EE 3171 Lecture 15

Serial Communications

Serial I/O Overview

Parallel vs. Serial Ports

Simplex vs. Duplex Communication

Error Control and Parity

Signaling Standards

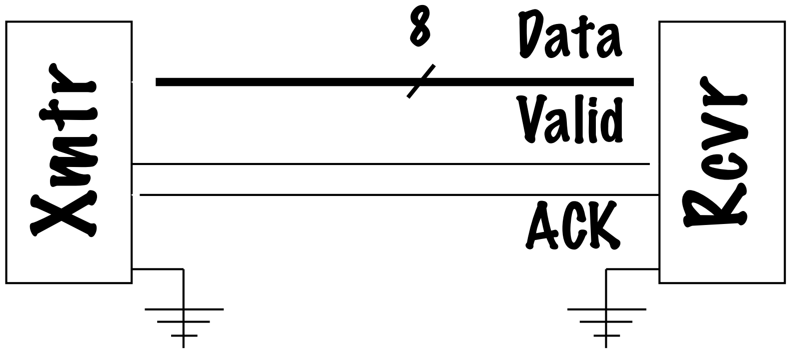
Data Formatting

Message Synchronization

Message Errors

Parallel I/O Ports

- I/O Ports so far have been parallel
  - Transmits or receives multiple bits at once
    - one pin per bit ⇒ one wire per bit
    - Control = strobe signals, e.g.
    - Data valid (ready)
    - Data received (acknowledge)



Parallel I/O Ports

Advantages

- Maximum throughput (bits in parallel)
- Good for short-range, high-data rate busses
    - e.g. disk or printer interface

Disadvantages

- Need many wires (one per bit + strobes)
    - Cabling gets expensive
    - Cabling gets clumsy (e.g. ribbon cable)
- Some long-distance media are inherently single-line
    - phone-lines or wireless frequencies

Advantages of Serial Communication

As little as one wire between transmitter and receiver

- Therefore, cabling is cheaper and more flexible (literally).

- Obviously more appropriate for inherently serial media.

Good for longer range and/or lower data rate communications

- e.g. remote terminal or modem

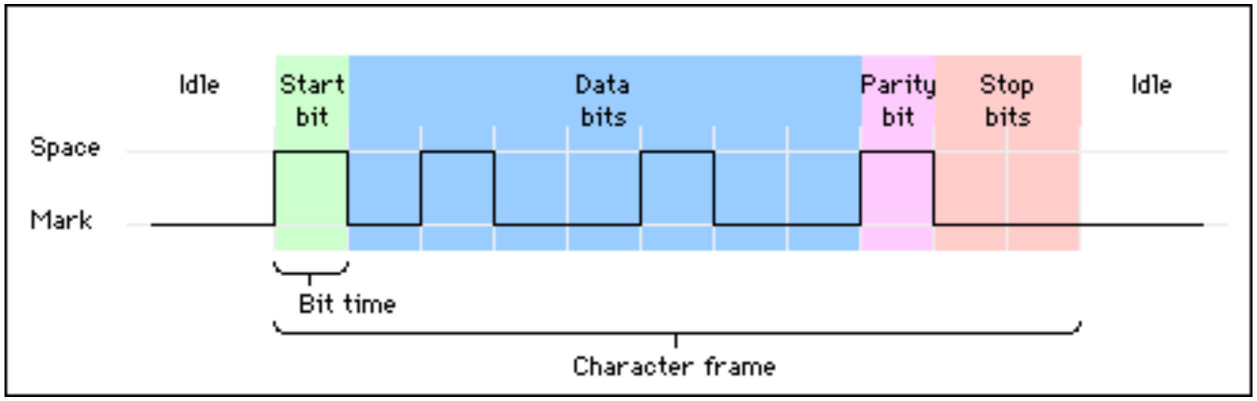
Serial Disadvantages

Reduced throughput relative to parallel

- Only one bit a time
- But as we've discussed, a high transfer rate can mitigate the disadvantage of single-bit transmission.

Added complexity of self-synchronizing protocol

Serial I/O Format



Length of Bit Time

Data Bits : Number of Data bits per Character Frame

Parity : Even, Odd, None

Stop Bits : 1, 2

Serial I/O Protocol Examples

9600 – 8 – N – 1

9600 Baud : 8 data bits : No Parity : 1 Stop bit

(9600 bits / second) / (1 + 8 + 0 + 1 bits/frame) =
960 frames / sec = 960 Bytes / sec

Efficiency = 8/(1+8+1) = 0.8

38400 – 7 – E – 2

38400 Baud : 7 data bits : Even Parity :2 Stop bit

(38400 bits / second) / (1 + 7 + 1 + 2 bits/frame) =
3490 frames / sec * 7 bits / frame = 3053 Bytes / sec

Efficiency = 7/(1+7+1+2) = 0.64

Simplex/Duplex Definitions

Simplex = unidirectional communication

- Needs only one data channel
- e.g. keyboard-to-computer, computer-to-monitor

Duplex = bidirectional communication

- ***Half-Duplex*** = only one direction at a time
  - needs only one data channel
  - Transmit privilege must be time-shared
- ***Full-Duplex*** = both directions in parallel
  - needs two data channels
  - Both ends can transmit at once

Collision

Note: Half-Duplex needs *collision detection*

Both ends could start transmitting at once.

Neither message will get through.

Both ends must detect the collision and back-off.

Basic Collision Detection Schemes

CDMA - Code Division Multiple Access

Uses signal processing voodoo to multiplex multiple signals without collisions.

CSMA - Carrier Sense Multiple Access

Each device is responsible for detecting when the interface is unutilized before transmitting.

CSMA - BA (Bitwise Arbitration)

CSMA - CA (Collision Avoidance)

CSMA - CD (Collision Detection)

TDMA - Time Division Multiple Access

We all take our turns. (Aren’t we polite?)

Maybe This Helps?

As a trivial comparison imagine a UN party where couples from different countries are invited.

TDMA

Each couple takes turns talking. They talk for a short time and then stop to let another couple talk.

CDMA

Each couple talks at the same time; however they all use different languages.

CSMA

Each couple waits until it is silent before they try to talk. They have to determine what to do if another couple also decides to start talking.

With TDMA, nobody talks over anybody else.

With CDMA, nobody understands anybody else, and ignores the “noise”.

Oh no, something's wrong.

- Errors from noise and interference
  - Faster bit rate - more vulnerable to noise
  - Longer distance - more vulnerable to noise
- Receiver should detect or correct errors:
  - EDC = Error Detection Code
    - Most common is a single parity bit
  - ECC = Error Correction Code
    - Many exist.

A Few Signaling Standards

RS-232 (Conventional Serial I/O)

- Copper cable
- Actually does have some control lines
- Receiver Logic Levels:
  - 1 = -3V … -25V
  - 0 = +3V … +25 V
- Transmitter Logic levels
  - typically: 1 = -12V & 0 = +12 V
  - very robust, cheap hardware

Data Formatting

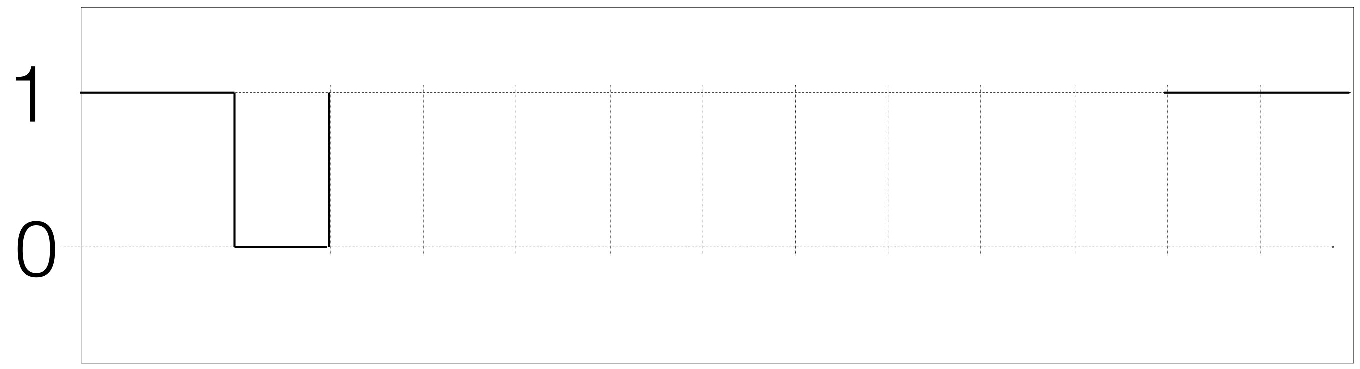
Transmitter and receiver must agree on

- Bit-clock period τ
- Endian ordering of the data
- Number of bits per word
- Parity Mode
- Framing bits to identify start of a data word
    - Start bit = 1
    - Stop bit(s) = 0
- Idle or *quiescent* state of the line

Data Formatting

Start Bit

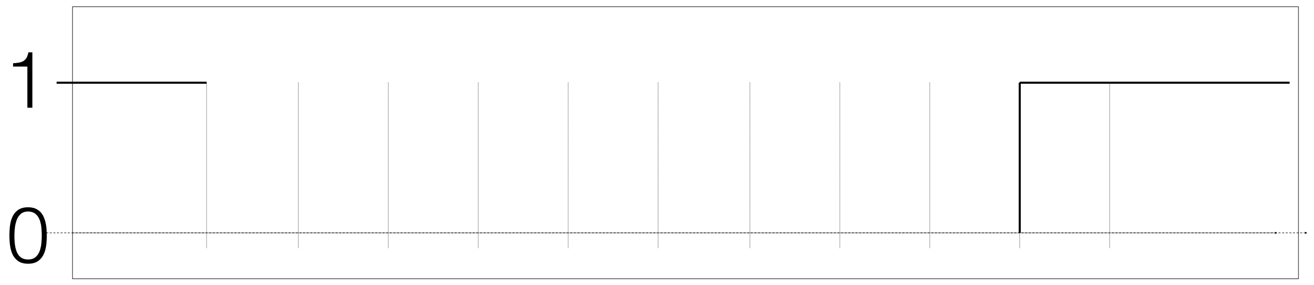
- Resting state = 1
- Need to ID start of a word to the receiver
- Start bit must be a 0
    - receiver looks for first 1 → 0 edge



Data Formatting

Stop Bit(s)

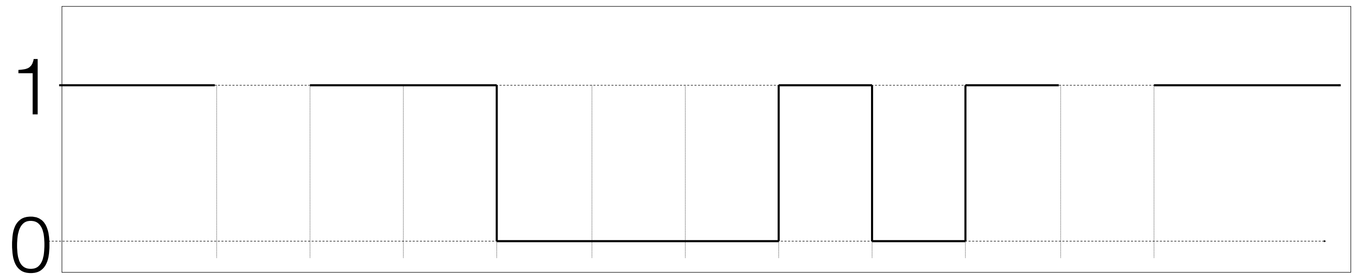
- Must return to rest state at end
  - guarantees receiver can re-sync on next start bit
- Last bit(s) of message must be a 1
  - receiver can looks for first 1 → 0 edge for next word



Data Formatting

1-Byte Data Ordering

- Low-Order bits first
- Last bit transmitted is the parity bit (if any)
  - allows transmitter to generate parity on-the-fly
  - XMIT = `%10100011`



Message Synchronization

Transmitter and receiver internal clocks don't operate at the same frequency.

Receiver must detect the start-bit (1st falling edge).

- Receiver samples much faster than the bit-rate.
    - Typically $\boxed{\text{16x}}$ bit rate
- Allows accurate detection of the falling edge

Check out that factor of 16. It will be important later.

Once start-bit is detected

- Receiver samples middle of each bit
    - allows for maximum skew between clock frequencies
    - typically does > 1 sample for noise immunity

Timing Errors

Occurs if the clock frequencies are too far out of sync.

The sample point drifts from center of bit and results in erroneous samples.

Fortunately crystal clocks are very accurate.

Consequently, not a common error anymore.

Framing Error

Start and stop bits do not properly frame the character.

- Received character doesn't end with stop bit.

Occurs if :

- Receiver gets confused on start of message
    - Interprets some zero data value as start bit
        - (If the expected start bit is a zero)
- Receiver can't find stop bit either in this character or in following character

Message Errors

Overrun Errors

ROE = Receive Overrun Error

CPU must read current word out of Rx Data Reg
before next word arrives in shift reg

Error occurs if CPU is too slow

TOE = Transmit Overrun Error

UART must load previous word from Transmit Data Register into shift register
before CPU can write next word to Transmit Data Register

Error occurs if CPU is too fast or not polling properly

I'm going to be frank: Overrun errors are almost always the programmer's fault.

UARTs

The TM4C123GH6PM controller includes eight Universal Asynchronous Receiver/Transmitter (UART) with the following features:

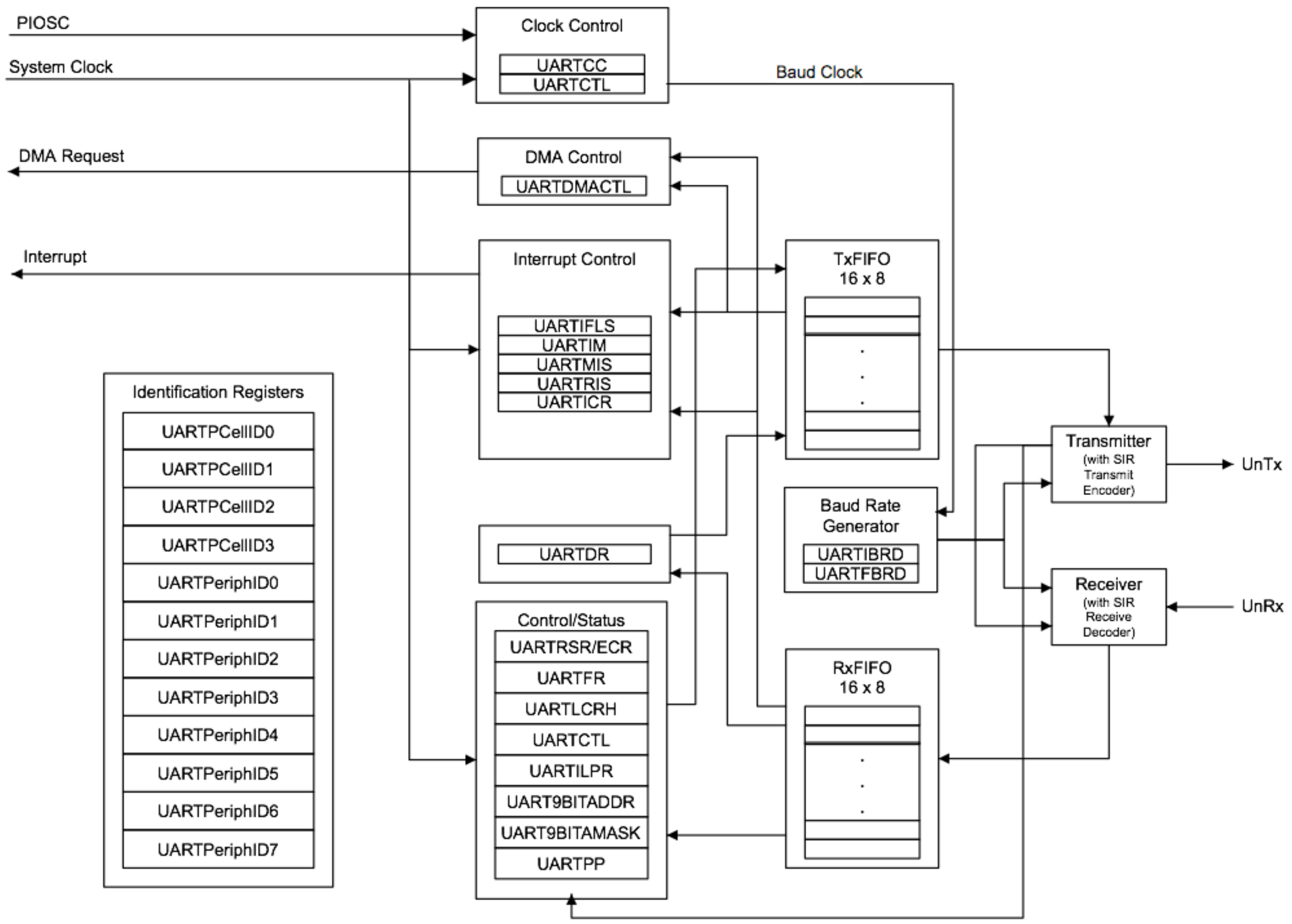
- Programmable baud-rate generator allowing speeds up to 5 Mbps for regular speed and 10 Mbps for high speed
- Separate 16x8 transmit (TX) and receive (RX) FIFOs to reduce CPU interrupt service loading
- Programmable FIFO length, including 1-byte deep operation providing conventional double-buffered interface
- Various FIFO trigger levels
- Standard asynchronous communication bits for start, stop, and parity
- Line-break generation and detection
- Fully programmable serial interface characteristics
  - 5, 6, 7 or 8 databits
  - Even, odd, stick, or no-parity bit generation/detection
  - 1 or 2 stop bit generation

UARTs

The TM4C123GH6PM controller includes eight Universal Asynchronous Receiver/Transmitter (UART) with the following features:

- IrDA serial-IR (SIR) encoder/decoder providing
  - Programmable use of IrDA Serial Infrared (SIR) or UART input/output
  - Support of IrDA SIR encoder/decoder functions for data rates up to 115.2 Kbps half-duplex
  - Programmable internal clock generator enabling division of reference clock by 1 to 256 for low-power mode bit duration
- Support for communication with ISO 7816 smart cards
- Modem flow control (on UART1)
- EIA-485 9-bit support
- Standard FIFO-level and End-of-Transmission interrupts
- Efficient transfers using Micro Direct Memory Access Controller (µDMA)

UART Block Diagram



Calculating Baud Rate Dividers

Creating a baud rate is done through a 22-bit divider.

16 bits of integer and 6 bits of fraction

We need a few variables:

By default, the 80 MHz clock.

`UARTSysClk` — The system clock

I told you 16 was going to be important.

`ClkDiv` — The divider (16 for regular speed, 8 for high speed)

`Baud Rate` — The target baud rate

`BDRI` — The integer portion of the divider

`BDRF` — The fractional portion of the divider

`BRD` — The overall divider value

The equations are:

```
BRD = BRDI + BRDF = UARTSysClk / (ClkDiv * Baud Rate)
```

Fractional Divider Calculations

If `BRDF` is a fractional value, how do we put it in a register?

`integer(BRDF` * 64 + 0.5)

Baud Rate Example

Let’s say our target baud rate is 14,400.

`BRD` = `BRDI` + `BRDF` = 80000000 / (16 * 14400) = 347.2222222222222

`BRDI` = 347, `BRDF` = .22222222

Value in the register = `integer`(.222222 * 64 + 0.5) = `integer`(14.722222222222208) = 14.

Flow Control

Remember our old discussion about throughput vs. response time?

Sometimes serial communications devices can't keep up with the data being sent to them.

So we use *handshaking* signals to keep problems from arising.

One device asserts a signal indicating that it is ready to send.

  This is the `RTS` signal.

When appropriate, the other device asserts a signal indicating it is clear to send.

  This is the `CTS` signal.

The Tiva C UART is capable of using these signals (or not).

Interrupts

The UART can generate interrupts on the following conditions:

- Overrun Error
- Break Error
- Parity Error
- Framing Error
- Receive Timeout
- Transmit Complete
- Receiver Complete

DMA Integration

There are separate µDMA channels for transmitting and receiving data from the UART.

If enabled, the controller is triggered by a specific FIFO threshold.

Initialization and Configuration

To enable and initialize the UART, the following steps are necessary:

1. Enable the UART module using the `RCGCUART` register

2. Enable the clock to the appropriate GPIO module via the `RCGCGPIO` register.

    To find out which GPIO port to enable, refer to Table 23-5 in the Reference Manual.

3. Set the GPIO `AFSEL` bits for the appropriate pins.

    To determine which GPIOs to configure, see Table 23-4 in the Reference Manual.

4. Configure the GPIO current level and/or slew rate as specified for the mode selected.

5. Configure the `PMCn` fields in the `GPIOPCTL` register to assign the UART signals to the appropriate pins.

At this point, the UART is *enabled*, but not configured for operation. Let's do that next.

Configuration

The UART configuration is written to the module in the following order:

1. Disable the UART by clearing the `UARTEN` bit in the `UARTCTL` register.

2. Write the integer portion of the BRD to the `UARTIBRD` register.

3. Write the fractional portion of the BRD to the `UARTFBRD` register.

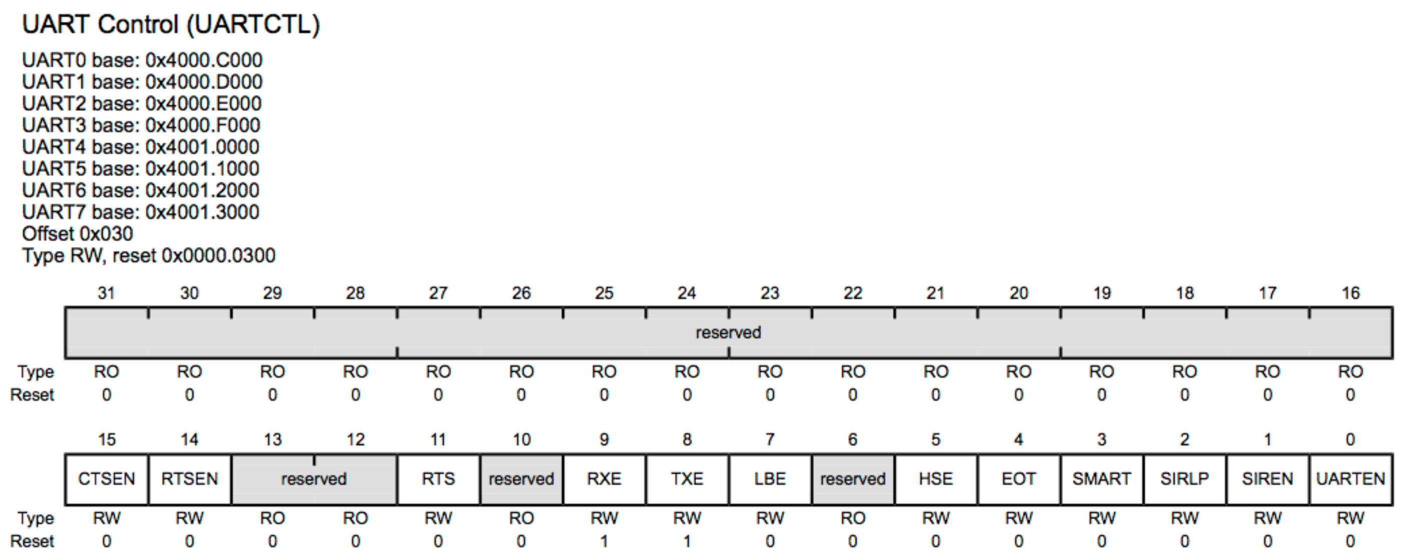
4. Write the desired serial parameters to the `UARTLCRH` register.

5. Configure the UART clock source by writing to the `UARTCC` register.

6. Optionally, configure the µDMA channel and enable the DMA option(s) in the `UARTDMACTL` register.

7. Enable the UART by setting the `UARTEN` bit in the `UARTCTL` register.

UARTCTL



UART Control (UARTCTL)

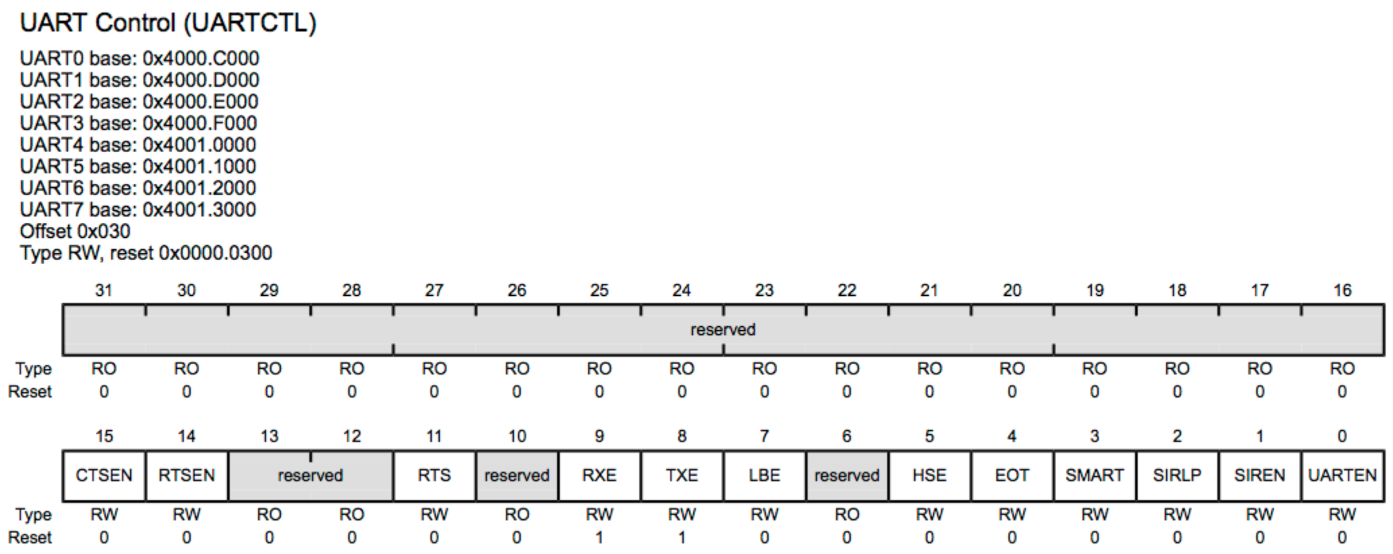
UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x030
Type RW, reset 0x0000.0300

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CTSEN	RTSEN	reserved		RTS	reserved	RXE	TXE	LBE	reserved	HSE	EOT	SMART	SIRLP	SIREN	UARTEN
Type	RW	RW	RO	RO	RW	RO	RW	RW	RW	RO	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Bit	0	1
`CTSEN`	CTS Flow Control Disabled	CTS Flow Control Enabled
`RSTEN`	RTS Flow Control Disabled	RTS Flow Control Enabled
`RTS`	UART Not Ready to Send	UART Ready to Send
`RXE`	Receiver Disabled	Receiver Enabled
`TXE`	Transmitter Disabled	Transmitter Enabled

UARTCTL



UART Control (UARTCTL)

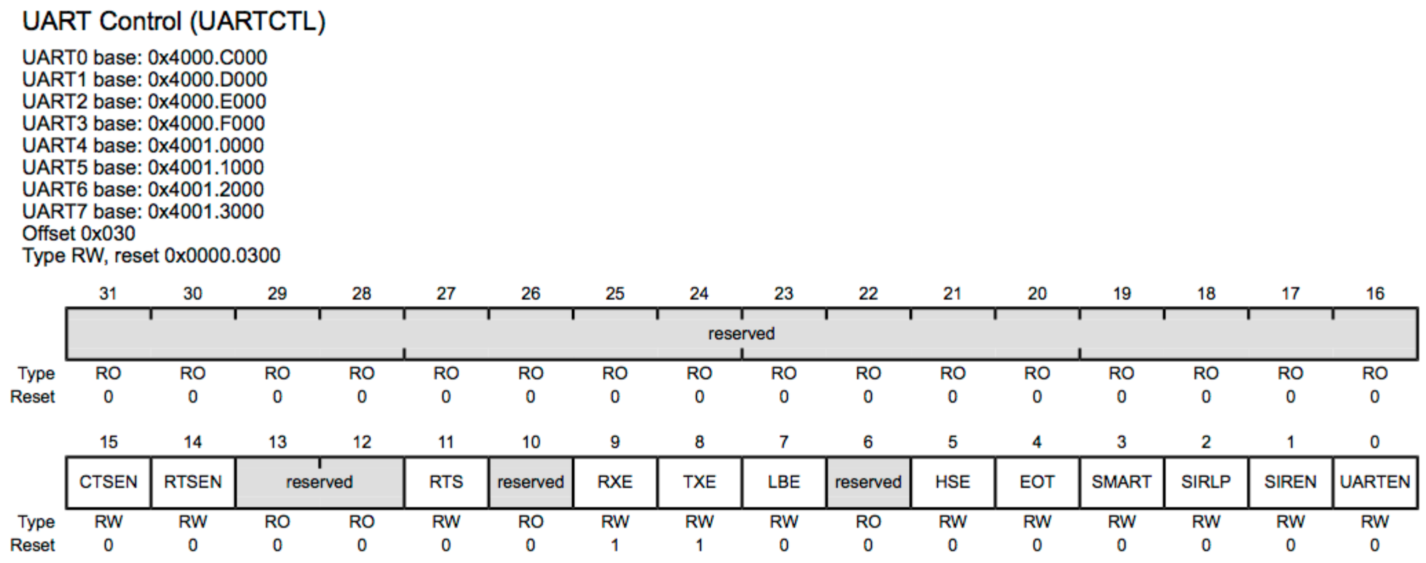
UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x030
Type RW, reset 0x0000.0300

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CTSEN	RTSEN	reserved		RTS	reserved	RXE	TXE	LBE	reserved	HSE	EOT	SMART	SIRLP	SIREN	UARTEN
Type	RW	RW	RO	RO	RW	RO	RW	RW	RW	RO	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Bit	0	1
LBE	Loopback Disabled	Loopback Enabled
HSE	High Speed Disabled (Clock Divider = 16)	High Speed Enabled (Clock Divider = 8)
EOT	END Flag set at FIFO threshold	END Flag set when 0 bits left to send

UARTCTL



UART Control (UARTCTL)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x030
Type RW, reset 0x0000.0300

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CTSEN	RTSEN	reserved		RTS	reserved	RXE	TXE	LBE	reserved	HSE	EOT	SMART	SIRLP	SIREN	UARTEN
Type	RW	RW	RO	RO	RW	RO	RW	RW	RW	RO	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Bit	0	1
`SMART`	Normal Operation	Smart Code Mode
`SIRLP`	Normal IrDA Operation	Low-power IrDA Operation
`SIREN`	Normal UART Operation	IrDA UART Operation
`UARTEN`	UART Disabled	UART Enabled

A Note on `UARTCTL`

We have to be careful to access `UARTCTL` appropriately. The safe way to do it:

1. Disable the UART.

2. Wait for the end of transmission or reception of the current character.

3. Flush the transmit FIFO by clearing bit 4 (`FEN`) in the line control register (`UARTLCRH`).

4. Reprogram the control register.

5. Enable the UART.

Sample Configuration

Configuration for a UART with full flow control:

```
UART1_CTL_R = 0xC301;
```

Sets `CTSEN`, `RTSEN`, `RXE`, `TXE`, `UARTEN`.

How about the TivaWare functions?

There are many…

TivaWare UART Configuration Functions

`UARTDisable()`

  And `UARTDisableSIR()`

`UARTEnable()`

  And `UARTEnableSIR()`

`UARTFlowControlSet()`

`UARTModemControlGet() // Just for RTS bit`

`UARTSmartCardDisable()`

`UARTSmartCardEnable()`

`UARTTxIntModeSet()`

TivaWare UART Configuration Functions

`UARTDisable()` ←

  And `UARTDisableSIR()`

`UARTEnable()` ←

  And `UARTEnableSIR()`

`UARTFlowControlSet()` ←

`UARTModemControlGet() // Just for RTS bit`

`UARTSmartCardDisable()`

`UARTSmartCardEnable()`

`UARTTxIntModeSet()`

Let's just talk about these.

`UARTDisable()`

Disables transmitting and receiving.

```
void UARTDisable(uint32_t ui32Base)
```

`ui32Base` is the base address of the UART port.

This function disables the UART, waits for the end of transmission of the current character, and flushes the transmit FIFO.

`UARTEnable()`

Enables transmitting and receiving.

```
void UARTEnable(uint32_t ui32Base)
```

`ui32Base` is the base address of the UART port.

This function enables the UART and its transmit and receive FIFOs.

`UARTFlowControlSet()`

Sets the UART hardware flow control mode to be used.

```
void UARTFlowControlSet(uint32_t ui32Base, uint32_t ui32Mode)
```

- `ui32Base` is the base address of the UART port.
- `ui32Mode` indicates the flow control modes to be used.
  - This parameter is a logical OR combination of values `UART_FLOWCONTROL_TX` and `UART_FLOWCONTROL_RX` to enable hardware transmit (`CTS`) and receive (`RTS`) flow control or `UART_FLOWCONTROL_NONE` to disable hardware flow control.

This function configures the required hardware flow control modes. If `ui32Mode` contains flag `UART_FLOWCONTROL_TX`, data is only transmitted if the incoming `CTS` signal is asserted. If `ui32Mode` contains flag `UART_FLOWCONTROL_RX`, the `RTS` output is controlled by the hardware and is asserted only when there is space available in the receive FIFO. If no hardware flow control is required, `UART_FLOWCONTROL_NONE` should be passed.

Sample Configuration

Configuration for a UART with full flow control:

```
UART1_CTL_R = 0xC301;
```

Sets `CTSEN`, `RTSEN`, `RXE`, `TXE`, `UARTEN`.

How about the TivaWare functions?

```
UARTFlowControlSet(UART1_BASE,
UART_FLOWCONTROL_TX |
UART_FLOWCONTROL_RX);

UARTEnable();
```

How About Bit Rate Configuration?

What are the registers and TivaWare functions for setting the bit rate?

Integer Portion of Bit Rate: `UARTIBRD`

Fractional Portion of Bit Rate: `UARTFBRD`

These two registers just hold numbers, which is not interesting.

TivaWare Function: `UARTConfigSetExpClk()`

Let's talk about this function instead.

UARTConfigSetExpClk()

Sets the configuration of a UART.

```
void UARTConfigSetExpClk(uint32_t ui32Base,
uint32_t ui32UARTClk, uint32_t ui32Baud, uint32_t
ui32Config)
```

`ui32Base` is the base address of the UART port.

`ui32UARTClk` is the rate of the clock supplied to the UART module.

`ui32Baud` is the desired baud rate.

`ui32Config` is the data format for the port (number of data bits, number of stop bits, and parity).

`UARTConfigSetExpClk()`

The `ui32Config` parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity.

> `UART_CONFIG_WLEN_8`, `UART_CONFIG_WLEN_7`, `UART_CONFIG_WLEN_6`, and `UART_CONFIG_WLEN_5` select from eight to five data bits per byte (respectively).
>
> `UART_CONFIG_STOP_ONE` and `UART_CONFIG_STOP_TWO` select one or two stop bits (respectively).
>
> `UART_CONFIG_PAR_NONE`, `UART_CONFIG_PAR_EVEN`, `UART_CONFIG_PAR_ODD`, `UART_CONFIG_PAR_ONE`, and `UART_CONFIG_PAR_ZERO` select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).

The peripheral clock is the same as the processor clock. The frequency of the system clock is the value returned by `SysCtlClockGet()`.

Sample Configuration

Desired connection: 115,200-8-E-2

TivaWare Function:

```
UARTConfigSetExpClk(UART0_BASE,
SysCtlClockGet(), 115200, UART_CONFIG_WLEN_8 |
UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_TWO);
```

Just for reference, the DRA code:

```
UART0_IBRD_R = 43;
UART0_FBRD_R = 26;
// WLEN=0x3, STP2=EPS=PEN=1, so…
UART0_CRH_R = 0x6E;
```

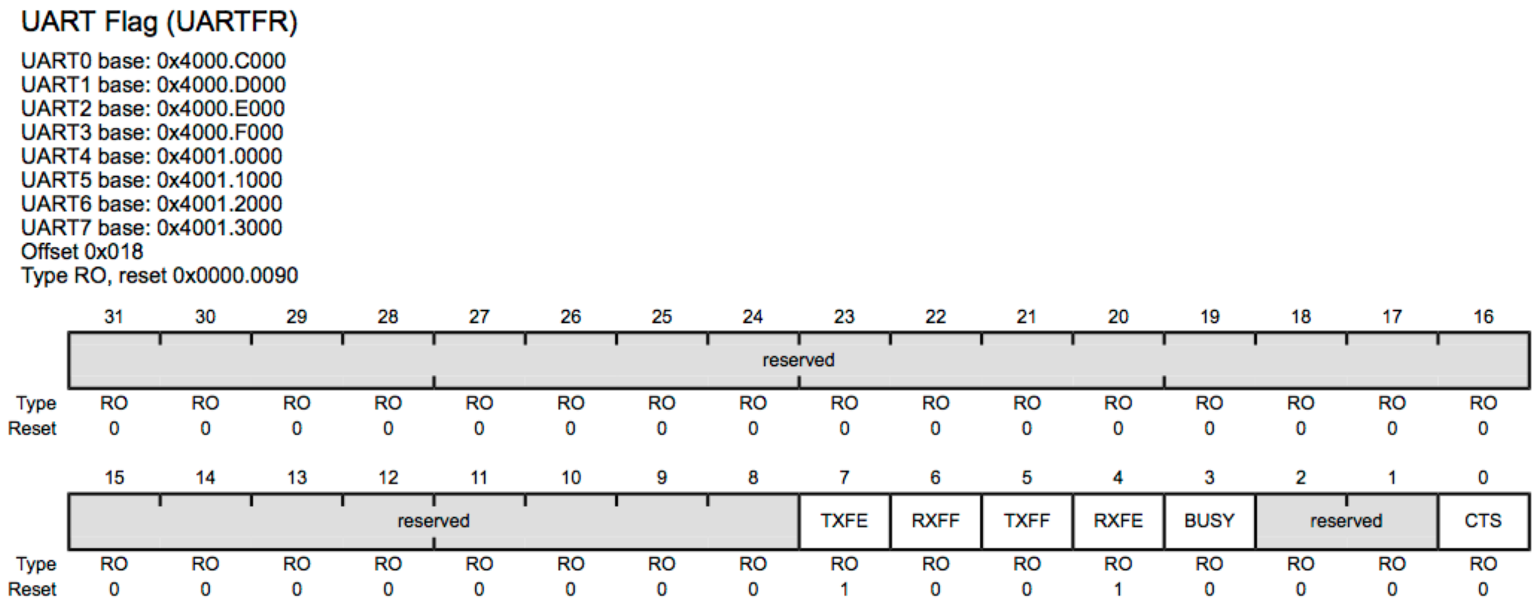
Let's Send Some Characters!

The DRA Method: Check the flags in `UARTFR` to see if `TXE=1` (there's space to write a character). Then, write a character to `UARTDR`.

The TivaWare Method: `UARTCharPut()`

Let's look at both a little.

UARTFR



UART Flag (UARTFR)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x018
Type RO, reset 0x0000.0090

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								TXFE	RXFF	TXFF	RXFE	BUSY	reserved		CTS
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0

Bit	0	1
TXFE	Transmit FIFO Full	Transmit FIFO Empty
RXFF	Receive FIFO has space	Receive FIFO Full
TXFF	Transmit FIFO has space	Transmit FIFO Full
RXFE	Receive FIFO Full	Receive FIFO Empty

Sample DRA Code

```
char charToSend = ‘k’;
while (0x80 != (UART0_FR_R & 0x80)) ;
UART0_DR_R = charToSend;
```

UARTCharPut()

Waits to send a character from the specified port.

```
void UARTCharPut(uint32_t ui32Base,
unsigned char ucData)
```

`ui32Base` is the base address of the UART port.

`ucData` is the character to be transmitted.

This function sends the character `ucData` to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.

Sample TivaWare Code

```
char charToSend = ‘k’;
UARTCharPut(UART0_BASE, charToSend);
```

Receiving Characters

Still poll flags in `UARTFR`, then read values from `UARTDR`.

Use the `UARTCharGet()` function.

Or… `UARTCharGetNonBlocking()`.

Sample DRA Code

```
char charToGet;
while (0x10 != (UART0_FR_R & 0x10)) ;
charToSend = UART0_DR_R;
```

UARTCharGet()

Waits for a character from the specified port.

`int32_t UARTCharGet(uint32_t ui32Base)`

`ui32Base` is the base address of the UART port.

This function gets a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning.

This "waiting" thing is critical!

`UARTCharGetNonBlocking()`

Receives a character from the specified port.

```
int32_t UARTCharGetNonBlocking(uint32_t ui32Base)
```

`ui32Base` is the base address of the UART port.

This function gets a character from the receive FIFO for the specified port.

Returns the character read from the specified port, cast as a `int32_t`. The function returns `-1` is returned if there are no characters present in the receive FIFO.

The `UARTCharsAvail()` function should be called before attempting to call this function.

`UARTCharsAvail()`

Determines if there are any characters in the receive FIFO.

```
bool UARTCharsAvail(uint32_t ui32Base)
```

`ui32Base` is the base address of the UART port.

This function returns a flag indicating whether or not there is data available in the receive FIFO.

Returns `true` if there is data in the receive FIFO or `false` if there is no data in the receive FIFO.

What’s the Difference?

`UARTCharGet()` will wait until a character arrives, potentially preventing anything else from occurring.

`UARTCharGetNonBlocking()` will return almost immediately if there are no characters to receive.

> But you shouldn’t call it then. You should always call `UARTCharsAvail()` first.

Sample TivaWare Code

Version 1:

```
char charIn = (char) UARTCharGet(UART0_BASE);
```

Version 2:

```
while (!UARTCharsAvail()) ;
char charIn = (char) UARTCharGetNonBlocking();
```

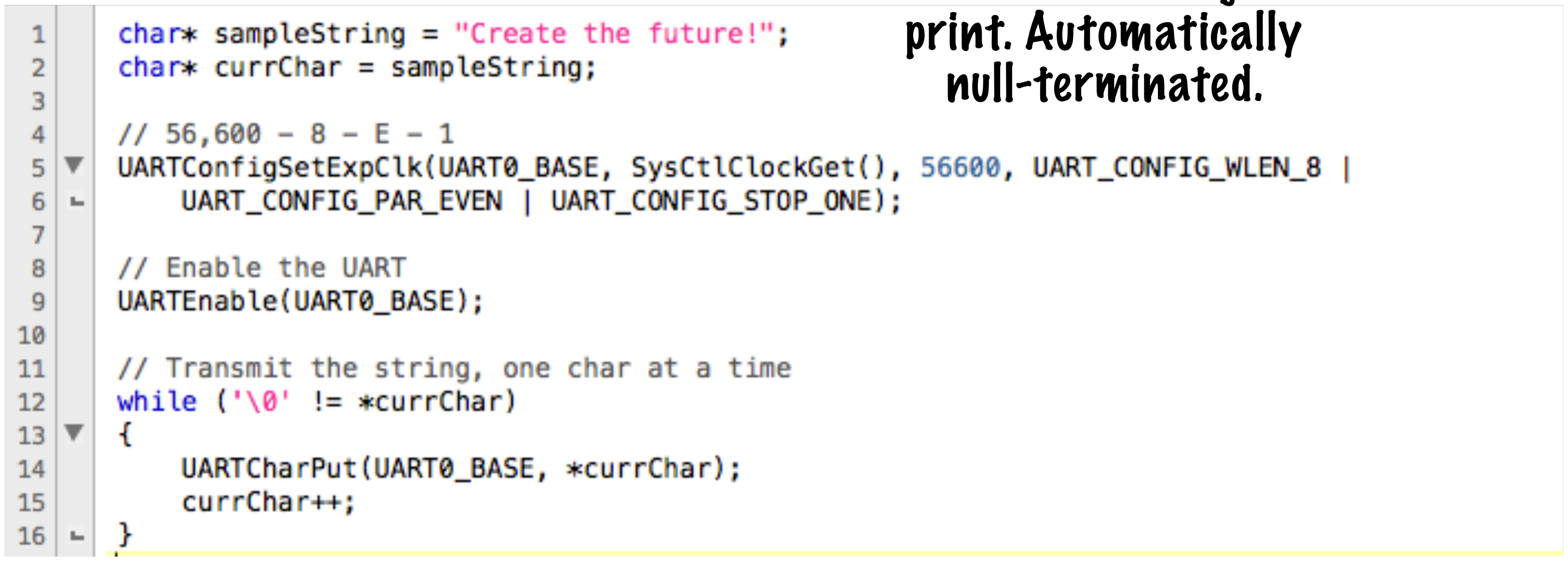
Bigger Example

Configure the Tiva C for a 56600-8-E-1 serial link.

Send a buffer of characters out the link.

Code

Here's the string to print. Automatically null-terminated.



```c
char* sampleString = "Create the future!";
char* currChar = sampleString;

// 56,600 - 8 - E - 1
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 56600, UART_CONFIG_WLEN_8 |
    UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_ONE);

// Enable the UART
UARTEnable(UART0_BASE);

// Transmit the string, one char at a time
while ('\0' != *currChar)
{
    UARTCharPut(UART0_BASE, *currChar);
    currChar++;
}
```

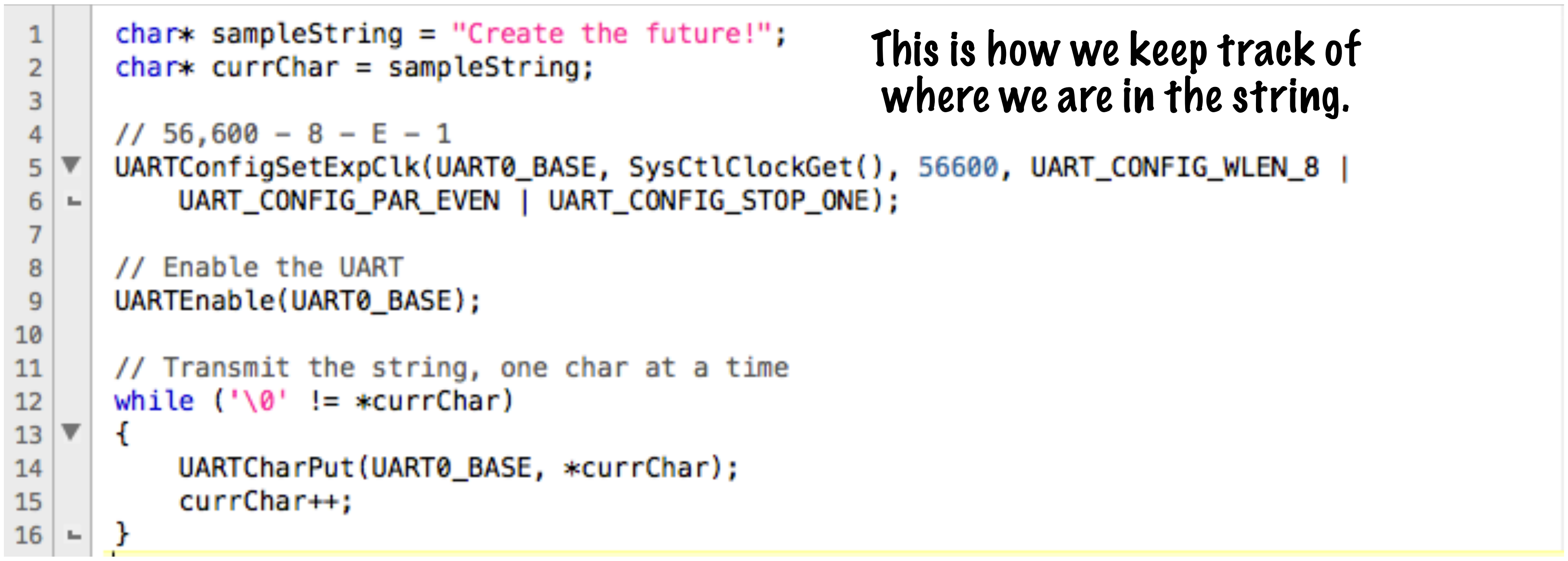
Null-Termination

C automatically appends the ASCII character `\0` to all strings.

This acts as a *sentinel* value, to help us find the end.

Bonus: We don't need to know how many characters are in the string.

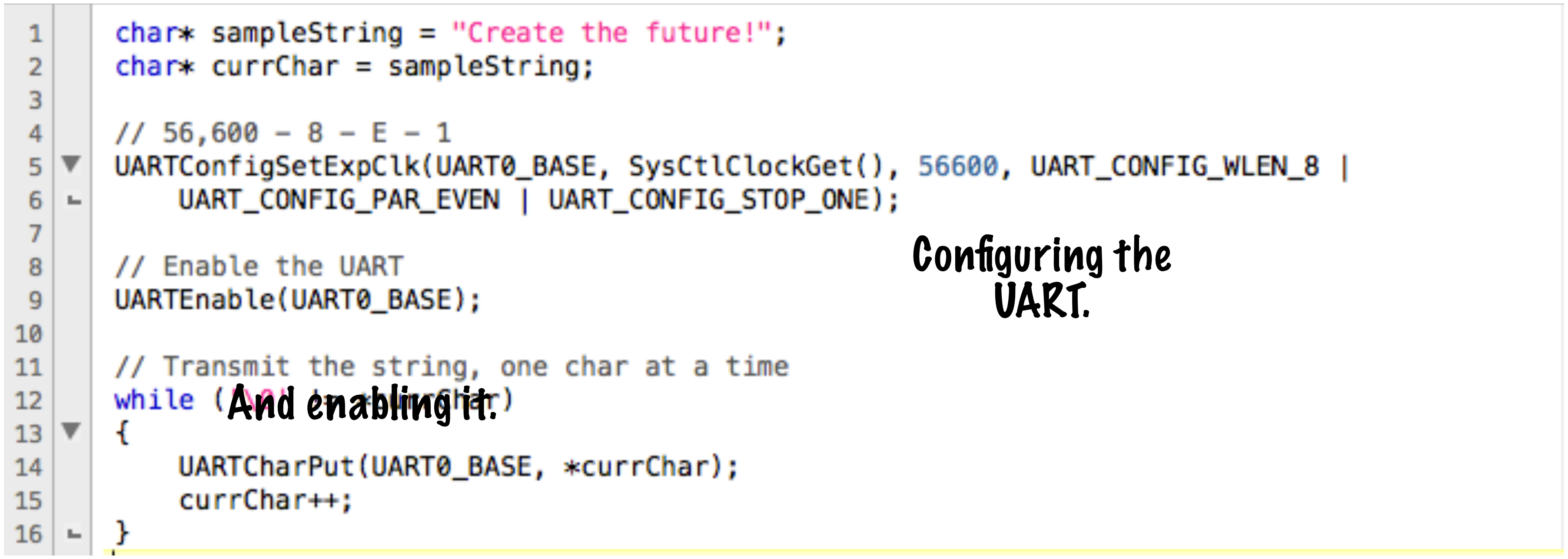
Code



```
char* sampleString = "Create the future!";
char* currChar = sampleString;

// 56,600 - 8 - E - 1
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 56600, UART_CONFIG_WLEN_8 |
    UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_ONE);

// Enable the UART
UARTEnable(UART0_BASE);

// Transmit the string, one char at a time
while ('\0' != *currChar)
{
    UARTCharPut(UART0_BASE, *currChar);
    currChar++;
}
```

This is how we keep track of where we are in the string.

Code

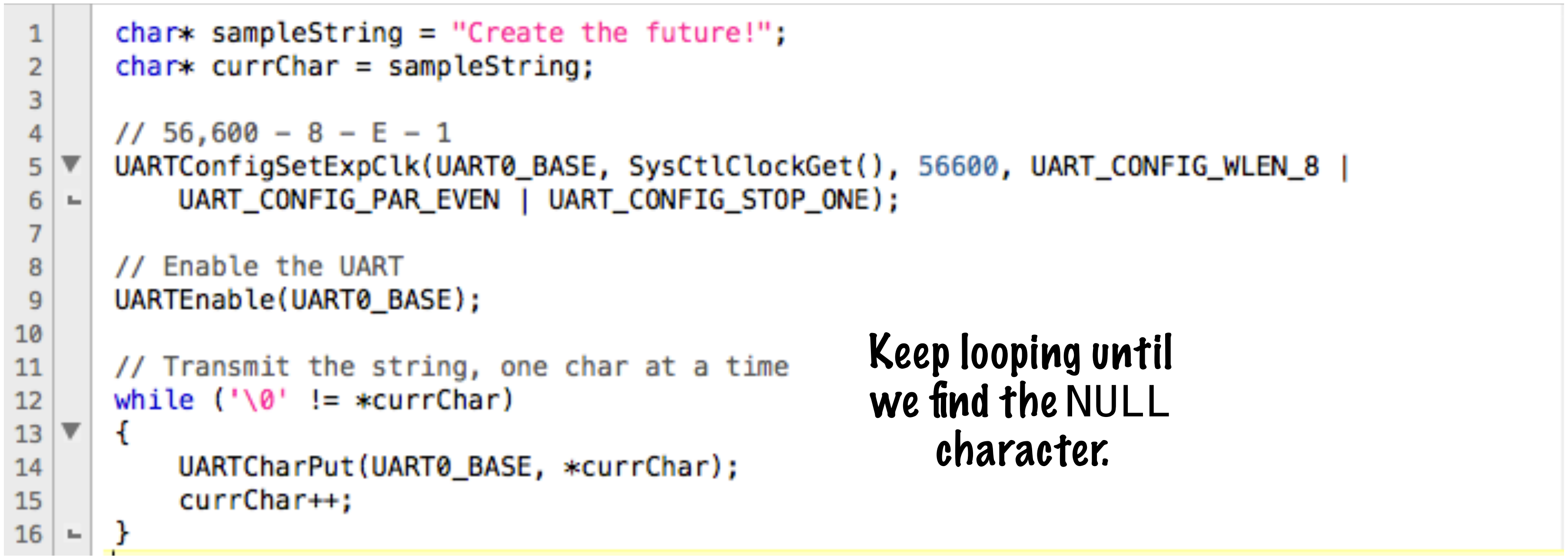


```c
char* sampleString = "Create the future!";
char* currChar = sampleString;

// 56,600 - 8 - E - 1
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 56600, UART_CONFIG_WLEN_8 |
    UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_ONE);

// Enable the UART
UARTEnable(UART0_BASE);

// Transmit the string, one char at a time
while ('\0' != *currChar)
{
    UARTCharPut(UART0_BASE, *currChar);
    currChar++;
}
```

Configuring the UART.

And enabling it.

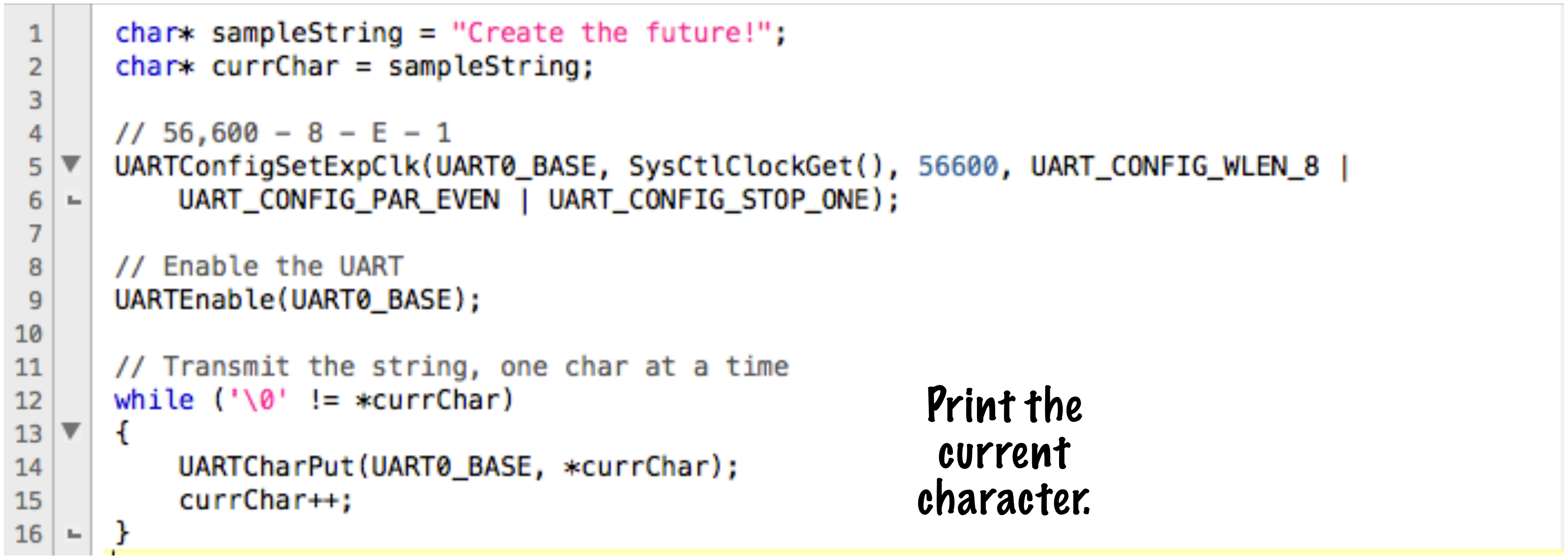
Code



```c
char* sampleString = "Create the future!";
char* currChar = sampleString;

// 56,600 - 8 - E - 1
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 56600, UART_CONFIG_WLEN_8 |
    UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_ONE);

// Enable the UART
UARTEnable(UART0_BASE);

// Transmit the string, one char at a time
while ('\0' != *currChar)
{
    UARTCharPut(UART0_BASE, *currChar);
    currChar++;
}
```

Keep looping until we find the NULL character.

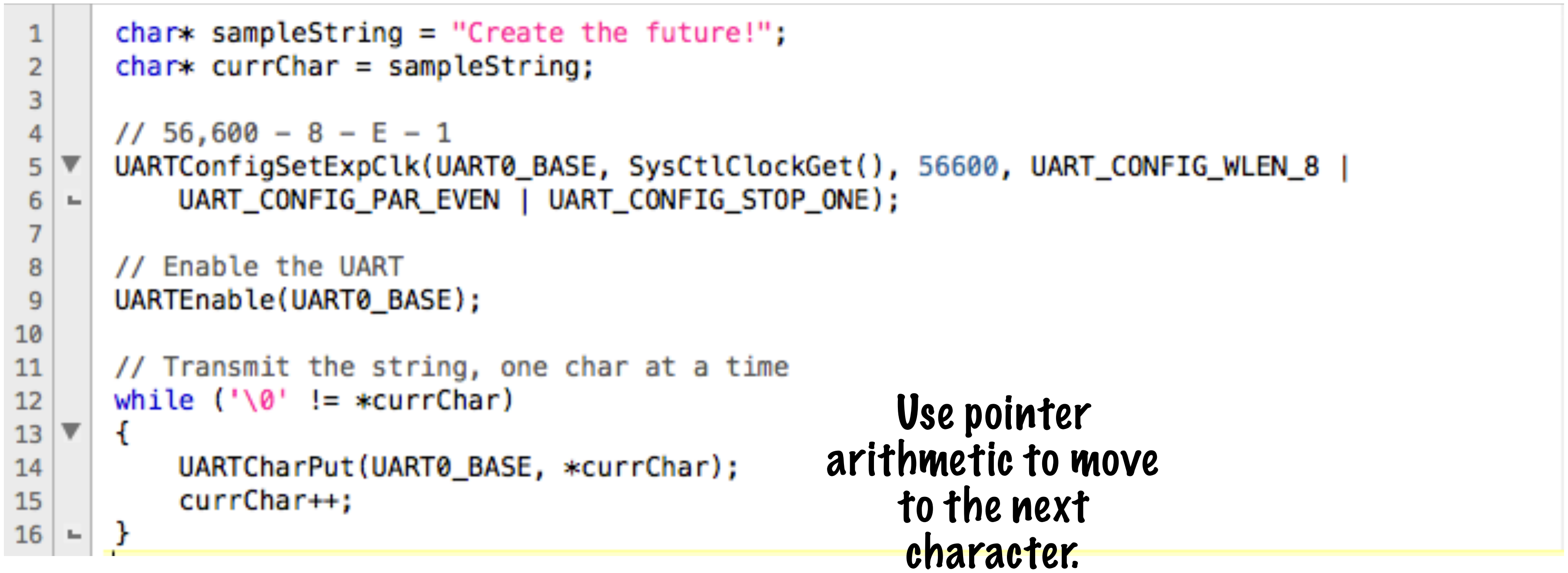
Code



```c
char* sampleString = "Create the future!";
char* currChar = sampleString;

// 56,600 - 8 - E - 1
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 56600, UART_CONFIG_WLEN_8 |
    UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_ONE);

// Enable the UART
UARTEnable(UART0_BASE);

// Transmit the string, one char at a time
while ('\0' != *currChar)
{
    UARTCharPut(UART0_BASE, *currChar);
    currChar++;
}
```

Print the current character.

Code



```c
char* sampleString = "Create the future!";
char* currChar = sampleString;

// 56,600 - 8 - E - 1
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 56600, UART_CONFIG_WLEN_8 |
    UART_CONFIG_PAR_EVEN | UART_CONFIG_STOP_ONE);

// Enable the UART
UARTEnable(UART0_BASE);

// Transmit the string, one char at a time
while ('\0' != *currChar)
{
    UARTCharPut(UART0_BASE, *currChar);
    currChar++;
}
```

Use pointer arithmetic to move to the next character.

Interrupts and Errors

The UART is capable of generating interrupts on the following conditions:

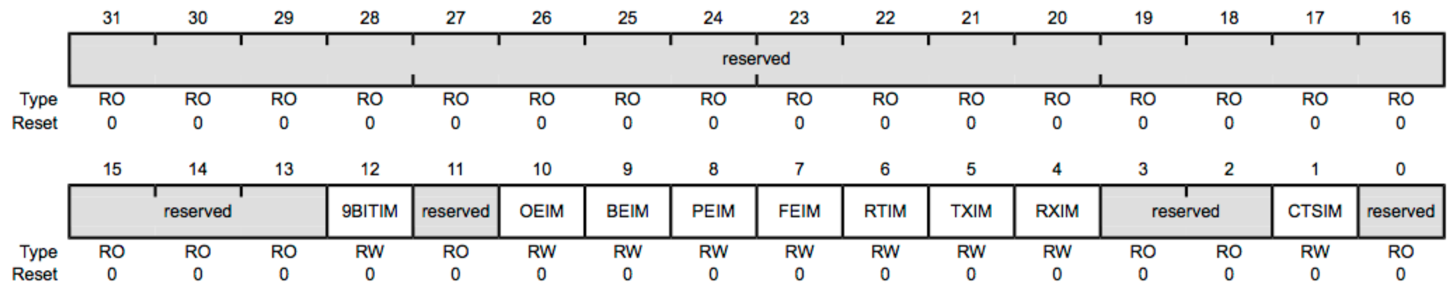
- 9-bit Address Match
- Overrun error
- Break error
- Parity error
- Framing error
- Receive Timeout
- Transmit Event
- Receive Event
- CTS Signal Received

These are all configured in the `UARTIM` register or using `UARTIntEnable()`.

UART Interrupt Masks

UART Interrupt Mask (UARTIM)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x038
Type RW, reset 0x0000.0000



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved			9BITIM	reserved	OEIM	BEIM	PEIM	FEIM	RTIM	TXIM	RXIM	reserved		CTSIM	reserved
Type	RO	RO	RO	RW	RO	RW	RW	RW	RW	RW	RW	RW	RO	RO	RW	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

`UARTIntEnable()`

Enables individual UART interrupt sources.

```
void UARTIntEnable(uint32_t ui32Base, uint32_t
ui32IntFlags)
```

`ui32Base` is the base address of the UART port.

`ui32IntFlags` is the bit mask of the interrupt sources to be enabled.

This function enables the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

`UARTIntEnable()`

The ui32IntFlags parameter is the logical OR of any of the following:

- `UART_INT_9BIT` - 9-bit Address Match interrupt
- `UART_INT_OE` - Overrun Error interrupt
- `UART_INT_BE` - Break Error interrupt
- `UART_INT_PE` - Parity Error interrupt
- `UART_INT_FE` - Framing Error interrupt
- `UART_INT_RT` - Receive Timeout interrupt
- `UART_INT_TX` - Transmit interrupt
- `UART_INT_RX` - Receive interrupt
- `UART_INT_CTS` - CTS interrupt

Summary

Serial I/O is used when parallel is impractical due to distance or interconnect limitations.

Timing is inherent in the Serial I/O protocol.

Serial I/O functional unit is called UART in most modern microcontrollers.

The UARTs on the Tiva C are flexible and powerful, but still relatively slow (as a result of the protocol itself).