DRIVING THE NXP SC28L91 UART

This document is one of several articles addressing the adaptation of an NXP® IMPACT™ universal asynchronous receiver/transmitter (UART) to a computer system powered by a Western Design Center 65C02 or 65C816 microprocessor. This information is intended to supplement, not replace, the information in the applicable NXP data sheet.

Reasonable assumptions have been made about your ability to understand technical computer terminology. If there is something you don't understand please seek help from a knowledgeable friend, instructor, co-worker, etc. A recommended on-line resource for 6502 family hardware and programming is 6502.org (http://6502.org). If you are a 6502.org member you can post queries in the forum on subjects you don't understand and will likely receive expert help.

NXP's IMPACT line consists of UARTs with one, two, four and eight communications channels, all sharing a similar architecture. These devices have been targeted to industrial and automotive environments, and thus have design characteristics intended to promote high performance and reliable operation under all reasonable conditions. As the title suggests, this article focuses on developing device driver software for NXP's SC28L91 ("28L91") single-channel UART when installed in a 65C02 or 65C816 computer system. If you are also interested in the hardware aspects of the 28L91 be sure to read our article *INTERFACING THE NXP SC28L91 UART*.

Code examples will follow the MOS Technology 6502 assembly language standard as promulgated by the Western Design Center for the 65C02 and 65C816. Please note that all references to the 65C816 will assume native mode operation. If the 65C816 is being operated in emulation mode it must be treated as a 65C02 for programming purposes.[A] As the 65C816 has more expansive resources than the 65C02, separate code examples illustrating a particular algorithm may be presented for each microprocessor.

[A] The emulation mode of the 65C816 does not make it behave exactly like the 65C02. All 65C816-unique instructions are usable in emulation mode to varying degrees, and the BBR, BBS, RMB and SMB instructions have not been implemented—their opcodes have been reassigned. Code examples for the 65C02 will not use any instructions that cannot be used by the 65C816 while in emulation mode.

TABLE OF CONTENTS

1 GLOSSARY

The following definitions will be found throughout this text.

8N1 The most commonly used data format in TIA-232 circuits, meaning 8 data bits, No parity and 1 stop bit. A total of 10 bits are transferred per datum (see below) when using 8N1 format, they being a start bit, stop bit and eight data bits. Hence the maximum possible effective speed in datums per second of a serial interface using this format is one-tenth the bits-per-second rate.

API Application programming interface. This term is used to describe the means by which programs make use of an operating environment's services. In the case of the UART driver, its API is the means by which driver services are accessed.

Baud The signaling rate used on an interface, in symbols per second. Baud is not always the same as the actual data rate—baud usually describes the communication speed between modems.

bps Bits per second, which is the actual data rate on the interface.

Bps Bytes per second.

C/T Counter/timer, a hardware feature of the 28L91.

CTS Clear-to-send. A 28L91 hardware input that is controlled by the remote device to tell the UART when it is permissible to transmit a datum. CTS is used in hardware flow control.

Datum A singular quantity of data when discussing the UART. In serial data transmission/reception, a datum is not always equivalent to a byte due to differing formats that may be used, for example, seven bits, even parity and one stop bit (referred to as 7E1 format). Hence the use of "datum" rather than "byte." When discussing the processing of a datum by the microprocessor one may mentally substitute "byte."

The term "datums" is a plural that will be used when speaking of a datum as an entity. "Data" is also the plural of "datum," but is used to refer to a *stream* of "datums."

DP Direct page. Page zero (see ZP) when programming the 65C816.

FIFO An acronym that means "first-in, first-out." A FIFO is a data storage and retrieval structure that is sequentially accessed one byte at a time. As usually implemented in memory (RAM), a FIFO is structured as a circular queue.

IRQ Interrupt request.

ISR Interrupt service routine or interrupt handler.

Kbps Kilobits per second.

KBps Kilobytes per second

LSB Least significant byte or bit, usually clear from context.

LSW Least significant word, where "word" means a 16-bit quantity.

MPU Microprocessor. The acronym and/or the word generically refer to both the 65C02 and 65C816.

MSB Most significant byte or bit, usually clear from context.

MSW Most significant word, where "word" means a 16-bit quantity.

Ø2 Symbol for the clock signal that drives the MPU.

RHR Receiver holding register. The RHR is a FIFO in the 28L91 that queues up a maximum of 16 datums that have been received.

RSR Receiver shift register. The RSR is the part of the UART's receiver that converts an incoming sequence of bits into a datum, a process referred to as "de-serialization."

RTS Request-to-send. A hardware output that the 28L91 controls to tell the remote device when it is permissible to transmit a datum. RTS is used in hardware flow control.

RxD The UART's receiver, which collectively refers to the RHR and the RSR.

RxQ Receive circular queue (see FIFO on page 3). Datums that have been read by the MPU from the UART are put into the RxQ pending further processing.

THR Transmitter holding register. The THR is a FIFO in the 28L91 that queues up a maximum of 16 datums that are to be transmitted.

TSR Transmitter shift register. The TSR is the part of the UART's transmitter that converts an outgoing datum into a sequence of bits, a process referred to as "serialization."

TxD The UART's transmitter, which collectively refers to the THR and the TSR.

TxQ Transmitter circular queue (see FIFO on page 3). Datums that are to be transmitted are put into the TxQ to be subsequently written by the MPU to the UART.

ZP Zero page, also page zero. Zero page is the address range from `$00` to `$FF` inclusive. It is referred to as "direct page" (DP) when programming the 65C816, as the latter is able to "see" zero page anywhere in the first 64 kilobytes of memory.

65C02 registers will be referenced in the text and code examples as follows:

SYMBOL	REGISTER
.A	accumulator
.X	X-index register
.Y	Y-index register
PC	Program counter
SP	Stack pointer
SR	Status register

65C816 registers will be referenced as follows:

SYMBOL	REGISTER	SIZE IN BITS
.A	primary accumulator	8
.B	secondary accumulator	8
.C	accumulator	16[B]
.X	X-index register	8 or 16[C]
.Y	Y-index register	8 or 16[C]
DB	Data bank	8
DP	Direct page pointer	16
PB	Program bank	8
PC	Program counter	16
SP	Stack pointer	16
SR	Status register	8
m	Accumulator/memory size flag	1
x	Index register size flag	1

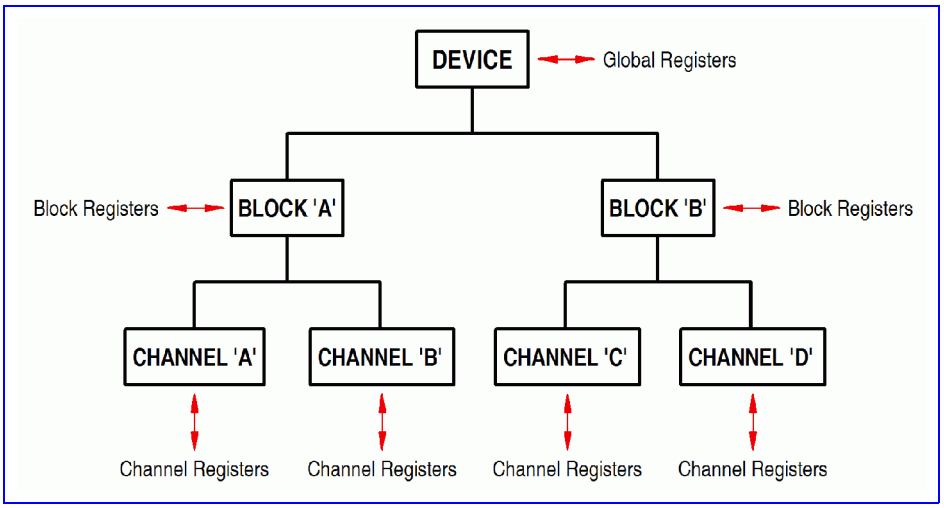
m and x represent the two bits in SR that determine register sizes.

[B] When m=0.

[C] 16 bits when x=0.

2 UART ARCHITECTURE & OPERATION

All NXP IMPACT UARTs share a common architectural model, which we refer to as the DBC (Device-Block-Channel) model.



DBC MODEL FOR SC28L194 FOUR-CHANNEL UART

The above illustration represents the DBC model of one of NXP's quadruple channel UARTs. Other UARTs follow this general layout. All multi-channel UARTs have two channels per block, and the largest UART in this lineup has eight channels. Hence UARTs have one, two or four blocks. In the case of the 28L91, there is only one channel in its one block, but the basic concept of the DBC model still applies.

Registers that affect the entire UART are referred to as "global registers," registers that affect a specific block are referred to as "block registers" and registers that affect a specific channel are referred to as "channel registers." Block registers are repeated for as many blocks are in the device and channel registers are repeated for as many channels are in the device. In the case of the 28L91 and some of the dual channel IMPACT UARTs there are no global registers.

2.1 Basic UART Operation

A serialized datum received by the 28L91 from a remote device is deserialized by the RSR and then transferred to the RHR if space exists therein. Otherwise, the datum remains in the RSR until at least one datum has been read by the MPU from RxD. In a configuration that has enabled hardware flow control, if a datum is in the RSR and the RHR is full, the 28L91 will de-assert its RTS output to tell the remote device to cease transmission. Should another datum arrive from the remote device while a datum is still in the RSR, the newly arrived datum will replace the one in the RSR, causing an overrun error and data stream corruption.

Depending on configuration, the 28L91 may interrupt each time a datum is transferred from the RSR to the RHR, or only when the RHR fills beyond a certain level. The latter configuration takes better advantage of the RHR and improves overall performance during periods of high volume data flow.

Datums written to TxD are queued into the THR and subsequently serialized by the TSR and sent to the remote device. If hardware flow control has been configured and the remote device cannot accept data it will de-assert its RTS output, which usually is wired to the 28L91's CTS input. If RTS is de-asserted transmission to be suspended until the remote device asserts RTS. Serialization and transmission will continue until the THR is empty.

Depending on configuration, the 28L91 may interrupt only when the THR is empty, or any time space is available in the THR for another datum. The former configuration takes better advantage of the THR and improves overall performance during periods of high volume data flow.

2.2 28L91 DBC Model

As previously noted, the 28L91's DBC model is simple, as there is only one block and only one channel—and no global registers. Accordingly, description of the DBC model is simple as well. The following is a synopsis of the channel and block registers. Further discussion on the entire set of registers will follow.

2.2.1 Channel

In hardware terms, a channel is a full-duplex communication path between the host system containing the UART and the remote device, such as a modem, printer or machine controller. In software terms, channel refers to the set of UART registers that directly affects the communication path in various ways. Here is a list of the channel registers:

REGISTER	OFFSET	MODE	SYMBOL	SECTION
Mode setup	00	R/W	nx_mr	3.1.5
Status	01	R	nx_sr	3.1.4
Clock select	01	W	nx_csr	3.1.6
Command	02	W	nx_cr	3.1.7
Serial in	03	R	nx_rhr	3.1.1
Serial out	03	W	nx_thr	3.1.2
Serial I/O	03	R/W	nx_sio	3.1.3

Columns in the above table are defined as follows:

- Register A terse description of the register's function.
- Offset The hexadecimal register offset relative to the base address of the 28L91.

- Mode — Read or write mode, symbolized by R and W, respectively—R/W means "read-write." In some cases, reading from a given register offset will access a different internal UART register than when writing to the same offset. Also, a register offset may be read-only or write-only.

  CAUTION: Reading from a write-only register offset or writing to a read-only register offset may cause undefined device behavior.

- Symbol — The assembly language symbol that will be used to refer to the register in code examples.

- Section — The subsection in this document in which the register is described.

2.2.2 Block

The 28L91's block contains a 16 bit programmable counter/timer (C/T), registers for managing the general purpose inputs and outputs (GPIs and GPOs), and registers for configuring and analyzing UART interrupts. Here is a list of these registers:

REGISTER	OFFSET	MODE	SYMBOL	SECTION
GPI change/status	04	R	nx_ipcr	3.2.1
Auxiliary control	04	W	nx_acr	3.2.2
Interrupt status	05	R	nx_isr	3.2.3
Interrupt mask	05	W	nx_imr	3.2.4
C/T MSB	06	R/W	nx_ctu	3.2.5
C/T LSB	07	R/W	nx_ctl	3.2.5
Uncommited	0C	R/W	nx_ureg	3.2.6
GPI status	0D	R	nx_ipsr	3.2.7
GPO configuration	0D	W	nx_opcr	3.2.8
Start C/T	0E	R	nx_sct	3.2.9
Set GPO bits	0E	W	nx_sopb	3.2.10
Stop C/T	0F	R	nx_rct	3.2.11
Clear GPO bits	0F	W	nx_ropb	3.2.12

Register offsets `$08`, `$09`, `$0A` and `$0B` are not used in the 28L91 and should not be accessed, as undefined device behavior may occur.

3 REGISTER DESCRIPTIONS

In this section, discussion will focus on the UART's registers. The discussion is not exhaustive—consult the data sheet for more detail.

3.1 Channel Registers

The following registers affect only the communication channel.

3.1.1 Serial in - `nx_rhr`, offset `$03`

Reading from this read-only register accesses the RHR and if any datums are present therein, removes the oldest one. Channel status (§3.1.4) should always be checked before reading from this register to determine if a datum is available and if so, if framing, overrun and/or parity errors were detected. Reading from this register when the RHR is empty will return undefined content.

3.1.2 Serial out - `nx_thr`, offset `$03`

Writing to this write-only register accesses the THR and if room exists, stores a datum for transmission. Channel status (§3.1.4) should always be checked before writing to this register to determine if room is available in the THR—a write to a full THR will have no effect.

3.1.3 Serial I/O - `nx_sio`, offset `$03`

See §3.1.1 and §3.1.2, above. The `nx_sio` definition is primarily of value in setting up pointer tables that are subsequently used for indexing into the UART address space.

3.1.4 Status - `nx_sr`, offset `$01`

This read-only register reports the channel's status, mostly in real time, and is routinely read while servicing a UART interrupt. As this register is very important in processing channel activity, each bit's function will be described.

Bit	Meaning if Set	Explanation
0	RHR not empty	At least one datum is in the RHR. This bit is cleared when a read from RxD returns the last datum in the RHR, or if the receiver has been disabled. This bit should always be tested before reading RxD to assure the returned datum is valid.
1	RHR full	The RHR has filled to the level that has been configured, either eight or 16 datums. This bit will be cleared when the MPU reads the last datum from the RHR, or if the receiver has been disabled.

Bit	Meaning if Set	Explanation
2	THR not full	There is room in the THR for another datum. This bit is cleared when the THR has been filled to the level that has been configured, or if the transmitter has been disabled. The UART sets this bit as soon as a datum is transferred from the THR to the TSR, or when the transmitter is enabled.
3	THR empty	The transmitter has run out of datums. This bit is cleared as soon as a datum is written to TxD and subsequently remains cleared until the last datum in the THR has been transferred to the TSR and the TSR has transmitted the final bit of that final datum. Disabling the transmitter will clear this bit. Upon initially enabling the transmitter this bit will be set.
4	RxD overrun error	The RHR is full and another datum has arrived from the remote device, causing the data stream to be corrupted due to the loss of a datum. This bit latches upon detection of an overrun and can only be cleared by writing a command to the channel's command register (§3.1.7).
5	RxD parity error	Parity that was computed during the deserialization of a datum does not match the datum's "payload." This error would only be possible if a data format that uses parity has been selected. Parity is a simple form of error detection that is seldom used, so it will not be discussed any further. This bit latches upon detection of a parity error and can only be cleared by writing a command to the channel's command register (§3.1.7).
6	RxD framing error	Also referred to as a missing stop bit error, a framing error occurs when a stop bit is not detected after a datum has been received and deserialized. For example, if 8N1 format is being used, a framing error would occur if only nine bits were received during the datum time period (approximately 104 microseconds at 9600 bps), instead of the expected 10 bits. Typically, a framing error is due to data link issues or a faulty UART in the remote device. This bit latches upon detection of a framing error and can only be cleared by writing a command to the channel's command register (§3.1.7).

Bit	Meaning if Set	Explanation
7	Received break	A break occurs when the serial data link is in a continuous spacing condition for the duration of one datum period. The UART treats a break specially, placing `$00` into the RHR in lieu of an actual datum and latching this bit until it is cleared via a command to the channel's command register (§3.1.7). The most common source of a break is a user striking the [Break] key on his terminal keyboard to get the host system's attention.

3.1.5 Mode setup - `nx_mr`, offset `$00`

Also referred to as MR (mode register), there are actually three registers at this offset, designated MR0, MR1 and MR2. At power-on or device reset, MR1 is exposed—for example, a read of `nx_mr` would return the content of MR1. Following any access to MR1, MR2 will be exposed and remain exposed unless a command is issued to change to MR1 or MR0. Similarly, if MR0 has been exposed and is accessed, MR1 will be exposed. In other words, MR increments each time it is accessed until MR2 has been exposed. Selecting a particular mode register will be discussed at §3.1.7.3 and §3.1.7.4.

MR configures the channel's data format, RHR and THR sizes, choice of bit rate tables, and several other operating parameters. A discussion of the many mode options would run to many pages, and merely serve to repeat what is already in the NXP data sheet. These options will only be discussed in the context of developing a driver.

NOTE: NXP recommends that RxD and TxD be disabled while writing to the mode registers. Changes to MR have an immediate effect on the channel and in some cases, may cause errors that will disrupt communication.

3.1.6 Clock select - `nx_csr`, offset `$01`

This write-only register is used to set up the data rate for the channel. Despite the register's description, it doesn't directly affect any clocks. Bits 0-3 set the TxD rate and bits 4-7 set the RxD rate. The bit rate table from which the selection is made depends on a configuration setting in MR0 (§3.1.5)—a wide range of standard speeds is available, as well as the ability to configure non-standard speeds.

3.1.7 Command - `nx_cr`, offset `$02`

This write-only register is used to issue a variety of commands to the channel. Most of these commands are issued during device setup. However, several commands are used during routine driver processing activities and will be explained in more detail.

NOTE: According to the NXP data sheet, proper device operation requires that consecutive commands written to bits 4-7 of this register be separated by a minimum of three X1 clock edges. Testing, however, has revealed that the required separation is three X1 clock *cycles*, which is approximately 813 nanoseconds if the X1 frequency is 3.6864 MHZ. It is possible to violate this requirement in a system with a high Ø2 rate, which will lead to difficult-to-diagnose hardware errors.

UART timing can also be inadvertently violated by improper device selection and read/write qualification in the hardware, especially in 65C816 systems.

3.1.7.1 `%00000100`: This command enables the transmitter. If TxD is already enabled when this command is issued the effect is "no operation" (NOP).

3.1.7.2 `%00001000`: This command disables the transmitter. If there are datums in the THR when this command is issued they will processed by the TSR before transmission is actually disabled. Disabling TxD automatically cancels any TxD IRQ that is pending. If TxD is already disabled when this command is issued the effect is a NOP. Writing to TxD is prohibited when the transmitter has been disabled—a datum will not be accepted.

3.1.7.3 `%00010000`: This command sets MR (§3.1.5) to MR1.

3.1.7.4 `%10110000`: This command sets MR to MR0.

There are numerous other commands that may be used. Some of them will be covered later on.

3.2 Block Registers

Most of the following registers only affect the block. Exceptions are noted.

3.2.1 GPI change/status - `nx_ipcr`, offset `$04`

This read-only register serves two purposes:

3.2.1.1 Bits 0-3 reflect the current logic state of the IP0, IP1, IP2 and IP3 general purpose inputs (GPI), respectively, with a set bit indicating that the corresponding GPI is in a logic 1 state. As only these GPIs have change-of-state detection (discussed in *INTERFACING THE NXP SC28L91 UART*), the presence of these bits reduces the processing required within an ISR to respond to and identify an input change-of-state IRQ. Practically speaking, change-of-state detection is of limited value unless the corresponding interrupt has been enabled, as the alternative, polling this register at regular intervals, is an expensive process to implement.

3.2.1.2 Bits 4-7 indicate which of the GPIs has registered a change of state, with a set bit indicating that a particular GPI was responsible. If the GPI change-of-state interrupt has been enabled, reading this register will identify which GPI(s) triggered the IRQ.

3.2.2 Auxiliary control - `nx_acr`, offset `$04`

This write-only register is used to configure several block features:

3.2.2.1 Bits 0-3 enable (1) or disable (0) GPI change-of-state detection, with bit 0 corresponding to IP0, bit 1 to IP1, etc. For example, writing `%0110` into these bits will enable change-of-state detection for IP1 and IP2 only. As this register is write-only, it may be necessary to shadow it somewhere in RAM.

3.2.2.2 Bits 4-6 determine the operating mode of the C/T, as well as the clock source that is its time base. In the sections in which code examples are presented, we will describe how to configure the C/T to generate a "jiffy" interrupt that can drive software timekeeping features.

3.2.2.3 Bit 7 affects the channel, as it selects one of two sets of clocking rates that determine the speed at which the bit rate generator (BRG) runs (the BRG is what drives the RSR and TSR). The state of this bit, along with MR0 configuration (§3.1.5), determines the range of data rates that may be selected.

3.2.3 Interrupt status - `nx_isr`, offset `$05`

This read-only register reports the state of the five possible interrupt sources within the 28L91, with set bits indicating active interrupts. Interrupt sources must be configured to generate hardware interrupts by writing an appropriate bit pattern to the interrupt mask register (§3.2.4).

As knowing the interrupt status of the 28L91 is essential to driver operation some detail on this register will be presented.

Bit	Interrupt Source	Explanation
0	TxD	When set, the datum count in the THR has decreased below a configured threshold, which is set via MR (§3.1.5). Depending upon channel configuration, TxD may not interrupt until the THR is empty, which means as many as 16 datums could be transmitted before TxD interrupts. This interrupt is cleared by writing to TxD, or by disabling the transmitter (§3.1.7.2). Enabling the transmitter (§3.1.7.1) will set this bit and if the corresponding bit in the interrupt mask register (§3.2.4) is set, the MPU will be interrupted.
1	RxD	When set. one or more datums are available in the RHR. Depending upon channel configuration, RxD may not interrupt until the RHR is full, which means as many as 16 datums could be received before RxD interrupts. This interrupt is cleared by reading RxD until no more data is available, or by disabling the receiver.

Bit	Interrupt Source	Explanation
2	Change-of-Break	When the 28L91 detects the start or end of a break it sets this bit. Detection of a break is latched (§3.1.4) and must be cleared by writing a command into the command register (§3.1.7). This interrupt is cleared by writing a "clear break interrupt" command to the channel's command register.
3	C/T Ready	The C/T has reached terminal count. The exact meaning of "terminal count" depends on whether the C/T is operating in counter mode or timer mode. In timer mode, the IRQ occurs each time the count reaches `$0000`. This interrupt is cleared by issuing a "stop counter" command (§3.2.11).
4-6	——	These bits are not used and should be masked.
7	GPI change-of-state	A GPI's logic state has changed. This interrupt only indicates that a change-of-state has occurred, not which GPI caused it or the direction of change, e.g., high to low. A read of the GPI change/status register (§3.2.1) is necessary to ascertain the cause of this interrupt, as well as to clear it.

3.2.4 Interrupt mask - `nx_imr`, offset `$05`

This write-only register is used to enable or disable interrupt sources in the 28L91. It exactly corresponds with the interrupt status register (§3.2.3) in bit format. The byte written into this register is a mask: if a bit is set in the mask the corresponding interrupt source will be able to interrupt the MPU. If a bit is cleared in the mask the corresponding interrupt source will not be able to interrupt the MPU.

As bits 4, 5 and 6 in the interrupt status register are unused, they should not be set in any value written into this register in order to avoid undefined behavior. If your driver depends on being able to turn on and off interrupts during runtime you will need shadow this register in RAM.

3.2.5 C/T MSB - `nx_ctu`, offset `$06`
C/T LSB - `nx_ctl`, offset `$07`

When read, these registers return the current value in the counter/timer. 65C816 programmers should note that these registers are returned in big-endian format when read by a 16-bit register. Once the C/T has been started, these registers may be in a constant state of change. A "carry error" may occur when reading the C/T "on the fly," as one register may change while the other is being read.

The values written to these registers set the C/T's starting count. Once started in timer mode, the C/T will count down from the starting count to `$0000`, reload the starting count and again count down, endlessly repeating the sequence.

If a C/T interrupt has been enabled the MPU will be interrupted each time the C/T reaches `$0000`. This behavior makes the C/T suitable for generating a regularly spaced interrupt for system timekeeping and/or initiating periodic task switching.

The most often-used time base to drive the C/T is the X1 clock. However, it is possible to drive the C/T with an external clock or have the C/T count an external pulse train.

3.2.6 Uncommited - `nx_ureg`, offset `$0C`

When operated in the recommended x86 (Intel) bus compatibility mode, the 28L91 does not use this register for anything. The 28L91 data sheet indicates that any value written into this register will be retained and may be read back, allowing this register to act as a one byte temporary store. At power-on or reset the value `$0F` (`%00001111`) will be present in this register.

3.2.7 GPI status - `nx_ipsr`, offset `$0D`

Bits 0-6 of this read-only register report the logic states of the six GPIs in real time, with bit 0 reporting IP0, bit 1 reporting IP1, etc., and bit 6 reporting IP6. Bits 0-3 of this register repeat bits 0-3 of the GPI change/status register (§3.2.1). Bit 7 is unused and should be masked. A set bit corresponds to a logic 1 state at the corresponding GPI.

3.2.8 GPO configuration - `nx_opcr`, offset `$0D`

This write-only register is used to configure the behavior of the general purpose outputs (GPO). Setting up the GPOs will be presented when the driver is discussed.

3.2.9 Start C/T - `nx_sct`, offset `$0E`

Reading this read-only register will start the C/T if it is not running. In timer mode operation, a start C/T operation will load the 16-bit value written to the C/T registers (§3.2.5) into the counter and start the count-down if not already running. If the C/T is already running, start C/T will terminate the current timing cycle and start a fresh one. Note that once started in timer mode it is not possible to stop the C/T.

3.2.10 Set GPO bits - `nx_sopb`, offset `$0E`

This write-only register is used to set the state of the GPOs that have been configured to act as general purpose outputs. The byte written into this register is a mask, with bit 0 corresponding to OP0, bit 1 corresponding to OP1, etc., and bit 7 corresponding to OP7. Set bits in the mask set the corresponding GPOs to the logic 1 state. Cleared bits have no effect on the corresponding GPOs.

3.2.11 Stop C/T - `nx_rct`, offset `$0F`

Reading this read-only register will clear any pending C/T interrupt and will stop the C/T if it is running in counter mode. In timer mode operation, a stop C/T operation will not actually stop the C/T—only a pending interrupt will be cleared.

3.2.12 Clear GPO bits - `nx_ropb`, offset `$0F`

This write-only register is used to clear the state of the GPOs that have been configured to act as general purpose outputs. The value written into this register is a mask, with bit 0 corresponding to OP0, bit 1 corresponding to OP1, etc., and bit 7 corresponding to OP7. Set bits in the mask clear the corresponding GPOs to the logic 0 state. Cleared bits have no effect on the corresponding GPOs.

4 DEVICE DRIVER DESIGN

A device driver's role in a computer system is to act as an interface between the local operating environment[D] and a particular part of the hardware, the 28L91 in this case. The device driver that will be presented fulfills that role in three code sections:

Driver Section	What It Does
Initialization	Initializes the UART and driver.
Foreground	Processes serial input/output requests.
Background	Processes UART interrupts.

These code sections interact in various ways with the 28L91 and with each other so as to present an application programming interface (API) to the local operating environment. The following discussion will focus on the general design of the driver, as well as present code examples and other material to aid understanding.

4.1 Application Programming Interface (API)

The driver's API doesn't account for the possibility of a program running on a 65C816 system with extended RAM calling a driver function from a bank other than the one in which the driver is loaded. Accommodating cross-bank access is an exercise left for the 65C816 programmer implementing the driver.

Here are the API functions that will be implemented.

4.1.1 `sioinit`

`sioinit` initializes the driver's data structures, loads the 28L91's registers with required setup parameters and otherwise prepares the UART for service. `sioinit` must be called once during system initialization. Upon completion, sioinit will set a flag in `nx_ureg` (§3.2.6) to indicate that initialization has been performed.

The syntax for calling `sioinit` is:

```
        jsr sioinit             ;initialize driver & UART
```

[D] In this context, "local operating environment" could also mean an application program running on a machine without a formal operating system.

4.1.2 sioget

`sioget` returns a received datum in .A if one is available, otherwise .A is unchanged upon return. Carry will be set if no datum is available or if the driver initialization function `sioinit` has not been called.

The syntax for calling `sioget` is:

```
        jsr sioget          ;retrieve a datum
        bcs nodatum         ;no datum or...
                            ;driver not initialized
```

4.1.3 sioput

`sioput` submits the datum in .A for transmission. `sioput` will block until the datum has been accepted and queued for transmission, unless the initialization function has not been called, in which case an immediate return with carry set will occur.

The syntax for calling `sioput` is:

```
        lda #datum          ;datum in .A
        jsr sioput          ;transmit datum
        bcs noinit          ;driver not initialized
```

4.2 Input/Output Mechanics

The UART driver uses circular queues (RxQ and TxQ, page 4) to pass data between the foreground, background and UART itself. Queuing helps to decouple the pace at which the UART operates from the pace at which the rest of the system operates, allowing, for example, for datums to be received and promptly processed in the background as they arrive, yet be retrieved and processed by the foreground when convenient.

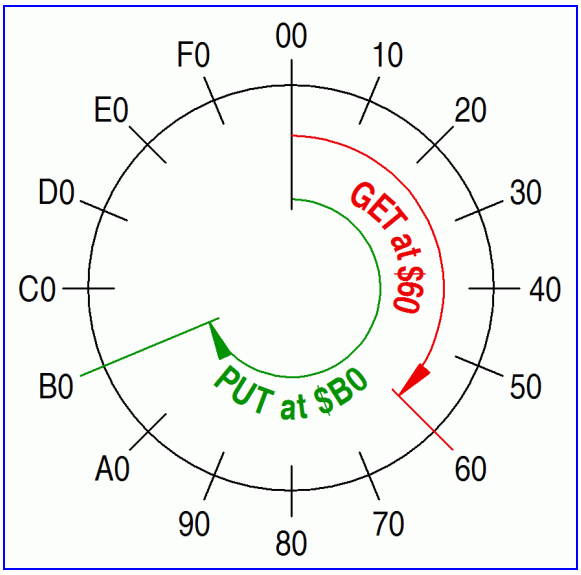
As queued input/output (I/O) is a key part of the driver's operation some elaboration will be presented.

4.2.1 Circular queue design & operation

In the driver, RxQ and TxQ are implemented as 256 contiguous byte circular queues. 256 bytes is a convenient size when working with the 65Cxx microprocessors and in practice, has been found to be more than adequate, even with high performance UARTs such as the 28L91.[E]

[E] A smaller queue size is possible with some minor changes to the code that manages the queue. BCS Technology Limited has tested with as small as a 64 byte queue, while maintaining satisfactory performance. The most noticeable effect of a small queue size is more frequent blocking when transmitting a lengthy data stream, such as a full screen repaint of a console display. Unequal queue sizes are also feasible.

Illustrated below is a graphical representation of a 256 byte circular queue.



256 Byte Circular Queue

A circular queue of this size may be managed by two one byte indices called "get" and "put," their current values being interpreted as offsets into the queue. Both indices will be initialized to the same value during driver initialization, typically but not necessarily `$00`.

The "put" index determines where the next datum will be stored into the queue. As can be seen in the above illustration, the next "put" will occur at `$B0` relative to the start or "head" of the queue. When a datum has been "put" into the queue the "put" index will be incremented and will wrap to `$00` if incremented past `$FF`. Hence successive "puts" will appear to be circular in nature.

The "get" index determines where the next datum will be fetched from the queue. As can be seen in the above illustration, the next "get" will occur at `$60` relative to the head of the queue. When a datum has been gotten from the queue the "get" index will be incremented and will wrap to `$00` if incremented past `$FF`. Hence successive "gets" will also appear to be circular.

Prior to performing a "put," it must be known if room exists in the queue to accept a datum. Likewise, it must be known if a datum is in the queue before attempting a "get." There are two conditions that are of interest in making these determinations:

CONDITION	MEANING
`put = get`	Queue is empty
`put+1 = get`	Queue is full

As can be seen, a "get" is only possible when the expression put = get evaluates false, and a "put" is only possible when the expression put+1 = get evaluates false (the evaluation will discard carry). During queue operation, "put" should always be equal to or "ahead" of "get." Any other condition will result in the corruption of the queue's contents following a "put," or undefined content being returned following a "get."

Although not germane to the discussion at this point, subtracting the "get" index from the "put" index and discarding the borrow reports how many datums are in the queue.

4.2.2 "Get" and "put" indices & queues

As each queue has a "get" and "put" index and as there is a queue for received datums (RxQ) and datums awaiting transmission (TxQ), two sets of indices must be provided to manage the queues. In the UART driver, the indices are defined as follows:

```
nx_psize =1                     ;index & flag size
nx_base  = $00                  ;start of driver ZP workspace
nx_rxget = nx_base              ;RxQ "get" index
nx_txget = nx_rxget+nx_psize    ;TxQ "get" index
nx_rxput = nx_txget+nx_psize    ;RxQ "put" index
nx_txput = nx_rxput+nx_psize    ;TxQ "put" index
```

It is customary, though not essential, to place these indices on page zero (direct page) in order to improve the performance of the routines that refer to them. The indices may be relocated by changing the highlighted **nx_base** assignment. More will be added to the above later on.

As noted before, both RxQ and TxQ are 256 byte structures, which may be located anywhere in RAM that is convenient. The following code defines the queues.

```
nx_qsize =256                   ;queues' size in bytes
nx_qbase =$C000                 ;base address for queues
nx_rxq   =nx_qbase              ;RxQ
nx_txq   =nx_rxq+nx_qsize       ;TxQ
```

The queues may be relocated by changing the highlighted **nx_qbase** assignment. Later on, some other assignments will be added to the above.

The above code fragments illustrate an important aspect of good assembly language programming practice. "Magic numbers," such as those represented by nx_psize and nx_qbase, should not be "hard coded" in the instructions that reference them, but instead formally assigned to symbols and the symbols used in subsequent statements. If constants are symbolically represented as shown instead of hard-coded, bugs that may be caused by incorrectly typing numbers into code statements will be avoided. Also, revisions will be more easily accomplished, since only one or two assignments would need to be edited prior to reassembly.

The alternative would be searching through source files to locate and edit all assignments affected by any changes.

As the presentation of the driver progresses more symbols will be defined. It is customary in larger projects to place such definitions into one or more INCLUDE files for convenience and so they may be reused as needed in other projects.

4.2.3 Queue management algorithms

Having described the basics of working with the queues, some code examples will now be presented in which several queue management algorithms will be illustrated. First, the logic will be presented for determining if RxQ has any datums.

```
        ldx nx_rxget            ;RxQ "get" index
        cpx nx_rxput            ;RxQ "put" index
        beq rxqempty            ;RxQ is empty
```

Recall from the table in §4.2.1 (page 19) that if "put" equals "get" then the queue is empty. The above code tests for that case. Virtually identical code would be used with TxQ.

```
        ldx nx_txget            ;TxQ "get" index
        cpx nx_txput            ;TxQ "put" index
        beq txqempty            ;TxQ queue empty
```

Next, the logic will presented for determining if RxQ can accept another datum.

```
        ldx nx_rxput            ;RxQ "put" index
        inx                     ;"put" +1
        cpx nx_rxget            ;RxQ "get" index
        beq rxqfull             ;RxQ is full
```

Recall from the table in §4.2.1 (page 19) that if "put+1" equals "get" then the queue is full. The above code tests for that case. The same logic would apply to TxQ.

```
        ldx nx_txput            ;TxQ "put" index
        inx                     ;"put" +1
        cpx nx_txget            ;TxQ "get" index
        beq txqfull             ;TxQ is full
```

65C816 programmers should note that all of the above operations are to be performed with the index registers set to eight bits.

Having devised ways in which to assess the state of RxQ and TxQ, it is now possible to write code that will perform a "get" or "put" on either queue.

Below is example code that gets a datum from RxQ. If carry is clear following the operation then .A will be loaded with a datum. Otherwise, .A will be unchanged and carry will be set to indicate that there were no datums in the queue. 65C816 programmers should note that the accumulator must be set to eight bits.

```
          ldx nx_rxget          ;RxQ "get" index
          cpx nx_rxput          ;RxQ "put" index
          beq rxqempty          ;RxQ is empty
;
          lda nx_rxq,x          ;get datum from RxQ &...
          inc nx_rxget          ;increment "get" index
          clc                   ;datum gotten
;
rxqempty ...program continues...
```

The above code fragment would be part of the driver's foreground section and is called as part of the API that returns a received datum. The code that puts datums into RxQ would be part of the background.

Note that the datum is gotten before `nx_rxget` is incremented. It is essential that the "get" index not be changed until the "get" has completed. Otherwise, the background might get confused as to whether or not the queue has room for the next "put," since the interrupt that will process the background "put" could occur at any time while the above code is being executed.

A "get" on TxQ is performed in the background as part of servicing a UART interrupt. The code for performing a "get" on TxQ is identical to that for RxQ except for the label references.

```
          ldx nx_txget          ;TxQ "get" index
          cpx nx_txput          ;TxQ "put" index
          beq txqempty          ;TxQ is empty
;
          lda nx_txq,x          ;get datum from TxQ &...
          inc nx_txget          ;increment "get" index
          clc                   ;datum gotten
;
txqempty ...program continues...
```

Again, carry will be clear if a datum was gotten or set if no datum was available. .A will be unchanged in the latter case. The implications of TxQ being empty when a "get" is attempted will be discussed as part of the interrupt processing subsection (§4.3.7).

Example code that performs a "put" will now be presented. A "put" on RxQ is performed in the background as part of servicing an RxD interrupt. The implications of RxQ being full when the "put" operation is attempted will be discussed as part of the interrupt processing subsection (§4.3.6).

```
         ldx nx_rxput          ;RxQ "put" index
         inx                   ;"put +1"
         cpx nx_rxget          ;RxQ "get" index
         beq rxqfull           ;RxQ is full
;
         dex                   ;realign "put" index
         sta nx_rxq,x          ;put datum into RxQ
         inc nx_rxput          ;increment "put" index
         clc                   ;operation completed
;
rxqfull  ...program continues...
```

The code required to perform a "put" to TxQ is the same as for RxQ except for the label references. The below code fragment would be part of the driver's foreground section and is called as part of the API that transmits a datum. As previously noted, the code that gets a datum from TxQ would be part of the driver's background.

```
         ldx nx_txput          ;TxQ "put" index
         inx                   ;"put" +1
         cpx nx_txget          ;TxQ "get" index
         beq txqfull           ;TxQ is full
;
         dex                   ;realign "put" index
         sta nx_txq,x          ;put datum into TxQ
         inc nx_txput          ;increment "put" index
         clc                   ;operation completed
;
txqfull  ...program continues...
```

The subsection on foreground processing, of which the above code is part, will discuss the implications of TxQ being full when a "put" is attempted.

4.3 Interrupt processing

In prior discussion on queue management, mention has been made of processing being handled by background code. This subsection will expand on the role played by background processing.

In the driver, input/output interaction with the UART occurs exclusively in an interrupt service routine (ISR), which is the background part of the driver. For example, when the UART is ready to accept a datum for transmission it will interrupt the MPU and if a datum is waiting in TxQ, code in the ISR will get that datum and write it to TxD. Similarly, when a datum is received and de-serialized, the UART will interrupt the MPU, which will read the datum from RxD and put it into RxQ.

For more than a few 6502 assembly programmers, writing interrupt-driven code of any kind is something to be approached with trepidation.

However, developing a bug-free ISR is not a difficult task once one understands the device behavior when it interrupts, has an understanding of the required logic flow, and has developed disciplined programming habits.

In the driver implementation, three of the five possible interrupt sources in the 28L91 will be enabled and serviced, namely TxRDY, RxRDY and Counter Ready (bits 0, 1 and 3, respectively, in the interrupt status and mask registers—see §3.2.3 and 3.2.4). When the UART interrupts, the ISR will be required to determine why and then execute the appropriate service code.

4.3.1 Counter Ready

To recapitulate, the C/T is a precision counter with sub-microsecond resolution that is normally driven from the X1 clock (the X1 clock is described in the article *INTERFACING THE NXP SC28L91 UART*). The C/T may be operated in counter mode or timer mode, with the latter being the mode used in the driver. Upon being started in timer mode, the C/T will count down from the 16-bit prescalar value loaded into its registers until it reaches `$0000`. At that time, a Counter Ready interrupt will occur, the C/T will reload its registers with the prescalar value and again count down to `$0000`, endlessly repeating the sequence. Issuing a stop C/T command (§3.2.11) to the UART will clear a Counter Ready interrupt but will not stop the C/T. The result is the C/T will generate an evenly-spaced interrupt that is traditionally referred to as a "jiffy IRQ."

4.3.2 RxRDY

An RxRDY interrupt will occur when RxD is enabled and data is present in the RHR, subject to certain configurable rules. The interrupt is cleared by reading all datums from RxD, or by disabling the receiver.

A UART with a receiver FIFO is often configured to not interrupt until the FIFO is nearly or completely filled with received datums, a programming tactic that reduces the frequency at which the MPU must service receiver IRQs. However, such a configuration creates a situation in which if some datums arrive but not enough to fill the FIFO to the level required to trigger an IRQ, the datums will go "stale" from not being processed in a timely fashion.

This possibility is addressed in the 28L91 with the RxD watchdog timer, which will cause an RxRDY interrupt if datums remain in the RHR for too long, where "too long" is determined by the bit rate at which RxD is operating. Hence if a continuous inflow of datums occurs, advantage is taken of the full capacity of the RHR. If only one datum arrives, such as would happen with a terminal on which only one keystroke was made, the UART will eventually interrupt due to the action of the RxD watchdog timer and the one datum will be processed.

4.3.3 TxRDY

A TxRDY interrupt will occur when TxD is enabled and the THR is empty. The interrupt is cleared by writing at least one datum into TxD, or by disabling the transmitter.

In the event there are no datums in TxQ then the TxD interrupt must be suppressed so the system doesn't go into deadlock. This may be accomplished by clearing the TxD bit in the interrupt mask register (§3.2.4) or by disabling the transmitter (§3.1.7.2), the latter being the recommended method, as it involves a quick write to the channel's command register (§3.1.7). Regardless of which method is used, a flag must be set somewhere to indicate when the TxRDY IRQ has been disabled so it can be restarted when more data becomes available for transmission.

Any ISR design must account for interrupt priority. In the driver, C/T and RxRDY interrupts are given greater priority than other interrupts the UART may generate.[F] The reasoning is that these two interrupts tend to be time-critical, especially RxRDY—failure to promptly service RxRDY may result in a receiver overrun error.

A requirement of any ISR is that it be "transparent" to the foreground so the latter can be resumed without error when interrupt processing has completed. Transparency is usually achieved by pushing the MPU's registers to the hardware stack at the beginning of the ISR and then pulling them from the stack at the completion of the ISR. Generally speaking, registers that will not be "touched" (used) within the ISR should not be pushed and pulled, a strategy that will slightly improve the performance of the ISR.[G] Other transparency measures include not using any memory other than that of the hardware stack, except in well defined cases.

In the interest of generality, the following examples will assume that all registers will be touched at some point as the ISR executes and therefore all registers will be preserved. As the 65C02 and 65C816 have differing architectures, an ISR for one cannot be exactly the same as the other, with differences starting in the ISR preambles and postambles in which the state of the MPU is respectively preserved and restored. The following subsections will offer some examples for each processor.

4.3.4 ISR pre- and postambles

The 65C02 ISR preamble (also called the "front end"—see next page) pushes all registers—the exact order is immaterial—and then owing to the lack of separate hardware vectors for interrupt requests (IRQ) and software interrupts (BRK), "sniffs" the stack to determine which interrupt type is being processed. The example code continues the earlier practice of symbolically defining all "magic numbers" rather than burying them in code.

[F] Consideration must also be given to the priority of interrupts that will be generated by other hardware in the system.

[G] If the reader is not conversant with 6502 family interrupt behavior he or she should visit **http://wilsonminesco.com/6502interrupts** and **http://sbc.bcstechnology.net/65c816interrupts.html** for an extensive discussion on the topic, as well as numerous code examples. ISR design discussion in this article will be mostly limited to that required to implement the UART driver.

```
;65C02 Interrupt Service Handler
;
hwstack  =$0100                 ;65C02 stack's base address
reg_sr   =$04                   ;SR's relative offset on stack
m_sr_b   =%00010000             ;SR BRK bit mask
;
         pha                    ;save .A
         phx                    ;save .X
         phy                    ;save .Y
         tsx                    ;current stack pointer
         lda hwstack+reg_sr,x   ;load stack copy of SR
         bit #m_sr_b            ;IRQ or BRK?
         bne is_brk             ;is BRK
;
         ...program falls through to IRQ handler...
```

Upon completion of the ISR, the following postamble example would be executed to return to the foreground task that was interrupted.

```
;65C02 Common Interrupt Return
;
crti     ply                    ;restore MPU state
         plx
         pla
         rti                    ;return to foreground
```

The above code section is given a label—`crti`, which stands for "common return from interrupt," as it can also be used to return from other interrupt handlers.

Unlike the 65C02, the 65C816 has separate native mode hardware vectors for IRQ and BRK, thus eliminating the stack "sniffing" required with the 65C02. However, the 65C816 has a more complex operating environment with more registers to preserve, plus the added complication of variable-sized user registers—the accumulator and index register sizes at the time the 65C816 responds to an interrupt will be not be known.

The 65C816 ISR preamble example on the next page addresses all such concerns. Following complete preservation of the MPU's state, DP is loaded with the local operating environment's notion of where direct (zero) page is located, which may not be the same as the foreground's notion. Similarly, DB is set to that of the local environment, as the foreground may have been working in a different bank in which local operating environment data structures would not be accessible without use of long addressing. For a fuller explanation it is recommended that the reader visit http://sbc.bcstechnology.net/65c816interrupts.html for a complete treatment of 65C816 interrupt processing.

```
;65C816 Interrupt Service Handler Preamble
;
         rep #%00110000        ;select 16-bit registers
         phb                   ;save DB
         phd                   ;save DP
         pha                   ;save .C
         phx                   ;save .X
         phy                   ;save .Y
         lda #kerneldp         ;set local OS...
         tcd                   ;direct page
         sep #%00100000        ;select 8-bit accumulator
         lda #kerneldb         ;set...
         pha                   ;local OS...
         plb                   ;data bank
;
         ...program continues...
```

A postamble to the above would be the following example.

```
;65C816 Common Interrupt Return
;
crti     rep #%00110000        ;select 16-bit registers
         ply                   ;restore MPU state
         plx
         pla
         pld
         plb
         rti
```

Again, `crti` can also be used as an exit point for other interrupt handlers in the system.

With ISR pre- and postambles presented, discussion will turn to the body of the ISR. As previously noted, the Counter Ready IRQ is given highest priority, followed by the RxRDY IRQ and finally the TxRDY IRQ.

4.3.5 Processing Counter Ready interrupts

In the following example code, the Counter Ready interrupt will update a software clock. The C/T itself will be set up during driver initialization to interrupt 100 times per second, hence generating IRQs at 10 millisecond intervals.

The software clock implemented in the driver is a 32-bit uptime counter that starts at zero when the system is booted and is incremented at one second intervals, thus keeping track of how long the system has been running. An eight-bit down-counter, referred to as the “jiffy counter,” is used to keep track of the number of jiffy interrupts that have occurred between updates of the uptime counter.

The jiffy counter is initialized to 100 at boot time and subsequently decremented with each Counter Ready IRQ. When the jiffy counter has reached zero it will be reset to 100 and the uptime counter will be incremented. Other software clocks, such as time-of-day, could be added to this mélange, using the principles that will be presented in the following examples.

Before presenting the actual ISR code, additions to some earlier definitions (§4.2.2 on page 20) is in order.

```
s_byte   = 1                    ;size of a byte
s_word   = 2                    ;size of a word
s_dword  = 4                    ;size of a double word
s_uptime = s_dword              ;size of uptime counter
nx_psize = 1                    ;index & flag size
hz       = 100                  ;frequency of C/T IRQs
nx_zbase = $00                  ;driver's ZP workspace
;
nx_jifct = nx_zbase             ;jiffy counter
nx_uptct = nx_jifct+s_byte      ;32 bit uptime counter
nx_rxget = nx_uptct+s_uptime    ;RxQ "get" index
nx_txget = nx_rxget+nx_psize    ;TxQ "get" index
nx_rxput = nx_txget+nx_psize    ;RxQ "put" index
nx_txput = nx_rxput+nx_psize    ;TxQ "put" index
nx_txstw = nx_txput+nx_psize    ;TxD status, $00 = TxD enabled
nx_zend  = nx_txstw+nx_psize    ;end of driver ZP workspace
```

The jiffy and uptime counters have been added, as well as data sizes and other declarations. In particular, note the symbol `hz`, which defines the number of jiffy IRQs that will occur in one second. Some other definition updates are also required.

```
cx_txdis =%00001000             ;disable transmitter command
cx_txenb =%00000100             ;enable transmitter command
mx_txdis =%10000000             ;TxD flag mask
mx_rxrdy =%00000001             ;RHR not empty mask
mx_txrdy =%00000100             ;THR not full mask
mx_txirq =%00000001             ;TxRDY IRQ mask
mx_rxirq =%00000010             ;RxRDY IRQ mask
mx_ctirq =%00001000             ;Counter Ready IRQ mask
mx_drivr =%10000000             ;"driver ready" flag
nx_qsize =256                   ;RxQ & TxQ size in bytes
;
nx_qbase =$C000                 ;base address for queues
nx_ubase =$D000                 ;base address for UART

nx_rxq   =nx_qbase              ;RxQ base address
nx_txq   =nx_rxq+nx_qsize       ;TxQ base address
```

In the above assignments, the UART's base address has been defined and is assumed to be at $D000. Some new constants that will be used in interrupt and foreground processing have also been added.

The first processing step is to determine if the 28L91 is interrupting.

```
;  determine interrupt source...
;
         sep #%00010000         ;8-bit .X & .Y (65C816 only)
         lda nx_ubase+nx_isr    ;UART interrupting?
         beq not_uart           ;no, must be something else
;
is_uart  ...program continues...
```

If no interrupts are pending in the 28L91 then %00000000 will be read from the interrupt status register (nx_isr, §3.2.3 on page 13) and the code will bypass the UART part of the ISR by jumping directly to not_uart. Otherwise, the cause of the UART's interrupt must be determined and appropriate action carried out. The following 65C02 example continues from the is_uart label in the above code example and processes a Counter Ready IRQ.

```
;  process Counter Ready interrupt (65C02)...
;
is_uart  bit #mx_ctirq          ;C/T interrupting?
         beq not_ct             ;no
;
         lda nx_ubase+nx_rct    ;yes, clear C/T IRQ
         lda nx_jifct           ;get jiffy counter
         dec A                  ;count -1
         bne .0000020           ;not time to update uptime
;
         lda #hz                ;reset jiffy counter
         ldx #0                 ;uptime update index
         ldy #s_uptime          ;size of uptime counter
;
.0000010 inc nx_uptct,x         ;uptime = uptime + 1
         bne .0000020           ;no carry, done with uptime
;
         inx                    ;next...
         dey                    ;uptime counter cell
         bne .0000010           ;continue with update
;
.0000020 sta nx_jifct           ;set new jiffy count value
         lda nx_ubase+nx_isr    ;reload interrupt status
;
not_ct   ...program continues...
```

In the above example, labels such as .0000010 and .0000020 are "local" labels, whose scope is bounded by the nearest "global" labels, is_uart and not_ct—use whatever symbology is supported by your assembler.

Recall that the earlier code segment that determined if the UART was interrupting had loaded the UART's interrupt status into .A. The above code tests the Counter Ready bit of the interrupt status value in .A and if it is set then the C/T had to have reached terminal count ($0000). That being the case, the Stop C/T command is executed to clear the interrupt before continuing.[H] The remainder of the code updates timekeeping, starting with decrementing the jiffy counter. If it reaches zero then the uptime counter is incremented in a loop and the jiffy count is reset to hz to begin anew.[I] The result is that the uptime increases once per second.

The 65C816 version is somewhat different.

```
;  process Counter Ready interrupt (65C816)...
;
is_uart  bit #mx_ctirq          ;C/T interrupting?
         beq not_ct             ;no
;
         lda nx_ubase+nx_rct    ;yes, clear C/T IRQ
         lda nx_jifct           ;get jiffy counter
         dec A                  ;count -1
         bne .0000020           ;not time to update uptime
;
         rep #%00100000         ;16-bit accumulator/memory
         inc uptct              ;increment uptime LSW
         bne .0000010           ;done with uptime
;
         inc uptct+s_word       ;increment uptime MSW
;
.0000010 sep #%00100000         ;8-bit accumulator/memory
         lda #hz                ;reset jiffy count
;
.0000020 sta nx_jifct           ;set new jiffy count value
         lda nx_ubase+nx_isr    ;reload interrupt status
;
not_ct   ...program continues...
```

The above code executes much faster than the 65C02 equivalent due to being able to increment 16 bits in one instruction.

[H] As a general rule, IRQs should be cleared as soon as possible to minimize the risk of a spurious interrupt occurring.

[I] As the jiffy count cycles between 100 and zero it indirectly reports the fractional part of the uptime to the nearest 10 milliseconds. The fractional part may be computed by evaluating **(hz - (jiffy count-1)) × (hz ÷ 10)**, where **hz** is the jiffy IRQ frequency (100 Hz).

In both examples, upon completion of Counter Ready interrupt processing the UART's interrupt status is reloaded in preparation for the next stage of processing.

4.3.6 Processing RxRDY interrupts

Processing an RxRDY IRQ requires that the RHR be completely emptied to clear the interrupt. Hence it isn't practical to read one datum from RxD and then move on, as would be done with a UART without a FIFO. This implies the use of a loop to process datums. As an aid to understanding the required logic, a simple flow chart has been presented on page 32.

RxRDY processing starts with determining that the receiver is interrupting. Next, a loop is entered in which the RHR is tested to see if it contains at least one datum. If a datum is available, it is read and if possible, put into RxQ. The sequence repeats until the 28L91 indicates that the RHR is empty, at which time the RxRDY IRQ will have been cleared.

During receiver processing an interesting problem arises if RxQ is full. In order to clear the RxRDY interrupt, all datums in the RHR must be read, even if RxQ cannot accept them. However, any datums read after RxQ has filled will have no place to go and therefore must be abandoned, which is what the logic in the flow chart does. The only way to avoid the data loss is to arrange the foreground to process RxQ in a more timely fashion.

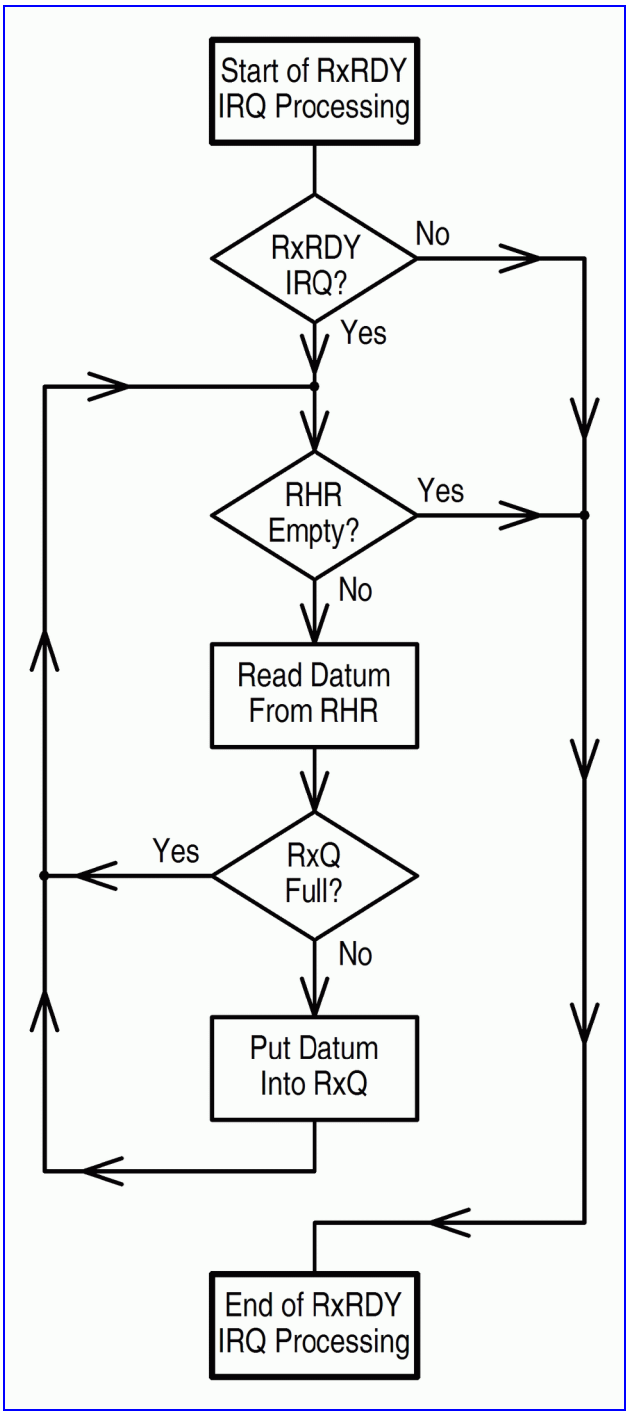
The code required to accomplish what is depicted in the flow chart is not at all complicated, and draws upon code elements that have already been presented. Refer to page 33 for an example that is suitable for both the 65C02 and 65C816.

4.3.7 Processing TxRDY interrupts

Processing a TxRDY IRQ requires that at least one datum be written to TxD to clear the interrupt. If no datum is available, TxD must be disabled to suppress the interrupt and prevent deadlock. As an aid to understanding the required logic, a simple flow chart has presented on page 34.

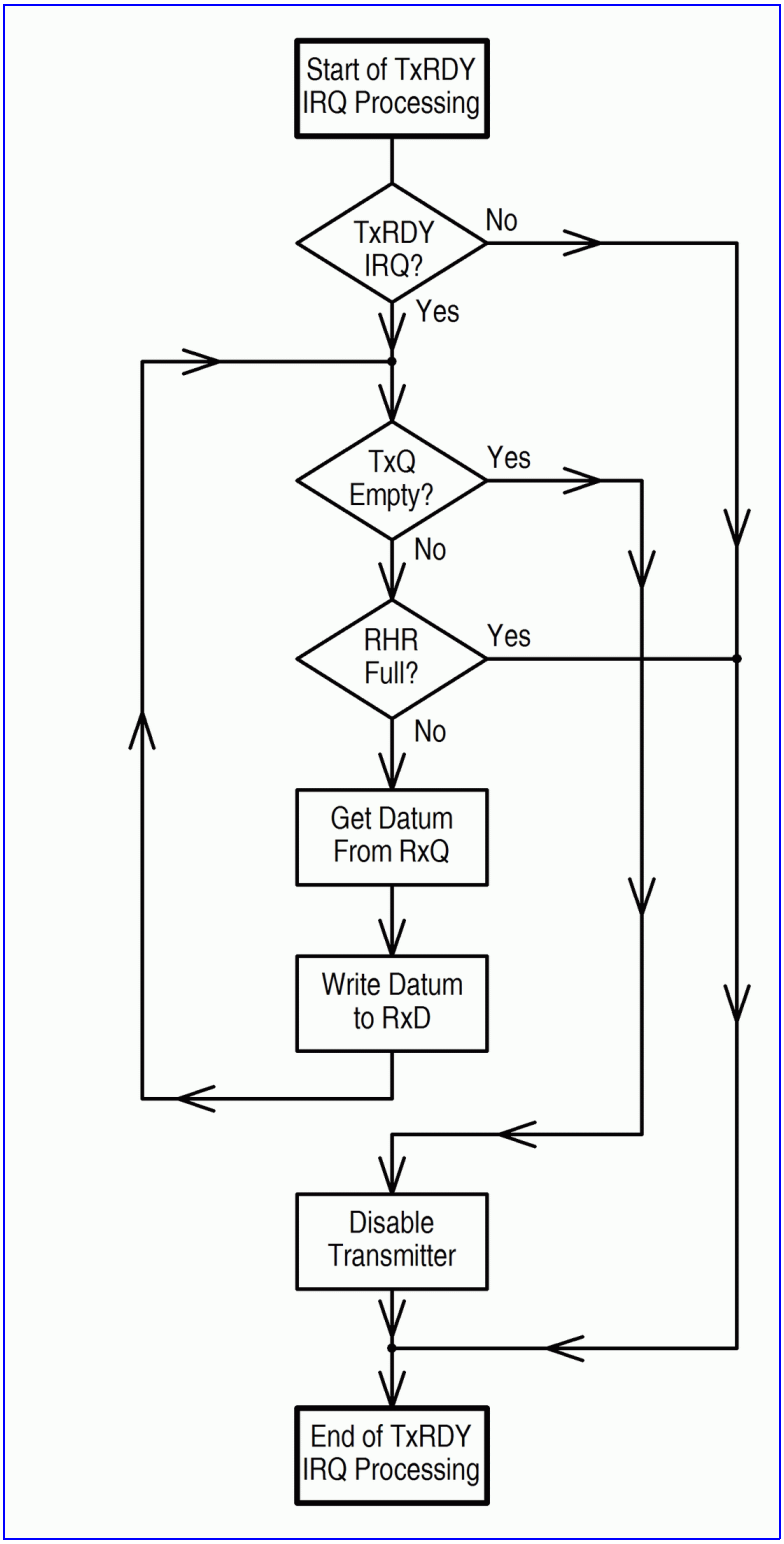
TxRDY processing starts with determining that the transmitter is interrupting. Next, a loop is entered in which TxQ is checked to see if it contains at least one datum. If a datum is available, the THR is checked to see if it can accept it and if so, a datum is gotten from TxQ and written to TxD, the write also clearing the TxRDY interrupt. The sequence repeats until TxQ is empty or THR is full.

The code required to service a TxRDY IRQ is only slightly more complex than that for the RxRDY interrupt, and also draws upon code elements that have already been presented. Refer to page 35 for an example that is suitable for both the 65C02 and 65C816.



RxRDY Interrupt Processing Flow

```
;  process RxRDY interrupt...
;
not_ct   bit #mx_rxirq          ;receiver interrupting?
         beq not_rx             ;no, skip receiver
;
.0000010 lda nx_ubase+nx_sr     ;get channel status
         bit #mx_rxrdy          ;any datums in RHR?
         beq .0000020           ;no, done with RxD
;
         lda nx_ubase+nx_sio    ;read datum from RxD
         ldx nx_rxput           ;RxQ "put" index
         inx                    ;"put+1"
         cpx nx_rxget           ;RxQ "get" index
         beq .0000010           ;RxQ is full, discard datum
;
         dex                    ;realign "put" index
         sta nx_rxq,x           ;put datum into RxQ
         inc nx_rxput           ;increment "put" index
         bra .0000010           ;loop for another
;
.0000020 lda nx_ubase+nx_isr    ;reload interrupt status
;
not_rx   ...program continues...
```



TxRDY Interrupt Processing Flow

```
;  process TxRDY interrupt...
;
not_rx   bit #mx_txirq          ;transmitter interrupting?
         beq not_rx             ;no, skip transmitter
;
         ldx nx_txget           ;current TxQ "get" index
;
.0000010 cpx nx_txput           ;any datums in TxQ?
         beq .0000020           ;no, done with transmitter
;
         lda nx_ubase+nx_sr     ;yes, get channel status
         bit #mx_txrdy          ;space in THR?
         beq .0000030           ;no, done for now
;
         lda nx_txq,x           ;get datum from queue &...
         sta nx_ubase+nx_thr    ;write to TxD
         inx                    ;"get" +1
         bra .0000010           ;loop
;
.0000020 lda #cx_txdis          ;shut down transmitter...
         sta nx_ubase+nx_cr     ;to suppress IRQs
         lda #mx_txdis          ;set transmitter...
         sta nx_txstw           ;disabled flag
;
.0000030 stx nx_txget           ;save final TxQ "get" index
;
not_tx   ...other ISR processing continues...
```

4.4 Foreground processing

The foreground part of the driver processes serial input/output requests on behalf of other applications. Applications call the driver's `sioget` function to receive a datum, or the driver's `sioput` function to transmit a datum.

4.4.1 Processing a receive request

Processing a receive request involves attempting a "get" on RxQ. If the "get" succeeds, a datum will be returned in .A and carry will be cleared. Otherwise, carry will be set and .A will be unchanged. Carry will also be set if the driver initialization function has not been executed. .X and .Y will be preserved. See the next page for the function example.

```
;sioget: PROCESS RECEIVE REQUEST

sioget   sec                    ;assume error
         bit nx_ubase+nx_ureg   ;driver initialized?
         bpl .0000020           ;no, error
;
         phx                    ;preserve
         ldx nx_rxget           ;RxQ "get" index
         cpx nx_rxput           ;RxQ "put" index
         beq .0000010           ;no datum available
;
         lda nx_rxq,x           ;get datum from RxQ
         inc nx_rxget           ;increment index
         clc                    ;datum gotten
;
.0000010 plx                    ;restore
;
.0000020 rts                    ;return to caller
```

In the above, the test of `nx_ureg` verifies that initialization was performed. As previously noted, `nx_ureg` will be conditioned to `$0F` (`%00001111`) by the UART hardware at power-on or reset. `%10000000` will be written into this register when the initialization process has been completed. The balance of the code is based upon the I/O mechanics discussed at §4.2.

As presented, `sioget` will work with either processor, assuming the 65C816 register widths have been set to eight bits by the calling function. It may be necessary to add instructions to the 65C816 version of `sioget` to account for register widths, as well as the state of DB and DP, either of which may not be set to agree with the driver's operating environment—such instructions will be omitted here.

4.4.2 Processing a transmit request

Processing a transmit request involves a "put" on TxQ and management of the transmitter, which may have been disabled in the ISR. Upon return, carry will be cleared to indicate that the datum was accepted and processed. Carry will be set if the driver initialization function has not been executed. All registers will be preserved. See the next page for the function example.

```
;sioput: PROCESS TRANSMIT REQUEST

sioput   sec                    ;assume error
         bit nx_ubase+nx_ureg   ;driver initialized?
         bpl .0000030           ;no, error
;
         pha                    ;preserve
         phx                    ;preserve
         ldx nx_txput           ;TxQ "put" index
         inx                    ;"put" +1
;
.0000010 cpx nx_txget           ;TxQ "get" index
         beq .0000010           ;TxQ is full, block
;
         dex                    ;realign "put" index &...
         sta nx_txq,x           ;put datum into TxQ
         inc nx_txput           ;bump "put" index
;
;
;  manage the transmitter...
;
         lda #mx_txdis          ;transmitter disabled flag
mask
         trb nx_txstw           ;is transmitter disabled?
         beq .0000020           ;no
;
         lda #cx_txenb          ;yes, enable...
         sta nx_ubase+nx_cr     ;it
;
;
;  clean up & exit...
;
.0000020 plx                    ;restore
         pla                    ;restore
         clc                    ;operation completed
;
.0000030 rts                    ;return to caller
```

Note how the MPU will enter a “spin” or “busy” loop at `.0000010` as long as TxQ is full, which means `sioput` will block until the datum can be accepted for transmission. The alternative would be to have `sioput` immediately return with carry set, but doing so would then produce an ambiguous result, since carry will also be set if `sioput` is called before the driver has been initialized. In any system that does not multitask the above algorithm is acceptable, even though it ties up the MPU until the datum can be accepted.

Once the datum has been put into TxQ `sioget` manages the transmitter. Recall that if TxQ is empty when a TxRDY interrupt occurs, the ISR will disable the transmitter to suppress the

IRQs and will also set the flag at `nx_txstw` to indicate that the transmitter has been shut down.

Hence `sioput` will check the `nx_txstw` transmitter status flag after each "put" to TxQ and if necessary, enable the transmitter. The TRB instruction is convenient in this case, as it reports the state of `nx_txstw` before modifying it.

4.5 Driver initialization

Initialization prepares driver data structures for use and configures the 28L91, procedures that would be carried out once during system boot. The initialization process will be described in three subsections: data structure preparation, UART initialization and configuration, and UART setup parameters.

4.5.1 Data structure preparation

At boot time, the driver's data must brought to a known state. This is especially important with the RxQ and TxQ "get" and "put" indices, which will contain random values at power-on. If these indices are not properly initialized it will appear that datums are present in the queues, possibly causing errors after the UART has been initialized and configured. Similarly, the uptime counter, which will also have random values at power-on, must be cleared, since uptime always starts at zero seconds.

Below is a data structure initialization function, using definitions that were previously presented on page 28).

```
         ldx #nx_zend-nx_zbase-1
;
.0000010 stz nx_zbase,x        ;clear driver ZP workspace
         dex
         bpl .0000010
;
         lda #hz               ;jiffy IRQ rate
         sta nx_jifct          ;initialize jiffy counter
```

4.5.2 UART initialization and configuration

Following power-on or hard reset, a number of the 28L91's registers will be in an undefined state. The 28L91 does not have a default operating mode following reset, which means most of the configurable registers must be loaded with parameters that are suitable for your system. There are several ways to go about doing so, the most efficient method being the use of a data table that pairs a register offset with the parameter that is to be loaded into that register. For example, here is an entry that configures the interrupt mask register (§3.2.4).

```
         .byte nx_imr,nxpiqmsk ;IMR (enables IRQs)
```

The first field, `nx_imr`, is the register offset and the second field, `nxpiqmsk`, is the parameter to be loaded into the register.

`nxpiqmsk` is defined in a separate INCLUDE file that contains the definitions for all parameters to be loaded into the UART at boot time. Defining the individual parameters independently of the setup data table simplifies the process of making changes.

In addition to the table's register-and-parameter pairs, a "sentinel" entry should be included to mark the end of the table (EOT)—the EOT sentinel will be the final entry in the table. When the UART setup function encounters the EOT sentinel it will know when to stop. For example, here is a data table with one register-and-parameter pair and the EOT sentinel.

```
nxpsutab .byte nx_imr,nxpiqmsk ;IMR (enables IRQs)
         .byte _eotsen_        ;EOT sentinel
```

`eotsen` is defined as `$FF`. As there is no register `$FF` in any NXP UART that value is unambiguous.

Using the above information, a relatively simple function may be written to set up the UART. The following code continues from the previous code that initialized the driver's data structures.

```
         ldy #0                ;UART setup data table index
;
.0000030 ldx nxpsutab,y        ;load register from data table
         cpx #_eotsen_         ;EOT?
         beq .0000040          ;yes, UART configured
;
         iny                   ;no, index to &...
         lda nxpsutab,y        ;load register's parameter
         sta nx_ubase,x        ;write parameter to register
         iny                   ;index to...
         bra .0000030          ;next table entry
;
.0000040 cli                   ;allow IRQs
```

Study of the above function will quickly make it clear that if the EOT sentinel is missing from the data table the MPU will get stuck in an unbreakable loop. Upon completion of UART setup, IRQs are enabled and the serial interface subsystem is ready for service.

4.5.3 UART setup parameters

In this section, recommended setup parameters for the 28L91 and an example setup data table will be presented. The following serial interface characteristics are assumed:

- Data rate is 115,200 bps.
- Data format is 8N1.
- Hardware flow control is enabled.
- C/T is operated in timer mode and interrupts 100 times per second (`hz = 100`).

The following table will list the parameter(s) for each register, along with an explanation as warranted. In some cases, multiple parameters will be defined for the same register, the use of which will be explained in the subsection on organizing the data table.

Register	Parameter	Value	Explanation
nx_acr	nx_paacr	%01100000	a) use bit rate table #1 b) run C/T in timer mode c) C/T clock source is the X1 clock
nx_cr	nx_crpa	%10010000	deassert RTS
nx_cr	nx_crpb	%00100000	reset receiver, also disables it
nx_cr	nx_crpc	%00110000	reset transmitter, also disables it
nx_cr	nx_crpd	%01010000	reset received break change IRQ
nx_cr	nx_crpe	%01000000	reset error status
nx_cr	nx_		

Here is the driver initialization function in its entirety.

```
;sioinit: INITIALIZE UART DRIVER
;
sioinit  sei                    ;ignore IRQs until ready
         stz nx_ubase+nx_ureg   ;driver not initialized
;
;
;  initialize driver data structures...
;
         ldx #nx_zend-nx_zbase-1
;
.0000010 stz nx_zbase,x         ;clear driver ZP workspace
         dex
         bpl .0000010
;
         lda #hz                ;jiffy IRQ rate
         sta nx_jifct           ;initialize jiffy counter
         ldy #0                 ;UART setup data table index
;
.0000030 ldx nxpsutab,y         ;load register from data table
         cpx #_eotsen_          ;EOT?
         beq .0000040           ;yes, UART configured
;
         iny                    ;no, index to &...
         lda nxpsutab,y         ;load register's parameter
         sta nx_ubase,x         ;write parameter to register
         iny                    ;index to...
         bra .0000030           ;next table entry
;
.0000040 lda #mx_drivr          ;indicate that driver...
         sta nx_ubase+nx_ureg   ;has been initialized
         cli                    ;allow IRQs
         clc                    ;all okay
         rts                    ;return to caller
```