}
```

这是当 UART 接收到数据时的中断处理程序。如果你看过了指导书的第二部分，你应该对中断的概率比较熟悉了。

在中断处理程序中，我们先点亮一个 LED，表示单片机已经接收到一个字节。然后我们读 UCA0RXBUF 寄存器，这个寄存器存储了收到的数据。

如果收到的数据为'a'，那么就重置计数变量 i，打开发送中断向 PC 发送反馈字符串。

当我们把字符串的第一个字符放入 UCA0TXBUF 时，这个寄存器就会做好准备将数据发送出去。

在这个中断处理程序的最后，记得关闭 LED，表示接收中断

```
#pragma vector=USCIAB0TX_VECTOR
__interrupt void USCI0TX_ISR(void)
{
    P1OUT |= TXLED;
    UCA0TXBUF = string[i++]; // TX next character
    if (i == sizeof(string) - 1) // TX over?
        UC0IE &= ~UCA0TXIE; // Disable USCI_A0
    TX interrupt
    P1OUT &= ~TXLED;
}
```

处理完毕。

这是UART的发送中断处理程序。和刚才一样，我们点亮一个LED表示进入发送中断。

我们将要发送的下一个字符放入缓存区中。如果字符串没有结束，关闭LED，中断处理程序结束。待当前字节发送结束后，发送中断会被再一次触发（因为buffer中还有待发送的数据）。如果反馈字符串已经全部载入buffer中，我们就可以关闭发送中断，回到主程序中。

烧代码看效果吧！

提示：通信时PC与单片机的互动可以在PC端下载一个串口助手，比较经典是sscom。串口助手是调试单片机程序的利器！

以下是这一节的完整代码：

```
#include "msp430g2553.h"

#define TXLED BIT0
#define RXLED BIT6
#define TXD BIT2
#define RXD BIT1

const char string[] = { "Hello World\n" };
unsigned int i; //Counter

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop WDT
    DCOCTL = 0; // Select lowest DCOx and MODx settings
    BCSCTL1 = CALBC1_1MHZ; // Set DCO
    DCOCTL = CALDCO_1MHZ;
    P2DIR |= 0xFF; // All P2.x outputs
    P2OUT &= 0x00; // All P2.x reset
    P1SEL |= RXD + TXD ; // P1.1 = RXD, P1.2=TXD
    P1SEL2 |= RXD + TXD ; // P1.1 = RXD, P1.2=TXD
    P1DIR |= RXLED + TXLED;
    P1OUT &= 0x00;
    UCA0CTL1 |= UCSSEL_2; // SMCLK
    UCA0BR0 = 0x08; // 1MHz 115200
    UCA0BR1 = 0x00; // 1MHz 115200
    UCA0MCTL = UCBRS2 + UCBRS0; // Modulation UCBRSx = 5
    UCA0CTL1 &= ~UCSWRST;
    // **Initialize USCI state machine**
    UCA0IE |= UCA0RXIE; // Enable USCI_A0 RX interrupt
    EINT();
    while (1)
    { }
}
```

```
#pragma vector=USCIAB0TX_VECTOR
__interrupt void USCI0TX_ISR(void)
{
    P1OUT |= TXLED;
    UCA0TXBUF = string[i++]; // TX next character
    if (i == sizeof(string) - 1) // TX over?
        UC0IE &= ~UCA0TXIE; // Disable USCI_A0 TX
interrupt
    P1OUT &= ~TXLED;
}

#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
    P1OUT |= RXLED;
    if (UCA0RXBUF == 'a') // 'a' received?
    {
        i = 0;
        UC0IE |= UCA0TXIE; // Enable USCI_A0 TX
interrupt
        UCA0TXBUF = string[i++];
    }
    P1OUT &= ~RXLED;
}
```