8-BitUART 数据表 UART V 5.3

Copyright © 2001-2010 Cypress Semiconductor Corporation. All Rights Reserved.

资源	PSoC® 模块			API 存储器 （字节）		引脚 （每个外部 I/O）
	数字	模拟 CT	模拟 SC	闪存	RAM	
CY8C29/27/24/22/21xxx、CY8C23x33、CY7C64215、CYWUSB6953、CY8CLED02/04/08/16、CY8CLED0xD、CY8CLED0xG、CY8CTST110, CY8CTMG110、CY8CTST120、CY8CTMG120、CY8CTMA120、CY8C21x45、CY8C22x45、CY8CTMA30xx、CY8C28x45、CY8CPLC20、CY8CLED16P01、CY8C28x43、 CY8C28x52						
底层 API	2			311	0	2
高层 API	2			506	3 + 缓冲区	2
底层 API	2			351	0	2
高层 API	2			546	3 + 缓冲区	2

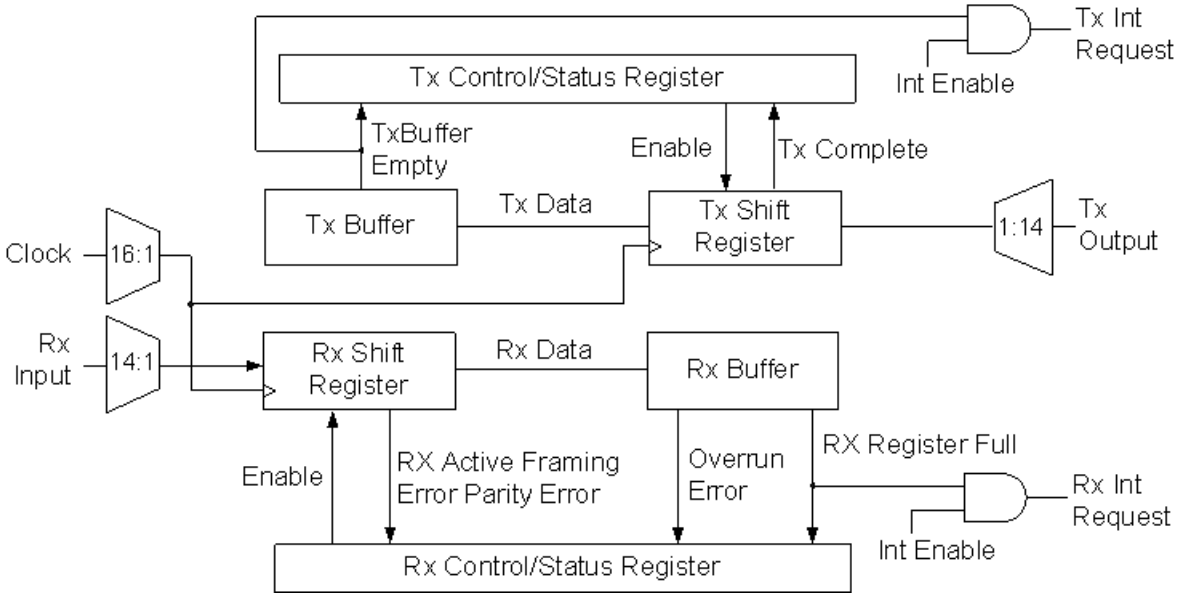
如需一个或多个使用此用户模块的完全配置的功能性示例工程，请转到

特性与概述

- 异步接收器和发射器
- 数据格式符合 RS-232 串行数据格式
- 突发速率高达 6 Mbits/秒
- 数据成帧包括起始位、可选奇偶校验位和停止位
- 可选择在接收寄存器满和 / 或发送缓冲区空的条件下触发中断
- 奇偶校验、过速和成帧错误检测
- 高层发送和接收函数

UART 用户模块是一款 8-bit 通用型异步接收发送器，支持通过 2 根接线实现符合 RS-232-compliant 数据格式的双工串行通信。 所接收和发送数据的格式包括了 1 个起始位、可选奇偶校验位和 1 个停止位。 支持可编程的时钟和可选择的中断或轮询的运行方式。 还提供了用于初始化、配置和操作 UART 的应用程序编程接口 (API) 固件子程序。 还额外提供了高层 API 以支持后台命令接收和字符串打印。

Figure 1. UART 原理方框图



功能说明

UART 用户模块实现了一种串行接收发送器。 UART 可以在 PSoC Designer 器件编辑器内映射到名为 TX 和 RX 的 2 个 PSoC 模块上。 TX PSoC 模块提供了发送器功能，RX PSoC 模块提供了接收器功能。

RX 模块和 TX 模块相互独立运行。 每个模块均拥有自己的控制寄存器和状态寄存器、可编程中断、I/O、缓冲区寄存器和移位寄存器。 模块共享相同的启用、时钟和数据格式。

在 RX 控制寄存器和 TX 控制寄存器内设置使能位能够启用 UART 的运行。 启用和禁用通过使用 API 提供的函数来执行。

UART 用户模块的时钟由 RX 组件和 TX 组件共享。 所选定的时钟频率必须是所要求数据位速率的 8 倍。 每个接收或发送的数据位均要求占用 8 个输入时钟周期。 此时钟采用 PSoC Designer 器件编辑器进行配置。

所接收和发送的数据是一种由 1 个起始位、8 个数据位、1 个可选奇偶校验位和 1 个停止位构成的位流。 奇偶校验方式可以设置为无、偶校验或奇校验，并且可以采用 PSoC Designer 器件编辑器或 UART API 来设置。 RX 和 TX 均设置使用相同的奇偶校验配置。

TX - UART 发送器

发送器使用了一个数字通信型 PSoC 模块的 TX 缓冲寄存器、TX 移位和 TX 控制寄存器。

TX 控制寄存器是通过 UART 用户模块的固件 API 子程序进行初始化和配置的。 当设置 TX 控制寄存器内的使能位时，将会生成 1 个内部 8 分频的位时钟。

将要发送的数据字节由 API 子程序写入至 TX 缓冲寄存器，同时会清空 TX 控制寄存器内的 “TX 缓冲区空 ” 状态位。 这个状态位可以用于检测和防止发送过速错误。

在下一个位时钟的上升沿会把数据传输至移位寄存器并在 TX 控制寄存器内将 “TX 缓冲区空 ” 位置位。 如果中断使能掩码被启用，则将触发一个中断。 此中断可被用来启用下一个字节的排队，以便在当前数据字节传输完成时，新字节将在下一个可用发送时钟发送出去。

在数据字节从 TX 缓冲区寄存器传输至 TX 移位寄存器的同时，起始位将被发送出去。 接下来的位时钟将串行位流移位至输出。 这个位流由数据字节的每个位（低位在前）、1 个可选的奇偶校验位以及最后的 1

[+] Feedback

个停止位构成。 在完成停止位的发送时，TX 控制寄存器的 “TX 完成 ” 状态位已经置位。 此位将在被读取之前一直有效。 如果新数据字节已经写入至 TX 缓冲区寄存器，则数据字节将传输至 TX 移位寄存器而且数据的传输将在位时钟的下一个上升沿开始。

RX - UART 接收器

接收器使用了一个数字通信型 PSoC 模块的 RX 缓冲器、RX 移位和 RX 控制寄存器。

RX 控制寄存器是在 UART 用户模块的固件 API 子程序中执行初始化和配置的。 RX 的初始化包括设置 UART 奇偶校验、可选择性地启用在 RX 寄存器已满的条件下发生中断，以及随后使能 UART 的操作。

当 RX 输入信号检测到起始位时，将会启动一个 8 分频位时钟，并同步成刚好在所接收的各比特数据中间部位进行采样。 在接下来的 8 位时钟的上升沿，输入数据被采样并移位到 RX 移位寄存器内。 如果启用了奇偶校验，则下一个位时钟将对奇偶校验位进行采样。 在下一个时钟上对停止位采样会让已接收到的数值字节传输至 RX 缓冲区寄存器，然后触发一个或多个下列事件：

- RX 控制寄存器内的 “RX 寄存器满 ” 位被置位，如果 RX 中断已经启用，则相关联的中断被触发。
- 如果停止位没有在数据流的预期位置被检测到，则 RX 控制寄存器内的成帧错误位被置位。
- 如果缓冲区寄存器没有在当前接收数据的停止位之前被读取，则 RX 控制寄存器内的 “过速错误 ” 位被置位。
- 如果检测到奇偶错误，则 RX 控制寄存器内的 “ 奇偶错误 ” 位被置位。

如果要通过轮询来检测已完全接收数据字节，则应当对 RX 控制寄存器内的 “RX 寄存器满 ” 位进行监测。 在下一个字节完整接收之前，必须从 RX 缓冲器寄存器内将数据读出，以防止出现过速错误的情况。

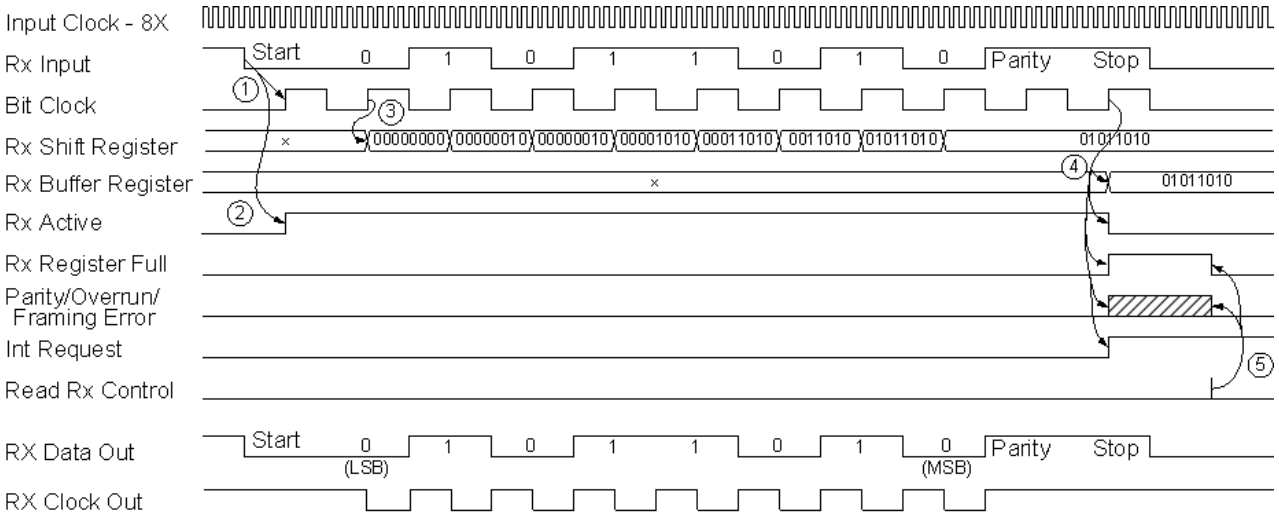
通信系统精度

当使用 PSoC 器件的 SLIMO 模式时，PSoC 系统时钟 (SysClk) 没有足够的精度来保证有效的 UART 通信。而且，当 PSoC 未连接到 USB 时，PSoC CY8C24x94 系列器件中的 SysClk 没有足够的精度来保证有效的 UART 通信。 系统错误（即通信链路两端的错误总数）必须小于 2%，以便 UART 通信能够正常工作。 有关 SysClk 精度的详细信息，请参见器件数据表。

时序

下面的 RX 时序图说明了 UART 用户模块 RX 组件的操作。

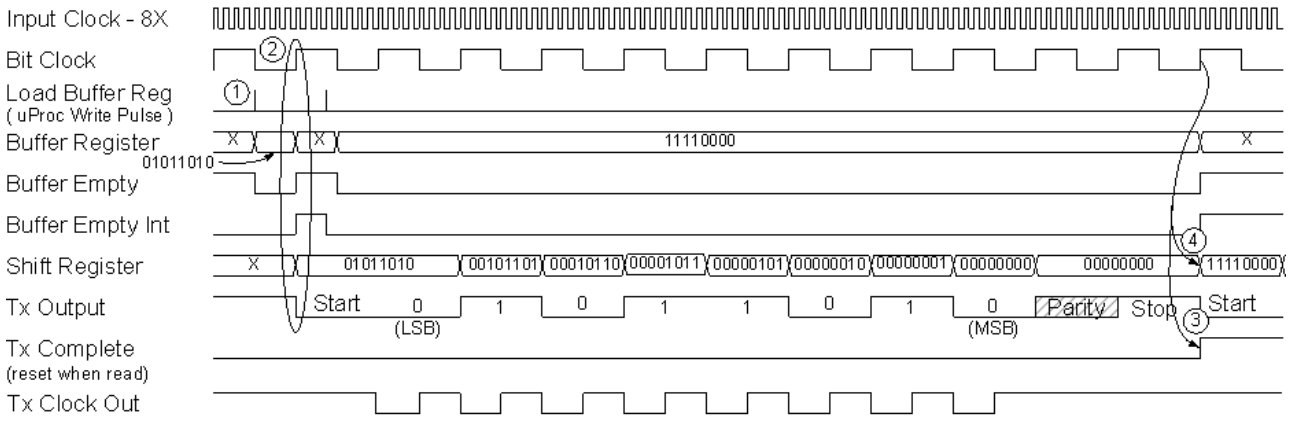
Figure 2. RX 时序图



[+] Feedback

下面的 TX 时序图说明了 UART 用户模块 TX 组件的操作。

Figure 3. TX 时序图



直流和交流电气特性

Table 1. UART 的直流和交流电气特性

参数	条件和注释	典型值	极限值	单位
F_{max}	最大字符传输频率		6	Mbits

放置

UART 用户模块可以置于任何两个数字通信模块内。 请注意，接收器和发送器组件使用了相同的时钟源。

参数和资源

时钟

UART 的时钟来自于 16 个可能来源之一。 全局 I/O 总线可以用于连接时钟输入至某个外部引脚或由不同 PSoC 模块所生成的时钟。 当模块使用外部数字时钟时，应将行输入同步关闭，以获得最佳精度以及进行睡眠操作。 48 MHz 时钟、CPU_32 kHz 时钟、分频后时钟中的任一个（24V1 或 24V2）或者另一个 PSoC 模块输出均可以指定为时钟输入。

时钟频率必须设置为所期望的位速率的 8 倍。 每 8 个时钟周期接收或发送 1 个数据位。 时钟容差必须为 ± 2% 才能保证其正常工作。

RX 输入

接收器的输入可以连接至固定低电平、高电平、邻近的 PSoC 模块、模拟比较器输出总线、或全局总线之一。 使用全局总线时，输入可以连接到外部引脚之一。

反相 RX 输入

此参数让用户对 RX 输入信号进行反相。

TX 输出

发送器的输出可以被连接至全局输出总线。 全局输出总线可以再连接至外部引脚或其它 PSoC 模块以执行下一步的处理。

[+] Feedback

TX 中断模式

此选项决定了何时为 TX 模块产生中断。“TxRegEmpty” 选项能够让中断在数据从数据寄存器传输至移位寄存器后立刻发生。 选择第 2 个选项 “TxComplete” 时，能够延迟中断直到最后位移出移位寄存器。 当有必要知道字符何时被完全传输出去时，第 2 个选项很有用。 第 1 个选项 “TxRegEmpty” 项最适合用于让发送器的输出最有效率。 此选项允许在前 1 个字节正在发送时加载 1 个字节。 在中断服务子程序中，TX_CONTROL_REG 应当被读取以启用后续中断。

ClockSync

在 PSoC 器件中，数据模块可以在系统时钟以外提供时钟源。 数字时钟源甚至可以用串行方式串联起来。 这样就会引起与系统时钟相关的时滞。 这些时滞对于 CY8C29/27/24/22/21xxx 和 CY8CLED04/08/16 PSoC 器件系列具有更为关键的意义，因为其中存在各种数据路径优化，特别是那些应用于系统总线的部分。 此参数可以用于控制时钟时滞并确保其在读取和写入 PSoC 模块寄存器数值时的正确运行。 此参数的正确数值应当由下表决定。

ClockSync 数值	使用
Sync to SysClk	当使用任何由 24 MHz (SysClk) 经过二分频或更多分频所衍生出来的时钟源，使用此设定。 这样的时钟源包括 VC1、VC2、VC3 （在 VC3 由 SysClk 驱动时）、32 KHz 和以 SysClk 作为时钟源的数字 PSoC 模块。 外部生成的时钟源也应当使用此数值来确保产生正确的同步。
Sync to SysClk*2	此设置可以适用于任何基于 48 MHz (SysClk*2) 的时钟，除非产生的频率为 48 MHz （换句话说，所有分频器的乘积为 1 时）。
Use SysClk Direct	在需要 24 MHz (SysClk/1) 时钟时使用。 此项选择并不真正执行同步，但提供了对系统时钟本身的低时滞访问方式。 如果选择此项，则此选项将覆盖上述时钟参数的设置。 在所有分频器组合起来最终生成了 24 MHz 的输出时，一定要使用此项，而不使用 VC1、VC2、VC3 或数字模块。
Unsynchronized	在选定 48 MHz (SysClk*2) 输入时使用。 在需要未同步输入时使用。 一般来说，只有在中断生成是计数器的唯一应用时才推荐使用此选项。

RX 输出

此参数允许输入信号被连接至某一个行总线上。 此信号与数据时钟输出 (Data Clock Out) 信号一起可以被用来实现数据验证功能，例如循环冗余校验。

RX 时钟输出

此参数允许来自 RX 模块的位时钟连接至某一个行总线上。位时钟即为时钟输入信号的 8 分频信号。数据时钟输出信号的上升沿恰好是数据已经稳定并可以被采样的时刻。 此信号与 RX 输出信号一起可以被用来实现数据验证功能，例如循环冗余校验。

TX 时钟输出

此参数允许来自 TX 模块的位时钟被连接到某一个行总线上。 位时钟即为时钟输入信号的 8 分频信号。 数据时钟输出信号的上升沿恰好是数据已经稳定并且可以被采样的时刻。 此信号与 TX 输出信号一起可以被用来实现数据验证功能，例如循环冗余校验。

RxCmdBuffer

此参数用于启用接收命令缓冲区和用于命令处理的固件。 必须启用 UART RX 中断后，命令缓冲区才能正常运行。

[+] Feedback

RxBufferSize

此参数用于确定为接收缓存区保留的 RAM 的大小。 可以接收的最大命令长度是所选定的缓冲区大小减 1，因为字符串必须以空字符作为结尾。 此参数只在 RxCmdBuffer 参数启用后以及 UART RX 中断启用后有效。

CommandTerminator

此参数用于选择作为命令结尾的字符。 在接收到此字符时，将对一个标志位置位，标志着已经接收到了一个完整的命令。 一旦此标志位置位，直到调用 cmdReset() 函数之前，都不会再接收其它字符。

Param_Delimiter

此参数选择用于在命令接收缓冲区内分隔命令和参数的字符。 例如，如果 Param_Delimiter 设置为空格字符 (32)，则每个由空格分隔开的子字符串都将被解析为一个参数。 假设字符串为 “cmd foo bar c”，则参数将是 “cmd”、“foo”、“bar” 和 “c”。 每次调用 szGetParam() 函数均会返回一个指向下一个子字符串的指针，这个子字符串是一个按照从左至右的次序放置的，以空字符结尾的字符串。

IgnoreCharsBelow

此参数将让低于某一设定数值以下的字符被接收缓冲区所忽略。 这些字符将被接收，但将不会添加到接收缓冲区内。 此参数只在 RxCmdBuffer 参数启用后以及 UART RX 中断处于活动状态下有效。

Enable_BackSpace

此参数启用一个退格或删除字符操作来删除 UART 接收缓冲器内的最后 1 个字符。 这 3 个可能的设置值为 “ 禁用 ”、“ 退格 ” 和 “ 删除 ”。 此参数只在 RxCmdBuffer 参数启用后以及 UART RX 中断处于活动状态下有效。

中断生成控制

以下两项参数 InterruptAPI 和 IntDispatchMode 只能在 PSoC Designer 内将 **“ 启用中断生成控制 ”** 复选框选中后进行访问。 此复选框位于 **“ 项目 ” > “ 设置 ” > “ 芯片编辑器 ” 之下。**

InterruptAPI

InterruptAPI 参数允许有条件地生成一个用户模块的中断处理程序和中断矢量表入口。 选择 “ 启用 ” 可生成中断处理程序和中断矢量表入口。 选择 “ 禁用 ” 可避免生成中断处理程序和中断矢量表入口。 如果要使用接收命令缓冲区，则 InterruptAPI 参数应当设置为 “ 启用 ”。 在那些拥有多个程序层而且有多个程序层使用同一模块资源的项目中，特别推荐要正确地选择是否要生成中断 API。 仅在必要时选择生成中断 API，这样可以避免生成中断调度代码，从而减少开销。

IntDispatchMode

IntDispatchMode 参数用于指定如何处理中断请求（当处于同一模块的多个用户模块在不同的程序层共享该中断时） 选择 “ActiveStatus” 会导致固件在为共享的中断请求提供服务之前测试哪一个程序层正处于活动状态。 这项测试发生在每次请求共享中断的时候。 这样会导致延迟增加并且会产生一个服务于共享中断请求的非确定性的程序，但并不要求任何 RAM 资源。 选择 “OffsetPreCalc” 参数会导致固件只在最初已经有一个程序层运行时计算共享中断请求的来源。 这项计算减少了中断延迟会产生一个服务于共享中断请求的确定性程序，但其代价是 1 个字节的 RAM 资源。

[+] Feedback

应用程序编程接口

应用程序编程接口 (API) 程序作为用户模块的一部分提供，使设计人员能够在较高的层级处理模块。 本部分具体指明了每个函数的接口以及由 “include” 文件所提供的常数。

Note 在这里，如同所有用户模块 API 中的一样，A 和 X 寄存器的值可能会通过调用 API 函数时而发生更改。 发起调用 API 的函数有责任在调用之前保留 A 和 X 的值 （如果调用后需要再次用到它们）。 选择这种 “ 寄存器易变 ” 策略是为了提高效率，并且自从 PsoC Designer 的 1.0 版本起使用。 C 语言编译器将自动满足此项要求。 汇编语言编程人员也必须确保自己的代码遵守这一策略。 尽管有些用户模块的 API 函数会保留 A 和 X 不变，但并不保证在未来也将如此。

下表列出了随 UART 提供的低层 API 函数：

Table 2. 低层 UART API

函数	说明
void UART_Start(BYTE bParity)	启用用户模块并设置奇偶校验。
void UART_Stop(void)	禁用用户模块。
void UART_EnableInt(void)	启用 RX 和 TX 中断。
void UART_DisableInt(void)	禁用 RX 和 TX 中断。
void UART_SetTxIntMode(BYTE bTxIntMode)	设置 TX 中断的源。
void UART_SendData(BYTE bTxData)	在不检查 TX 状态的情况下发送字节。
BYTE UART_bReadTxStatus(void)	返回 TX 状态寄存器的状态。
BYTE UART_bReadRxData(void)	在不检查字符状态是否有效的情况下，返回 RX 数据寄存器中的数据。
BYTE UART_bReadRxStatus(void)	检查 RX 状态寄存器的状态。

UART_Start

说明：

设置奇偶校验并启用 UART 接收器和发送器。 一旦启用，就可以接收和发送数据。

C 语言原型：

```
void  UART_Start(BYTE bParitySetting)
```

汇编语言：

```
mov   A, UART_PARITY_NONE
lcall  UART_Start
```

参数：

bParitySetting： 1 个字节用于指定发送的奇偶校验类型。 下表给出了在 C 语言程序和汇编语言程序内提供的符号名称及其相关数值。

[+] Feedback

TX 奇偶校验	值
UART_PARITY_NONE	0x00
UART_PARITY_EVEN	0x02
UART_PARITY_ODD	0x06

返回值：

无

副作用：

寄存器 A 和寄存器 X 有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_Stop

说明：

禁用 UART 模块。

C 语言原型：

```
void  UART_Stop(void)
```

汇编语言：

```
lcall  UART_Stop
```

参数：

无

返回值：

无

副作用：

寄存器 A 和寄存器 X 有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_EnableInt

说明：

通过设置适当的中断启用位来选择和启用两个 UART 中断。

C 语言原型：

```
void  UART_EnableInt(void)
```

汇编语言：

```
lcall  UART_EnableInt
```

参数：

无

返回值：

无

副作用：

如果某个中断正等待解决而且此 API 被调用，则将立刻触发一个此中断。 此调用应当在调用 Start() 之前执行。 寄存器 A 和寄存器 X 有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保留这些值。

UART_DisableInt

说明：

禁用所有 UART 中断。

C 语言原型：

```
void  UART_DisableInt(void)
```

汇编语言：

```
lcall  UART_DisableInt
```

参数：

无

返回值：

无

副作用：

寄存器 A 和寄存器 X 有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任将调用前后的数值保存在 fastcall16 函数内。

UART_SetTxIntMode

说明：

设置 TX 中断的源。

C 语言原型：

```
void  UART_SetTxIntMode(BYTE bTxIntMode)
```

汇编语言：

```
lcall  UART_SetTxIntMode
```

参数：

bTxIntMode：用于指定 TX 模块中断模式的 1 个字节。 下表给出了在 C 语言程序和汇编语言程序内提供的符号名称及其相关数值。

TX 中断模式	值
UART_INT_MODE_TX_REG_EMPTY	0x00
UART_INT_MODE_TX_COMPLETE	0x01

[+] Feedback

返回值：

无

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保留这些值。

UART_SendData

说明：

通过将指定用于发送的数据加载到 TX 缓冲区寄存器来启动数据发送操作。 TX 控制寄存器内的 “TX 完成 ” 状态位应当得到监测以确保发送操作已经启动。

C 语言原型：

```
void  UART_SendData(BYTE bTxData)
```

汇编语言：

```
mov   A, bTxData
lcall  UART_SendData
```

参数：

bTxData： 将要加载至 TX 缓冲区寄存器的数据，该参数将通过累加器传递。

返回值：

无

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保留这些值。

UART_bReadTxStatus

说明：

返回 TX 控制寄存器的内容。

C 语言原型：

```
BYTE  UART_bReadTxStatus(void)
```

汇编语言：

```
lcall  UART_bReadTxStatus
and   A, UART_TX_COMPLETE
jnz   TxIsComplete
```

参数：

无

返回值：

返回 TX 状态字节。 利用定义的掩码来测试特定的状态条件。 请注意，这些掩码可以通过 “ 或 ” 运算结合起来同时检测多个条件。

[+] Feedback

RX 状态屏蔽码	值
UART_TX_COMPLETE	0x20
UART_TX_BUFFER_EMPTY	0x10

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保留这些值。

UART_bReadRxData

说明：

读取从 RX 缓冲区寄存器接收的数据字节。

C 语言原型：

```
BYTE  UART_bReadRxData(void)
```

汇编语言：

```
lcall  UART_bReadRxData
mov   [bRxData],A
```

参数：

无

返回值：

接收到的数据字节，通过累加器传递。

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保留这些值。

UART_bReadRxStatus

说明：

读取并返回 RX 控制寄存器。

C 语言原型：

```
BYTE  UART_bReadRxStatus(void)
```

汇编语言：

```
lcall  UART_bReadRxStatus
push  A
and   A, UART_RX_COMPLETE
pop   A
jz    RxNotCompleted
cmp   A, UART_RX_ERROR              ; determine if RX was without error
jz    RxCompleteWithoutError        ; receive was completed
```


[+] Feedback

参数：

无

返回值：

返回 RX 状态字节。 使用以下定义的掩码来测试特定的状态条件。 请注意，这些掩码可以通过 “ 或 ” 运算结合起来检测组合条件。

RX 状态屏蔽码	值	说明 （在相应位被置位时）
UART_RX_REG_FULL	0x08	数据寄存器包含未经读取的数据
UART_RX_PARITY_ERROR	0x80	自上次读取状态后发生了奇偶错误
UART_RX_OVERRUN_ERROR	0x40	数据已经被覆写至缓冲区寄存器
UART_RX_FRAMING_ERROR	0x20	停止位为低
UART_RX_ERROR	0xE0	所有错误位一起 “ 或 ” 运算

副作用：

一次对此寄存器的读取操作会清空除了 UART_RX_REG_FULL 位以外的所有状态位，而 UART_RX_REG_FULL 只有在数据寄存器被直接读取或采用 API 函数进行读取后才会被清空。 应当在抛弃返回数值前谨慎检查所有适用的状态条件。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

高层 API

高层 API 能够在基本函数之上添加额外的固件以提供命令和字符串级的函数而不是字符级的函数。 器件编辑器允许用户设置接收器命令缓冲区的大小、命令终止符、参数分隔符以及接收器应当在哪一数值之下忽略字符。 发送函数能够接受指向字符串的指针，所以用 1 次函数调用就可以打印整个字符串，而无需添加 C 语言或汇编语言代码行。 发送函数不要求启用 UART 发送中断。

要使用这种高层接收器函数，可进入器件编辑器窗口，选择 “UART”，再对 “RxCmdBuffer” 参数选择 “ 启用 ” 选项。 下一步，选择一个大小足以容纳用户的最大命令字节数 +1 的 “RxBufferSize” 参数。 选择一个命令终止符 “CommandTerminator”。 此参数最经常设置为回车符 (13) 或换行符 (10)。 如果用户的命令包含了 2 个或更多参数，则可选择参数分隔符 “Param_Delimiter.”。常见命令分隔符通常为 1 个空格 (32) 或 1 个逗号 (44) 字符。 位于某个选定数值以下的控制字符也可以被忽略。 绝大多数控制字符均处于 0 至 31 的范围内。 设置 “IgnoreCharsBelow” 为 32 将忽视这些字符。 如果所有字符都有效，则可设置此参数为 “1”。 选定作为命令终止器 (CommandTerminator) 的字符不受 “IgnoreCharsBelow” 选项的影响。 以下流程图表明了命令缓冲区函数基本操作的正确顺序。

命令缓冲区用于在 UART RX 中断服务子程序内收集缓冲区内的字符，直至收到命令终止符为止。 此时，将设置 1 个标志作为命令缓冲区已准备好读取的信号。 接收缓冲区可以通过读取数组 INSTANCE_NAME_aRxBuffer 来直接读取，或通过使用 szGetParam() 或 szGetRestOfParams() 函数来读取。 如果所接收到的字符多于 buffer_size -1，则随后的字符将被忽略。

命令缓冲区将采集字符直到缓冲区满或检测到命令终止符。 在这两种条件之后所接收到的任何字符均将被忽略，直到执行了 CmdReset 命令为止。 一旦执行了 CmdReset 命令，RX ISR 固件将开始再一次采集字符。

[+] Feedback

Figure 4. 命令缓冲区流程图

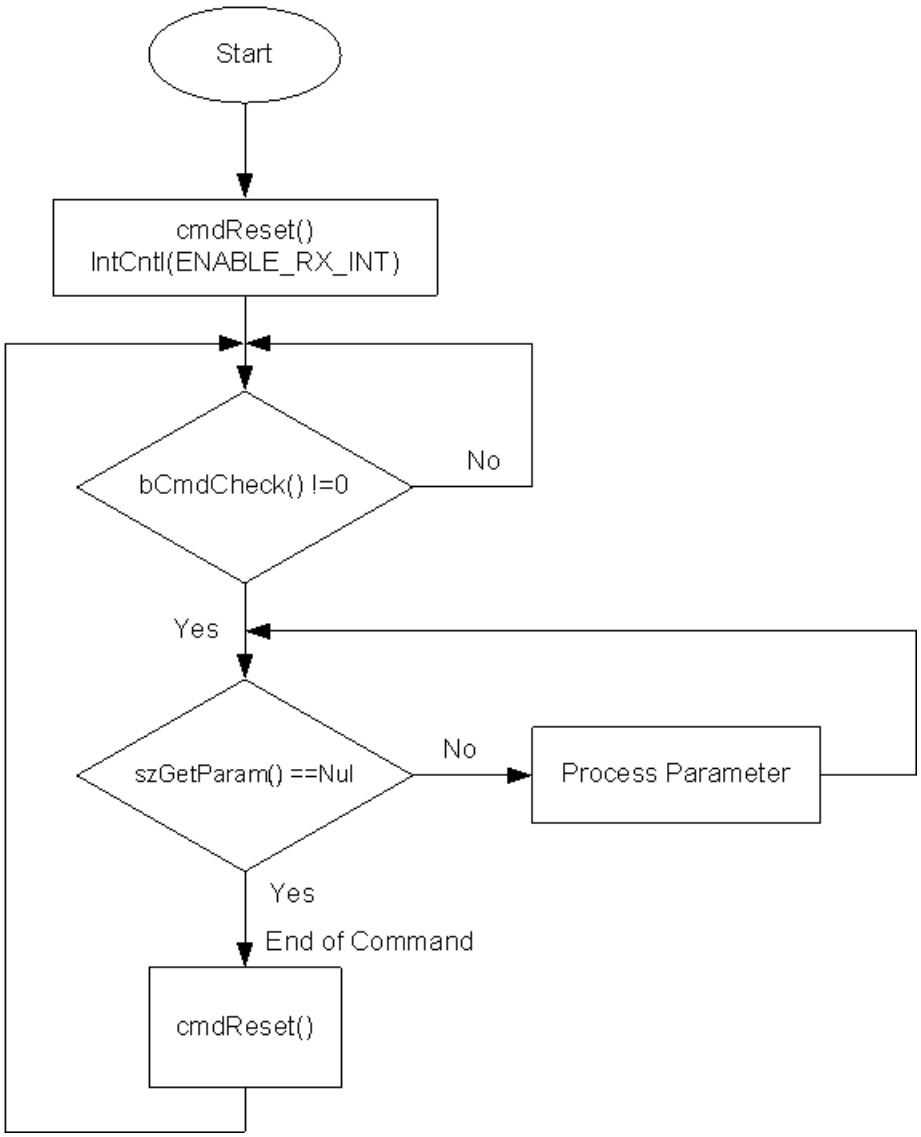


Table 3. 高层 UART API

函数	说明
void UART_IntCntl(BYTE bMask)	选择性启用 / 禁用 RX 和 TX 中断。
void UART_PutString(char * szStr)	从 TX 端口向外发送以空字符为结尾的字符串。
void UART_CPutString(const char * azStr)	从 TX 端口向外发送以空字符为结尾的常数 (ROM) 字符串。
void UART_PutChar(char bData)	在 TX 寄存器为空时发送字符至 TX 端口。 在 TX 数据寄存器可以被写入 （并且没有数据过速错误）之前，此函数将不会返回。
char UART_cGetChar(void)	在有效数据可用时，从 RX 数据寄存器内返回字符。 在字符接收到之前，此函数将不会返回。
void UART_Write(char * aStr, BYTE bCnt)	从 aStr 数组向 TX 端口发送 bCnt 字节。

[+] Feedback

函数	说明
void UART_CWrite(const char * aStr, int iCnt)	从常数 aStr 数组向 TX 端口发送 iCnt 字节。
char UART_cReadChar(void)	立即读取 RX 数据寄存器。 如果没有有效的数据，则返回 0，否则返回 1 至 255 之间的 ASCII 字符。
int UART_ iReadChar(void)	立即读取 RX 数据寄存器。 如果没有可用数据或存在错误状况，则在最高有效位 (MSB) 内返回一个错误状态。 所接收到的字符在最低有效位 (LSB) 内返回。
void UART_PutSHexByte(BYTE bValue)	发送 bValue 的 2 字符 16 进制表示形式的数据至 TX 端口。
void UART_PutSHexInt(int iValue)	发送 iValue 的 4 字符 16 进制表示形式的数据至 TX 端口。
void UART_PutCRLF(void)	向 TX 端口发送回车符 (0x0D) 和换行符 (0x0A)。
void UART_CmdReset(void)	复位 RX 命令缓冲区。
BYTE UART_bCmdCheck(void)	如果收到了有效的命令终止符，则返回一个非零数值。
BYTE UART_bCmdLength(void)	返回当前命令长度。
char * UART_szGetParam(void)	返回指向 RX 缓冲区内下一个参数的指针。
char * UART_szGetRestOfParams(void)	返回指向剩余参数字符串的指针。
BYTE UART_bErrCheck(void)	返回命令缓冲区错误状态。

UART_IntCntl

说明：

独立地启用 / 禁用 RX 和 TX 中断。

C 语言原型：

```
void  UART_IntCntl(BYTE bMask)
```

汇编语言：

```
mov   A,(UART_ENABLE_RX_INT|UART_ENABLE_TX_INT)
lcall  UART_IntCntl
```

参数：

BYTE bMask： 用于启用或禁用 TX 和 RX 模块的屏蔽码。

屏蔽码	说明
UART_ENABLE_RX_INT	启用接收器
UART_ENABLE_TX_INT	启用发送器
UART_DISABLE_RX_INT	禁用接收器
UART_DISABLE_TX_INT	禁用发送器

返回值：

无

[+] Feedback

副作用：

如果在中断启用时，TX 或 RX 上有一个待处理的中断，则此中断会立即发生。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_PutString

说明：

向 UART TX 发送以空字符结尾的 (RAM) 字符串。

C 语言原型：

```
void UART_PutString(char * szRamString)
```

汇编程序：

```
mov  A,>szRamString     ; Load MSB part of pointer to RAM based null
                        ; terminated string.
mov  X,<szRamString     ; Load LSB part of pointer to RAM based null
                        ; terminated string.
lcall UART_PutString     ; Call function to send string out UART TX port
```

参数：

char * aRamString: 指向将发送至 UART TX 的字符串的指针。 MSB 在累加器内传递，而 LSB 在 X 寄存器内传递。

返回值：

无

副作用：

程序的执行停留在此函数内，直到最后 1 个字符加载到 UART 发送缓冲区之内。 寄存器 A 和寄存器 X 有可能在此函数的当前版本中或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 IDX_PP 页面指针寄存器被修改。

UART_CPutString

说明：

从 UART TX 端口向外发送空字符结尾的常数 (ROM) 字符串。

C 语言原型：

```
void UART_CPutString(const char * szROMString)
```

汇编程序：

```
mov  A,>szRomString   ; Load MSB part of pointer to ROM based null
                      ; terminated string.
mov  X,<szRomString   ; Load LSB part of pointer to ROM based null
                      ; terminated string.
lcall UART_CPutString  ; Call function to send constant string out
                      ; UART TX
```


[+] Feedback

参数：

const char * szROMString：指向将要发送至 UART 的字符串的指针。字符串指针的 MSB 在累加器内传递，指针的 LSB 在 X 寄存器内传递。

返回值：

无

副作用：

程序执行停留在此函数内，直到最后 1 个字符加载到 UART 发送缓冲区之内。寄存器 A 和寄存器 X 有可能在此函数的当前版本中或后续版本中被修改。在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_PutChar

说明：

在端口缓冲区为空时，写入单个字符至 UART TX 端口。

C 语言原型：

```
void UART_PutChar(CHAR cData)
```

汇编程序：

```
mov  A,0x33            ; Load ASCII character “3" in A
lcall UART_PutChar      ; Call function to send single character to
                       ; UART TX port.
```

参数：

CHAR cData：将要发送至 UART TX 端口的字符。数据在累加器内传递。

返回值：

无

副作用：

在数据可以写入至 UART TX 缓冲区之前，程序执行将停留在此函数之内。寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_cGetChar

说明：

等待 UART RX 内的有效字符并返回其数值。

C 语言原型：

```
CHAR UART_cGetChar(void)
```

汇编程序：

```
lcall UART_cGetChar         ; Call function to print single character to
                           ; serial port.
mov  [CharBuffer,]A        ; Store retrieved character in buffer
```


[+] Feedback

参数：

无

返回值：

Char bData： 从 UART RX 中读取的字符在累加器内被返回。

副作用：

程序执行停留在此函数内，直到收到某个字符或遇到某个以前接收但未读取的字符。 在使用此函数时，UART RX 中断应禁用。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_Write

说明：

向 UART TX 端口发送 “n” 个字符 (RAM)。

C 语言原型：

```
void UART_Write(char * szRamString, BYTE bCount)
```

汇编程序：

```
mov   A,20              ; Load string/array count
push  A
mov   A,>szRamString     ; Load MSB part of pointer to RAM string
push  A
mov   A,<szRamString     ; Load LSB part of pointer to RAM string
push  A
mov   X,SP              ; Set X register to point to variables
dec   X
lcall  UART_Write        ; Make call to function
add   SP,253            ; Reset stack pointer to original position
```

参数：

CHAR * szRamString： 指向将发送至 UART TX 的字符串的指针。

BYTE bCount： 将要发送至 UART TX 的字符数。

返回值：

无

副作用：

程序执行保留在此函数内，直到最后 1 个字符加载到 UART TX 缓冲区之内。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 IDX_PP 页面指针寄存器被修改。

UART_CWrite

说明：

向 UART TX 端口发送 n 个字符 (ROM)。

C 语言原型：

```
void UART_CWrite(const char * szRomString, INT iCount)
```


[+] Feedback

汇编程序：

```
mov   A,0               ; Load MSB of string/array count
push  A
mov   A,20              ; Load LSB of string/array count
push  A
mov   A,>szRomString     ; Load MSB part of pointer to ROM string
push  A
mov   A,<szRomString     ; Load LSB part of pointer to ROM string
push  A
mov   X,SP              ; Set X register to point to variables
dec   X
lcall  UART_CWrite       ; Make call to function
add   SP,252            ; Reset stack pointer to original position
```

参数：

const char * szRomString： 指向将要发送至 UART TX 端口的字符串的指针。

int iCount： 将要发送至 UART TX 端口的字符数。

返回值：

无

副作用：

程序执行保留在此函数内，直到最后 1 个字符加载到 UART TX 缓冲区之内。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_cReadChar

说明：

如果数据不可用或存在错误状况，或者返回了零，则立即读取 UART RX 端口；否则，读取并返回字符。

C 语言原型：

```
CHAR UART_cReadChar(void)
```

汇编程序：

```
lcall UART_cReadChar          ; Call function to read a character
cmp  A,0x00                  ; Check for error
jz   ProcessError            ; If error, Process the error condition
mov  [CharBuffer],A          ; Store retrieved character in buffer
```

参数：

无

返回值：

CHAR bData： 从 UART RX 端口读取的字符。 1 至 255 之间的 ASCII 字符为有效字符。 返回零则表示出现错误状况或数据不可用。

副作用：

函数只接受 1 至 255 之间的有效字符。 检测到 0x00 (null) 字符时即认为是一种错误状况。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx

[+] Feedback

和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_iReadChar

说明：

立即读取 UART RX 端口，返回所接收到的字符和错误状况。

C 语言原型：

```
INT UART_iReadChar(void)
```

汇编程序：

```
lcall UART_iReadChar          ; Call function to read a character
cmp  X,0x00                  ; Check for error
jnz  ProcessError            ; If error, Process the error condition
mov  [CharBuffer]A           ; Store retrieved character in buffer
```

参数：

无

返回值：

unsigned int iData：MSB 包含状态，而 LSB 包含 UART RX 数据。 如果 MSB 为非零，则发生一项错误。 下表显示了可能在 MSB 内返回的错误代码：

错误标志	值	说明
UART_RX_PARITY_ERROR	0x80	奇偶校验错误
UART_RX_OVERRUN_ERROR	0x40	缓冲区过速错误
UART_RX_FRAMING_ERROR	0x20	字符成帧错误
UART_RX_NO_DATA	0x10	RX 缓冲区满

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_PutSHexByte

说明：

发送 2 个字节的 ASCII 16 进制表示形式的数据至 UART TX 端口。

C 语言原型：

```
void UART_PutSHexByte(BYTE bData)
```

汇编程序：

```
mov  A,0x33               ; Load data to be sent to UART TX
lcall UART_PutSHexByte     ; Call function to output hex
                          ; representation of data.  The output
                          ; for this value would be “33".
```


[+] Feedback

参数：

BYTE bData： 将要转换为 ASCII 字符串的字节。

返回值：

无

副作用：

程序执行停留在此函数内，直到最后 1 个字符加载到 UART 发送缓冲区之内。 寄存器 A 和寄存器 X 有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_PutSHexInt

说明：

发送 4 个字节的 ASCII 16 进制表示形式的数据至 UART TX 端口。

C 语言原型：

```
void UART_PutSHexInt(INT iData)
```

汇编程序：

```
mov  A,0x34           ; Load LSB in A
mov  X,0x12           ; Load MSB in X

lcall UART_PutSHexInt  ; Call function to output hex
                       ; representation of data.  The output
                       ; for this value would be “1234".
```

参数：

int iData： 将要转换为 ASCII 字符串的整数。

返回值：

无

副作用：

程序执行停留在此函数内，直到最后 1 个字符加载到 UART 发送缓冲区之内。 寄存器 A 和寄存器 X 有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_PutCRLF

说明：

从 UART TX 端口向外发送回车符 (0x0D) 和换行符 (0x0A) 的函数。

C 语言原型：

```
void UART_PutCRLF(void)
```

汇编程序：

```
lcall UART_PutCRLF          ; Send a carriage return and line feed out TX
```


[+] Feedback

参数：

无

返回值：

无

副作用：

程序执行会停留在此函数中，直到所有字符均已写入 UART TX 缓冲区。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。

UART_CmdReset

说明：

复位命令缓冲区和标志。 这样可以让下一个命令的字符得到接收。

C 语言原型：

```
void UART_CmdReset(void)
```

汇编程序：

```
lcall UART_CmdReset         ; Call function to reset command buffer.
```

参数：

无

返回值：

无

副作用：

复位 UART 字符计数并清空接收缓冲区。 仍然保留在缓冲区内的所有字符均将丢失。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 CUR_PP 页面指针寄存器被修改。

UART_bCmdCheck

说明：

检查是否已经接收到命令终止符。

C 语言原型：

```
BYTE UART_bCmdCheck(void)
```

汇编程序：

```
lcall UART_bCmdCheck      ; Call function to get command complete status.
cmp  A,0x00               ; Check if command complete
jnz  ProcessCmd           ; Process command buffer
```

参数：

无

[+] Feedback

返回值：

如果已经接收到命令终止符，则将返回一个非零值。

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 CUR_PP 页面指针寄存器被修改。

UART_bCmdLength

说明：

返回命令缓冲区内的字符长度。 此命令将返回当前命令长度，无论当前是否已经接收到一个命令终止符。

C 语言原型：

```
BYTE UART_bCmdLength(void)
```

汇编程序：

```
lcall UART_bCmdLength      ; Call function to get current command
                          ; Command length is returned in Accumulator
```

参数：

无

返回值：

接收缓冲区内当前字符串的长度。

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 CUR_PP 页面指针寄存器被修改。

UART_szGetParam

说明：

从接收缓冲区返回下一个参数，此参数由在器件编辑器内所设置的参数分隔符 (Param_Delimiter) 来分隔。 在所有参数均已经返回后，任何后续调用都将返回一个空指针 (zero)。

C 语言原型：

```
char * UART_szGetParam(void)
```

汇编程序：

```
lcall UART_szGetParam       ; Call function to return pointer to the
                           ; next parameter.
                           ; Pointer is returned in A and X.
```

参数：

无

返回值：

char * strPtr： 指向参数字符串的指针。

[+] Feedback

副作用：

接收缓冲区在每次 szGetParam 被调用时都会被修改。 在每个参数后均放置空字符。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，CUR_PP 和 IDX_PP 页面指针寄存器已发生更改。

UART_szGetRestOfParams

说明：

返回一个指向接收缓冲区内尚未由 szGetParam 函数取走的剩余字符串的指针。 如果在 szGetParam 之前调用此函数，则返回一个指向整个接收缓冲区的指针。 如果到达了这个字符串的结尾，则返回一个空字符 (0x00)。

C 语言原型：

```
char * UART_szGetRestOfParams(void)
```

汇编程序：

```
lcall UART_szGetRestOfParams    ; Call function to return pointer to the
                               ; remainder of the receive buffer.
                               ; Pointer is returned in A and X.
```

参数：

无

返回值：

char * strPtr： 指向所接收字符串剩余部分的指针。

副作用：

寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下 （CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 CUR_PP 页面指针寄存器被修改。

UART_bErrCheck

说明：

检查自从上一次调用 bErrCheck 函数后的命令错误。

C 语言原型：

```
BYTE UART_bErrCheck(void)
```

汇编程序：

```
lcall UART_bErrCheck             ; Call function to return error status
cmp  A,0x00                     ; Check for error
jnz  ProcessError               ; If error, Process the error condition
```

参数：

无

返回值：

BYTE bErr： MSB 包含了自从上一次调用本函数以来所发生的任何错误状况的状态。

[+] Feedback

错误标志	值	说明
UART_RX_PARITY_ERROR	0x80	奇偶校验错误
UART_RX_OVERRUN_ERROR	0x40	缓冲区过速错误
UART_RX_FRAMING_ERROR	0x20	字符成帧错误
UART_RX_BUF_OVERRUN	0x10	软件 RX 缓冲区过速

副作用：

错误状态被清空。 寄存器 A 和寄存器 X 均有可能在此函数的当前版本或后续版本中被修改。 在大内存模式下（CY8C29xxx 和 CY8CLED16），所有 RAM 页面指针寄存器也会出现这种状况。 在必要时，调用函数有责任在调用 fastcall16 函数之前保存这些值。 目前，只有 CUR_PP 页面指针寄存器被修改。

示例固件源代码

以下是 UART 用户模块的汇编语言代码范例：

```
;***************************************************************
;  LoopBack:
;
;  This code sample illustrates how to setup a UART loopback.
;  Data received is echoed back to the receiver.  If an error
;  is received, then a NAK is returned.
;
;  This sample is written to be performed in the non-interrupt
;  processing.  This code could easily be written to be
;  interrupt driven.
;  In this example a Counter8 user module is used to generate
;  this clock.  The following are the parameter settings for the
;  Counter8 User Module used for this example.
;  Counter8  (placed in DBA03)
;  Clock         48MHz
;  Enable        High
;  Output        None
;  Period        155
;  CompareValue  78
;  CompareType   Less Than Or Equal
;  InterruptType Terminal Count
;
;  Using the settings above the UART baud rate can be calculated.
;  baud rate = 48MHz / ((Period+1) * (over Sample rate))
;  = 48MHz / ((155+1) * 8)
;  = 38.4 kBaud
;
;  The settings for the UART are as follows:
;
;  UART  (Placement RX => DAC06, TX => DCA07)
;  Clock              DBA03 Broadcast
;  RX Input           Global_IN_5
;  TX Output          Global_OUT_7
;  TX InterruptMode   TXComplete
```

```
;  RxCmdBuffer        Enable
;  RxBufferSize       16 Bytes
;  CommandTerminator  13      (CR)
;  Param_Delimiter    32      (space)
;  IgnoreCharsBelow   32      (Ignore all chars below space, except ;  CR)
;
;
;  First the UART and Counter8 user modules are initialized.  The
;  period and compare values for the Counter8 User Module are set to
;  generate the 8X clock.  If the period and compare values for
;  the Counter8 User Module are set in the parameter window, it is
;  not required to set them in the user program.
;
;*****************************************************************
include "m8c.inc"       ; part specific constants and macros
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules

export  _main
NAK_RESPONSE:   equ   00
_main:

    mov   A,UART_PARITY_NONE  ; No parity
    call  UART_Start
    call  Counter8_Start
.WaitForData:               ; wait for data to be received
    call  UART_bReadRxStatus
    and   A, UART_RX_COMPLETE
    jz    .WaitForData

    and   A, UART_RX_ERROR ; data received - see if data is valid
    jz    .GetData

    mov   A, NAK_RESPONSE  ; error detected setup to send a NAK
    jmp   .TxData
.GetData:

    call  UART_bReadRxData  ; read the data from the receiver
.TxData:

    call  UART_SendData     ; transmit the response data
    jmp   .WaitForData      ; go wait for next byte
```

以下是 UART 用户模块的 C 代码范例：

```
//-------------------------------------------------------------------
// Description:
//    This example program shows how easy it is to use the PSoC
//    serial port with this user module.
//
//    The UART requires a clock that is 8 times the desired Baud Rate.
//    In this example a Counter8 user module is used to generate
//    this clock.  The following are the parameter settings for the
//    Counter8 User Module used for this example.
//
//    Counter8  (placed in DBA03)
//      Clock         48MHz
//      Enable        High
//      Output        None
//      Period        155
//      CompareValue  78
//      CompareType   Less Than Or Equal
//      InterruptType Terminal Count
//
//    Using the settings above the UART baud rate can be calculated.
//    baud rate = 48MHz / ((Period+1) * (over Sample rate))
//              = 48MHz / ((155+1) * 8)
//              = 38.4 kBaud
//
//    The settings for the UART are as follows:
//
//    UART  (Placement RX => DAC06, TX => DCA07)
//      Clock              DBA03 Broadcast
//      RX Input           Global_IN_5
//      TX Output          Global_OUT_7
//      TX InterruptMode   TXComplete
//      RxCmdBuffer        Enable
//      RxBufferSize       16 Bytes
//      CommandTerminator  13      (CR)
//      Param_Delimiter    32      (space)
//      IgnoreCharsBelow   32      (Ignore all chars below space, except CR)
//
//
//    First the UART and Counter8 user modules are initialized.  The
//    period and compare values for the Counter8 User Module are set to
//    generate the 8X clock.  If the period and compare values for
//    the Counter8 User Module are set in the parameter window, it is
//    not required to set them in the user program, but are set here
//    to make it clear how the UART clock is generated. Next a
//    sample string is printed to welcome you to the program.  The
//    program then sits and waits for characters until a command
//    terminator is received (CR).  Once the command terminator is received
//    it parses the parameters and outputs them to the UART TX port.
//    The command buffer is then reset and waits for the next command.
//
//  Example:
//  Input:
//             "foobar aa bbb cc"(CR)
//
```


[+] Feedback

```
//  Output:
//
//    Welcome to PSoC UARTlus test program. V1.1
//    Found valid command
//    Command =>foobar<
//    Parameters:
//      <aa>
//      <bbb>
//      <c>
//
//-------------------------------------------------------------------
#include <m8c.h>
#include "PSoCAPI.h"

void main(void)
{

    char * strPtr;                      // Parameter pointer

    UART_CmdReset();                    // Initialize receiver/cmd
                                        // buffer
    UART_IntCntl(UART_ENABLE_RX_INT);   // Enable RX interrupts

    Counter8_WritePeriod(155);          // Set up baud rate generator
    Counter8_WriteCompareValue(77);
    Counter8_Start();                   // Turn on baud rate generator

    UART_Start(UART_PARITY_NONE);       // Enable UART

    M8C_EnableGInt ;                    // Turn on interrupts

    UART_CPutString("\r\nWelcome to PSoC UART test program. V1.1 \r\n");

    while(1) {
       if(UART_bCmdCheck()) {                   // Wait for command
          if(strPtr = UART_szGetParam()) {      // More than delimiter"
             UART_CPutString("Found valid command\r\nCommand =>");
             UART_PutString(strPtr);            // Print out command
             UART_CPutString("<\r\nParamaters:\r\n");
             while(strPtr = UART_szGetParam()) { // loop on each parameter
                UART_CPutString("   <");
                UART_PutString(strPtr);         // Print each parameter
                UART_CPutString(">\r\n");
             }
          }
       UART_CmdReset();                         // Reset command buffer
       }
   }
}
```


[+] Feedback

寄存器配置

本节描述了 UART 用户模块所使用或修改的 PSoC 模块和其他系统寄存器。 只对经过参数化的符号进行了解释。

Table 4. RX 模块，数据移位寄存器： DR0

位	7	6	5	4	3	2	1	0
值	RX 移位寄存器							

RX 移位寄存器： 在输入端检测到起始位时，RX 状态机的硬件生成一个用于将数据移入此寄存器的 8 分频位时钟。

Table 5. RX 模块，数据寄存器： DR1

位	7	6	5	4	3	2	1	0
值	0	0	0	0	0	0	0	0

未使用该寄存器。

Table 6. RX 模块，数据缓冲区寄存器： DR2

位	7	6	5	4	3	2	1	0
值	RX 缓冲区寄存器							

RX 缓冲区寄存器： 在采样到停止位后，数据从 RX 移位寄存器传输到本寄存器。

Table 7. 模块 RX，控制寄存器： CR0

位	7	6	5	4	3	2	1	0
值	奇偶校验错误	过速错误	成帧错误	RX 活动	RX 寄存器满	奇偶校验类型	奇偶校验启用	RX 启用

奇偶错误 (Parity Error) 是用于指示所接收数据字节的奇偶计算结果的标志。 过速错误 (Overrun Error) 是用于指示 RX 缓冲区寄存器数据已被覆写的标志。 成帧错误 (Framing Error) 是用于指示停止位已经正确接收的标志。 RX 活动 (Rx Active) 是用于指示某个数据字节是否正在被接收的标志。 RX 寄存器满 (Rx Reg Full) 是用于指示某个数据字节已经完整接收，此数据字节已经传输至 RX 缓冲区寄存器及错误状况有效的标志。 奇偶校验类型 (Parity Type) 是要计算的奇偶校验类型。 如果奇偶校验未被启用，则此位无需被关注 奇偶校验启用 (Parity Enable) 用于启用或禁用所接收奇偶校验位的计算。 奇偶校验由 “ 奇偶校验类型 ” 位的设置值进行选择。 RX 启用 (Rx Enable) 用于启用或禁用 RX8 接收器。

读取 RX 控制寄存器可以将状态数据置于数据总线上，并清除所有状态位。 当检测到奇偶校验错误或溢出错误时，固件不需要执行任何操作。 如果检测到成帧错误，成帧器会立即开始寻找下一个数据字节。 假如系统内的 RX 输入有可能长时间卡在逻辑 0 处，则应停止接收器，并且持续查询该信号线直至其返回至逻辑 1 状态，然后才可以再次启用接收器。 如果 RX 输入卡在逻辑 0 处，这样处理可以避免当输入信号线为逻辑 0 时重复检测到 “ 虚假 ” 起始位，从而导致成帧错误 （在应该检测到起始位的位置检测到逻辑 0）

Table 8. RX 模块，寄存器： 函数

位	7	6	5	4	3	2	1	0
值	0	0	0	0	0	1	0	1

此寄存器定义将这个 “A” 型数字通信模块配置为接收器。

[+] Feedback

Table 9. RX 模块，寄存器： 输入

位	7	6	5	4	3	2	1	0
值	RX 输入				时钟源			

RX 输入 (RX Input) 用于选择 RX 输入源。 时钟源 (Clock Source) 用于选择驱动接收器时序的时钟。

Table 10. RX 模块，寄存器： 输出

位	7	6	5	4	3	2	1	0
值	0	0	0	0	0	0	0	0

Table 11. TX 模块，数据移位寄存器： DR0

位	7	6	5	4	3	2	1	0
值	TX 移位寄存器							

TX 移位寄存器由 TX 状态机硬件控制，用于移位内容以及发送最低有效位。 数据由 TX 状态机硬件从 DR1 数据寄存器加载到此寄存器内。

Table 12. TX 模块，数据缓冲区寄存器： DR1

位	7	6	5	4	3	2	1	0
值	TX 缓冲区寄存器							

TX 缓冲区寄存器用于发送已经由用户模块 API 写入到此寄存器的数据。 加载到这个寄存器内的数据通过 TX 状态机传输至 TX 移位寄存器。

Table 13. TX 模块，数据寄存器： DR2

位	7	6	5	4	3	2	1	0
值	0	0	0	0	0	0	0	0

未使用该寄存器。

Table 14. TX 模块，控制 / 状态寄存器： CR0

位	7	6	5	4	3	2	1	0
值	Rsvd	Rsvd	TX 完成	TX 寄存器空	Rsvd	奇偶校验类型	奇偶校验启用	TX 启用

TX 完成 (Tx Complete) 是用于指示某个数据字节是否正在发送过程中的标志。 此位在寄存器被读取时复位。 TX 寄存器空 (Tx Reg Empty) 是用于指示缓冲区寄存器为空的标志。 奇偶校验类型 (Parity Type) 表示将要计算的奇偶校验的类型。 如果奇偶校验未启用，则无需关注此位。 奇偶校验启用 (Parity Enable) 用于启用或禁用数据字节的奇偶校验位的计算和发送。 奇偶校验由 “ 奇偶校验类型 ” 位的设置值进行选择。 TX 启用 (Tx Enable) 用于启用或禁用 TX 发送器。

[+] Feedback

Table 15. TX 模块，寄存器： 函数

位	7	6	5	4	3	2	1	0
值	0	0	0	0	1	1	0	1

此寄存器定义将此 “A” 型数字通信模块配置为发送器。

Table 16. TX 模块，寄存器： 输入

位	7	6	5	4	3	2	1	0
值	0	0	0	0	时钟源			

时钟源 (Clock Source) 用于选择驱动发送器时序的时钟。

Table 17. TX 模块，寄存器： 输出

位	7	6	5	4	3	2	1	0
值	0	0	0	0	0	TX 输出启用和选择		

TX 输出启用和选择 (TX Output Enable and Select) 用于指定 TX8 的输出。

版本历史

版本	创建者	说明
5.3	DHA	添加了大内存模式芯片的兼容性

Note PSoC Designer 5.1 在所有用户模块数据表中引入了版本历史。 此部分记录了当前用户模块版本和以前用户模块版本之间区别的高级描述。

Copyright © 2001-2010 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.