UART Wildcard User Guide

Version 1.2.1
December, 2002
Copyright Mosaic Industries, Inc.
All rights reserved

UART Module User Guide 1

UART Wildcard User Guide 1

Serial Communications Basics 1

RS232 2

RS422 3

RS485 3

UART Module Hardware 3

Connecting To the Wildcard Carrier Board 4

Selecting the Module Address 4

RS422/485 Configuration Jumpers 5

Protocol Configuration and Direction Control Registers 5

Modem Handshaking Signals 6

UART Module Field Header 7

Cable Connections 8

Software 10

Overview of the Software Device Driver Functions 10

Installing the UART Module Driver Software 12

Using the Driver Code with C 13

Using the Driver Code with Forth 13

UART Direction Control in a Multitasking System 14

Glossary 16

Overview of Glossary Notation 16

Glossary Quick Reference 17

Glossary Entries 18

C Demonstration Program 24

Forth Demonstration Program 26

Hardware Schematics 29

UART Wildcard User Guide

The UART Module implements two full-duplex serial ports that can be configured for RS232, RS422, and RS485 protocols. This tiny 2" by 2.5" board is a member of the Wildcard™ series that connects to the QED Board or PanelTouch Controller host via the Wildcard Carrier Board, or connects directly to the EtherSmart™ Controller.

This document describes the capabilities of the UART Module, tells how to configure the hardware, and presents an overview of the driver software. A glossary of the software device driver functions, demonstration program source code, and complete hardware schematics are included.

UART Wildcard Specifications	
Ports:	Two full-duplex serial ports, each capable of simultaneous transmission and reception
Protocols:	RS232, RS422, and RS485
Baud Rates:	Standard baud rates up to 56000 baud
Buffers:	Dual 16-byte FIFO (first-in/first-out) buffers on each port
Handshaking:	Optional handshaking signals enable a modem connection for remote communications via any phone line
Drivers:	Precoded communications software

Serial Communications Basics

A "UART" is a "Universal Asynchronous Receiver/Transmitter" that converts parallel data from the host processor (in this case, the QED Board, PanelTouch Controller, or EtherSmart Controller) into a serial data stream. Each of the two UARTs on the module is capable of "full duplex" communications, meaning that both transmission and reception can occur simultaneously (although the RS485 protocol is half duplex as explained below). In other words, each "local" UART on the module can both send data to and receive data from a "remote" UART on the other end of a connecting serial cable. The local and remote must share a common ground, so all serial cables include at least one ground conductor.

The words "port" and "channel" are used interchangeably to refer to a serial communications link.

Baud Rate

The serial interface is asynchronous, meaning that there is no clock transmitted along with the data. Rather, the transmitter and receiver must be communicating using a known "baud rate", or bit frequency. Both the local and remote UARTs must be configured for the same baud rate. Software-selectable baud rates up to 56,000 baud are supported. Standard attainable baud rates are 300, 1200, 2400, 4800, 9600, 19200, 38400 and 56000 baud.

Data Format

Serial data is shifted out least-significant-bit first. At the UART, a logic high (5 volts) is called a "mark", and a logic low (0 volts) is called a "space". The serial output at the UART idles at the logic high (mark) level. A logic-low "start bit" marks the start of a character, followed by 5 to 8 data bits per character. An optional

"parity" bit can be specified to enable error detection by the UART. One to two logic-high "stop bits" mark the end of a character.

Parity options include even, odd, high, low, or no parity. Even parity means that the bits sum to an even number, and odd parity means that the bits sum to an odd number. High (mark) parity means that the parity bit is always logic 1 at the UART, and low (space) parity means that the parity bit is always logic 0 at the UART. No parity means that there is no parity bit.

A "break" sequence forces the serial output to a logic low (space) at the UART.

Both the local and remote UARTs must be configured for the same communications parameters. The standard data format for the QED product line is 8 data bits, no parity, and 1 stop bit.

FIFOs

Each UART implements transmit and receive "FIFO" buffers to reduce the required frequency of service by the host processor. A FIFO is a First In/First Out buffer that can queue a burst of outgoing characters for transmission, or save a set of incoming characters until the host can read them. Each of the two channels on the UART Module implements two 16-character FIFOs, one for outgoing characters and one for incoming characters.

Serial Driver Chips

The serial data stream at the UART is conditioned by serial driver chips that transmit and receive the data. The resulting signal levels on the interface cable connect the local and remote in a manner specified by a standard "protocol". The most widely used protocol is RS232, a full duplex protocol with a single-ended bipolar voltage swing on the serial cable. Newer protocols include the full duplex RS422 and the half duplex RS485 protocols, each of which drives differential 0 to 5 volt signals on the serial cable.

Each of the two channels on the UART module can be configured for RS232, RS422, or RS485. The protocols are described in turn.

RS232

Each RS232 driver uses inverting logic and implements a single-ended bipolar output voltage (that is, one signal that swings above and below ground). A logic high at the UART is converted by the RS232 output driver to a voltage between –7 and –12 volts, while a logic low at the UART is converted to a voltage between +7 and +12 volts. The RS232 receiver accepts signals in the range –12 to +12 volts and outputs logic level (0 or 5 volt) signals to the UART circuitry. The RS232 driver and receiver use separate conductors on the serial cables, enabling full duplex communications. Note that the local and the remote must share a common ground, so a minimum of 3 wires are required for full duplex RS232 communications: a transmit wire, a receive wire, and a common ground.

As of the time of this writing, more detailed information about the RS232 protocol is available at http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html.

Optional RS232 Modem Interface

A modem (modulator/demodulator) provides a way of encoding digital data as a set of audio signals that can be sent over a telephone line. Most modems communicate using RS232 and a set of hardware "handshaking" signals used to regulate data flow. The UART Module implements these optional RS232 modem handshaking signals on channel 1. The handshaking signals can be disabled and/or ignored by applications that do not need them.

Due to hardware constraints, if modem handshaking is needed on UART channel 1, then channel 1 must be configured for RS232, and channel 2 cannot be configured for RS232 communications. These factors are discussed in more detail in the "UART Module Hardware: Modem Handshaking Signals" section below.

RS422

Each RS422 driver transmits a differential pair of output signals at 0 and 5 volts. The RS422 receiver converts the differential signal to the 0 to 5 volt logic signal required by the UART circuitry. Because differential signals have inherently better signal-to-noise properties, reliable RS422 communications can be sent over much longer distances compared to RS232. The RS422 driver and receiver use separate differential conductor pairs on the serial cables, enabling full duplex communications. Note that the local and the remote must share a common ground, so a minimum of 5 wires are required for full duplex RS422 communications: two transmit wires, two receive wires, and a common ground.

RS485

RS485 is a half duplex version of RS422 that is capable of multi-drop operation. RS485 uses the same differential signaling scheme as RS422, and hence has the same superior signal-to-noise characteristics and range described above. In fact, a single driver chip on the UART Module is used to implement both RS422 and RS485 communications for a given serial channel. In RS485 mode, the RS422 transmit and receive pairs are shorted together with a pair of onboard jumpers as explained in the "UART Module Hardware: RS485 Jumpers" section below. Note that the local and the remote must share a common ground, so a minimum of 3 wires are required for half duplex RS485 communications: a pair of transceive wires and a common ground.

Multi-drop RS485

Because a single pair of conductors is used for both transmission and reception, RS485 is useful for "multi-drop" applications in which a "master" communicates with multiple "slave" serial devices, or "nodes".

To avoid contention on the RS485 bus, the application software must assure that only one transmitter is enabled at a time. The master is in charge of designating which receiver is on at any one time. In the simplest scheme, all RS485 transceivers come up in receive mode when the interface is initialized, and each transceiver node has a unique address known to it and the master. A single master can broadcast commands to all the slaves, and can direct commands to an individual slave using its unique address. The master can instruct a single slave to go into transmit mode, and then the master can put itself into receive mode, thereby allowing the master to retrieve data from the slave. After the slave transmission is complete, the slave puts itself back into receive mode so that the master can transmit additional commands. In this manner, data can be exchanged between the master and each slave on the bus.

Care must be taken when changing an RS485 node from transmit mode to receive mode, or visa versa. Characters may be lost if the transmit driver is turned off while pending characters are still being transmitted. To avoid this problem, the software function named `RS485_Rcv_When_Xmit_Done` that establishes the RS485 receive mode is coded such that it waits until all queued (pending) characters have been transmitted before the driver chip is taken out of transmit mode. Similarly, when establishing the transmit mode, the application software is responsible for ensuring that the RS485 receiver is not disabled until all expected characters have been received.

UART Module Hardware

The UART Module comprises a Wildcard bus header, field header, digital logic circuitry, a dual UART chip, and driver/receiver circuitry for the RS232, RS422, and RS485 protocols. Jumpers enable module address

selection, conversion from RS422 to RS485, and the installation of optional RS422/485 termination networks. The Wildcard bus header interfaces to the host processor (QED Board, Panel-Touch Controller, or EtherSmart Controller), and the field header brings out the serial I/O signals for the two serial channels.

Connecting To the Wildcard Carrier Board

To connect the UART Module to the Wildcard Carrier Board, follow these simple steps:

1. Connect the Wildcard Carrier Board (also known as the "Module Carrier Board") to the QED Board as outlined in the "Module Carrier Board Users Guide".

2. With the power off, connect the 24-pin Module Bus on the UART Module to Module Port 0 or Module Port 1 on the Wildcard Carrier Board. The corner mounting holes on the module should line up with the standoffs on the Wildcard Carrier Board. The module ports are labeled on the silkscreen of the Wildcard Carrier Board. Note that the UART Module headers are configured to allow direct stacking onto the Wildcard Carrier Board, even if other modules are also installed. Moreover, the latest version of the Wildcard Carrier Board is designed to directly stack onto the QED Board. Do not use ribbon cables to connect the UART Module to the Wildcard Carrier Board. Use of ribbon cables on the UART Module's field header is fine.

   CAUTION: The Wildcard Carrier Board does not have keyed connectors. Be sure to insert the module so that all pins are connected. The Wildcard Carrier Board and the UART Module can be permanently damaged if the connection is done incorrectly.

Selecting the Module Address

Once you have connected the UART Module to the Wildcard Carrier Board, you must set the address of the module using jumper shunts across J1 and J2.

The Module Select Jumpers, labeled J1 and J2, select a 2-bit code that sets a unique address on the module port of the Wildcard Carrier Board. Each module port on the Wildcard Carrier Board accommodates up to 4 modules. Module Port 0 provides access to modules 0-3 while Module Port 1 provides access to modules 4-7. Two modules on the same port cannot have the same address (jumper settings). Table 1-1 shows the possible jumper settings and the corresponding addresses.

Table 1-1 Jumper Settings and Associated Addresses

Module Port	Module Address	Installed Jumper Shunts
0	0	None
	1	J1
	2	J2
	3	J1 and J2
1	4	None
	5	J1
	6	J2
	7	J1 and J2

RS422/485 Configuration Jumpers

The remaining jumpers on the UART Module enable conversion from RS422 to RS485, and the installation of optional RS422/485 termination networks. There are three jumpers for each serial port. The silkscreen on the UART Module designates "Port1" and "Port 2" as the two jumper regions. Within each region is a pair of jumpers labeled "RS485", and an additional jumper labeled "Term", which stands for "termination".

RS485 Jumpers

To use the half duplex RS485 protocol on a given serial port, install the two jumper caps on the two jumpers labeled "RS485" in that port's jumper area (labeled "Port1" or "Port2"). To use RS422 or RS232, remove the jumper caps from the "RS485" jumpers. The remaining protocol configuration is software programmable as described below.

Optional RS422/485 Termination Network Jumpers

Each serial communications link can be thought of as a "transmission line" with a characteristic impedance. When the signals on the transmission line encounter a non-matching impedance, a portion of the signal energy is reflected, thereby degrading the signal-to-noise ratio of the communications link. One advantage of the differential signaling scheme used by the RS422 and RS485 protocols is that a straightforward termination network can be installed at the end of the transmission line to minimize unwanted reflections. The UART Module provides a default termination network for each serial port. To install the termination network, simply place a jumper cap on the jumper labeled "Term" (termination) in that port's jumper area (labeled "Port1" or "Port2"). This places a 100 Ω resistor in series with a 0.1 μF capacitor across the differential signal conductors. The capacitor blocks DC current that would greatly increase the power drain of the circuit. The 100 Ω resistor provides the effective termination resistance seen by the high frequency bit transitions.

Please note that only one termination network should be installed on a multi-drop RS485 link, and it should be at the last node in the network. At high baud rates, the termination network may reduce the amplitude of the received pulses, so make sure that you test any termination scheme that you put in place in your custom application.

Protocol Configuration and Direction Control Registers

Aside from the RS485 and termination network jumpers described above, all remaining protocol selections are performed under software control by writing to control registers implemented by logic on the module. It is not necessary to understand this implementation detail, as the pre-coded functions described in the Glossary transparently manage these registers for you. This information is presented for those who want to know how the low level hardware works.

The programmable logic chip on the UART Module implements four registers named CONFIG1, CONFIG2, DIRECTION1, and DIRECTION2. They are addressed sequentially starting at offset 0x10 in the module's address space. The CONFIG registers configure the associated serial channel's protocol. The DIRECTION registers are used to control the direction of the RS422/485 driver chips by enabling and disabling the channel's receiver and transmitter. Table 1-2 describes the contents of these registers. The D4 through D0 labels at the top of the table designate the data bits that access the information in the register.

Table 1-2 Protocol and Direction Control Registers

Name	D4	D3	D2	D1	D0
CONFIG1	Modem	Handshake_En	RS232_En	Ch1_RS232	Ch1_RS4xx
CONFIG2				Ch2_RS232	Ch2_RS4xx
DIRECTION1				Ch1_Rcv_Bar	Ch1_Xmit
DIRECTION2				Ch2_Rcv_Bar	Ch2_Xmit

CONFIG1 and CONFIG2 configure the protocols for serial Channel1 and Channel2, respectively. In both configuration registers, D1 is set true in RS232 mode and false otherwise, and D0 (labeled RS4xx) is set true in RS422 or RS485 mode, and false otherwise. Recall that RS422 and RS485 share a driver/receiver chip. These bits determine which input receiver drives the serial input pin on the UART chip for the specified channel.

When set, the RS232_En bit in CONFIG1 powers up the serial driver chip that handles the transmit and receive signals for Channels 1 and 2. When set, the Handshake_En bit powers up the serial driver chip that handles the modem handshaking signals for Channel1. The Modem bit is set to true if the optional modem handshaking signals are in use. When set, this bit causes the hardware to route the incoming signal from the RxD2/DCD1 pin on the field header to the /DCD1 pin on the UART. Handling of the modem signals is described in more detail in the next section.

The DIRECTION1 and DIRECTION2 registers control the transmitter and receiver enable signals for the RS422/RS485 driver/receiver chips for Channels 1 and 2, respectively. The Ch1_Xmit and Ch2_Xmit turn the respective transmitters on when the bit is set, and the active low Ch1_Rcv_Bar and Ch2_Rcv_Bar signals turn the respective receivers on when the bit is clear. In RS422 mode, both the driver and the receiver are enabled, thereby enabling full duplex communications. In RS485 mode, the driver is off and the receiver is on in receive mode, and the driver is on and the receiver is off in transmit mode.

The configuration and direction registers are transparently managed by the functions described in the Glossary below.

Modem Handshaking Signals

Most users of the UART Module implement direct computer-to-computer communications via a serial cable. In these cases, hardware handshaking signals are typically not required. Rather, frequent polling or software-controlled handshaking by means of the “XON” and “XOFF” characters serves to ensure that the receiving computer is not overwhelmed by the incoming data stream.

Some applications, however, require the use of hardware handshaking signals that enable the sender and receiver to signal their readiness to send and receive data. Communication over a telephone link via an RS232 modem is a prime example of such an application. The UART Module implements five handshaking signals defined by the RS232 protocol to support a modem connection on serial Channel 1. If these modem handshaking signals are enabled, serial Channel 2 cannot be used for independent RS232 communications; however, it can still be used for RS422 or RS485.

DTR, DSR, RTS, and CTS are handshaking signals that can be set and checked by the application software to control the flow of data. These signals are buffered by a single chip that can be powered down by the `Set_Protocols` function if the handshaking signals are not in use.

DCD is an input from the modem that becomes active when an incoming data stream is detected on the phone line. DCD shares the RS232 receiver for serial channel 2; thus, Channel 2 cannot be configured for RS232 if

the modem configuration is selected for Channel 1. The protocol initialization software function `Set_Protocols` is smart enough to detect an invalid protocol specification, returning an error parameter if it is attempted.

The supported modem signals are listed in Table 1-3.

Table 1-3 Modem Handshaking Signals

Signal	Description	Direction	Mates with…
DTR	Data Terminal Ready	output	DSR
DSR	Data Set Ready	input	DTR
RTS	Ready To Send	output	CTS
CTS	Clear To Send	input	RTS
DCD	Data Carrier Detect	input	

All of the signals listed in Table 1-3 are "active low". That is, when active they are at a logic low (0 volts) at the UART chip, which corresponds to a positive voltage on the RS232 cable.

The Glossary includes functions that control and monitor the signals listed in Table 1-3.

UART Module Field Header

The serial communications signals are brought out to a 24-pin dual row header on the UART Module as shown in Table 1-4.

Table 1-4 UART Module Field Header

Signal	Pins		Signal
TxD1	– 1	2 –	RxD1
GND	– 3	4 –	GND
RCV1-	– 5	6 –	RCV1+
XMIT1-	– 7	8 –	XMIT1+
DTR1	– 9	10 –	DSR1
RTS1	– 11	12 –	CTS1
V+RAW	– 13	14 –	GND
+5V	– 15	16 –	+5V
TxD2	– 17	18 –	RxD2/DCD1
GND	– 19	20 –	GND
RCV2-	– 21	22 –	RCV2+
XMIT2-	– 23	24 –	XMIT2+

TxD1 and RxD1 are the RS232 transmit and receive signals, respectively, for Channel 1. TxD2 and RxD2 are the RS232 transmit and receive signals for Channel 2.

RCV1+ and RCV1- are the Channel 1 differential RS422/485 receive signals. XMIT1+ and XMIT1- are the Channel 1 differential RS422/485 transmit signals. To implement an RS485 protocol on Channel 1, install the

jumper caps on the jumpers labeled “RS485” in the “Port1” jumper area on the UART Module. This connects RCV1+ to XMIT1+, and connects RCV1- to XMIT1- to implement the half-duplex serial link on Channel 1.

RCV2+ and RCV2- are the Channel 2 differential RS422/485 receive signals. XMIT2+ and XMIT2- are the Channel 2 differential RS422/485 transmit signals. To implement an RS485 protocol on Channel 2, install the jumper caps on the jumpers labeled “RS485” in the “Port2” jumper area on the UART Module. This connects RCV2+ to XMIT2+, and connects RCV2- to XMIT2- to implement the half-duplex serial link on Channel 2.

DTR1, DSR1, RTS1, and CTS1 are the optional modem handshaking signals. The DCD1 (data carrier detect) modem signal is shared with RxD2 on pin 18 of the header. The hardware automatically routes this signal to the DCD input on the Channel1 UART if Channel1 is configured for the RS232 modem protocol.

Cable Connections

This section presents some suggested cable connections for common protocols. Each application will typically require a custom cable assembly to meet the needs of that project.

Table 1-5 Cable for Dual RS232 25-pin D Connectors

Signal Name at UART Module	Pin# at UART Module Field Connector	Pin# on 25 pin Channel1 Connector	Pin# on 25 pin Channel2 Connector	Signal Name at Remote
TxD1	1	3		RxD
RxD1	2	2		TxD
DGND	3	1		FGND
DGND	4	7		SGND
TxD2	17		3	RxD
RxD2	18		2	TxD
DGND	19		1	FGND
DGND	20		7	SGND
		4	4	RTS
		5	5	CTS
		6	6	DSR
		20	20	DTR

Table 1-5 illustrates the connections required to implement two RS232 ports, each using a standard 25-pin D-type connector. Note that TxD at the local UART Module connects to RxD at the remote, and the local RxD connects to TxD at the remote. Also note that the RS232 handshaking signals RTS and CTS are connected to each other with a shorting wire at the D connector. Similarly, the RS232 handshaking signals DSR and DTR are connected to each other at the D connector. This standard scheme makes the remote computer think that it is always OK to send and receive data.

A cable that supports a modem on channel 1 is shown in Table 1-6. Note that the handshaking signals are brought out to the 25 pin D-connector. The CTS1 (clear to send) input to the UART Module connects to the RTS (ready to send) output from the remote. The DSR1 (data set ready) input to the UART Module connects to the DTR (data terminal ready) output from the remote. The DTR1 and RTS1 outputs from the UART Module connect to the DSR and CTS inputs to the remote, respectively. The DCD (data carrier detect) modem output connects to the RxD2/DCD1 input on the UART Module. Because this pin is shared with RxD2, Channel2 cannot be configured for RS232 when a modem is in use. In this case, the user is free to implement RS422 or RS485 on Channel 2.

Table 1-6 Cable for an RS232 Modem via a 25-pin D Connector

Signal Name at UART Module	Pin# at UART Module Field Connector	Pin# on 25 pin Channel1 Connector	Signal Name at Remote
TxD1	1	3	RxD
RxD1	2	2	TxD
DGND	3	1	FGND
DGND	4	7	SGND
CTS1	12	4	RTS
RTS1	11	5	CTS
DTR1	9	6	DSR
DSR1	10	20	DTR
RxD2/DCD1	18	8	DCD

Table 1-7 Cable for Dual RS232 9-pin D Connectors

Signal Name at UART Module	Pin# at UART Module Field Connector	Pin# on 9 pin Channel1 Connector	Pin# on 9 pin Channel2 Connector	Signal Name at Remote
TxD1	1	2		RxD
RxD1	2	3		TxD
DGND	3	5		SGND
TxD2	17		2	RxD
RxD2	18		3	TxD
DGND	19		5	SGND
		7	7	RTS
		8	8	CTS
		6	6	DSR
		4	4	DTR

Table 1-7 illustrates the connections required to implement two RS232 ports, each using a 9-pin D-type connector as found on many laptops. Note that TxD at the local UART Module connects to RxD at the remote, and the local RxD connects to TxD at the remote. Also note that the RS232 handshaking signals RTS and CTS are connected to each other with a shorting wire at the D connector. Similarly, the RS232 handshaking signals DSR and DTR are connected to each other at the D connector. This standard scheme makes the remote computer think that it is always OK to send and receive data.

Table 1-8 Connections for Dual RS422 Serial Ports

Signal Name at UART Module	Pin# at UART Module Field Connector	Signal Name at Remote
RCV1+	6	XMIT1+
RCV1-	5	XMIT1-
XMIT1+	8	RCV1+

Signal Name at UART Module	Pin# at UART Module Field Connector	Signal Name at Remote
XMIT1-	7	RCV1-
GND	3 or 4	GND
RCV2+	22	XMIT+
RCV2-	21	XMIT2-
XMIT+	24	RCV2+
XMIT2-	23	RCV2-
GND	19 or 20	GND

Table 1-8 defines the connections for dual RS422 ports. Local RCV pins connect to remote XMIT pins, and the polarities of the local and remote pins match (+ to +, - to -).

Table 1-9 defines the connections for dual RS485 ports. Local XCV (transceive) pins connect to remote XCV pins, and the polarities of the local and remote pins match (+ to +, - to -). Note that the two "RS485" jumper caps must be installed for each port that is configured as RS485; these short RCV+ to XMIT+, and RCV- to XMIT- for the specified serial channel.

Table 1-9 Connections for Dual RS485 Serial Ports

Signal Name at UART Module	Pin# at UART Module Field Connector	Signal Name at Remote
RCV1+/XMIT1+	6 or 8	XCV1+
RCV1-/XMIT1-	5 or 7	XCV1-
GND	3 or 4	GND
RCV2+/XMIT2+	22 or 24	XCV2+
RCV2-/XMIT2-	21 or 23	XCV-
GND	19 or 20	GND

Software

A package of pre-coded device driver functions is provided to make it easy to control the UART Module. This code is available as a pre-compiled "kernel extension" library to C and Forth programmers. Both C and Forth source code versions of a demonstration program are provided. This demo program illustrates how to initialize and use the UART Module in an RS232 multitasking application.

Overview of the Software Device Driver Functions

The UART Module driver code makes it easy to configure the baud rate, data format, and protocol for each channel, send and receive characters, and control the direction of an RS485 channel. A demonstration program shows how to use the functions, and how to revector the serial primitives so that standard I/O print routines (such as `."` in Forth and `printf` in C) will automatically use a specified serial channel on the UART Module.

Most of the functions accept as input parameters the channel number (1 or 2), and the UART module number (0 through 7). The constants `CHANNEL1` and `CHANNEL2` are synonyms for 1 and 2, respectively; they are

provided to allow for clean readable code. The module number passed to the software functions must correspond to the hardware jumper settings as described in Table 1-1 above.

The initialization functions are `Set_UART_Number`, `Set_Baud`, `Set_Data_Format`, and `Set_Protocols`. `Set_UART_Number` sets a variable that holds the module number used by the channel-specific serial I/O routines described below; this value can be read using the function `Read_UART_Number`. `Set_Baud` sets the bit frequency to a specified integer value that can span standard rates from 300 up to 56000 bits per second. `Set_Data_Format` accepts parameters that specify the number of data bits, number of stop bits, and parity for the specified channel. A set of pre-defined constants makes it easy to specify the desired parity; they are: `NO_PARITY`, `EVEN_PARITY`, `ODD_PARITY`, `HIGH_PARITY`, and `LOW_PARITY`. See the "Serial Communications Basics: Data Format" section above for a description of the parity modes. The standard QED serial ports operate with 8 data bits, 1 stop bit, no parity, and a baud rate of 9600 or 19200.

The fundamental serial I/O routines are called `Emit_UART`, `Ask_Key_UART`, and `Key_UART`. Each accepts a channel number and module number as input parameters. `Emit_UART` also accepts an input character, which it queues in the outgoing FIFO for transmission on the specified serial channel. `Ask_Key_UART` tests whether an input character is pending in the receiver FIFO; if so, it returns a true (-1) flag, and if not, it returns a false (0) flag. `Key_UART` returns the next pending character in the input FIFO; if the FIFO is empty, it waits for a character and returns it.

The channel-specific serial I/O routines are called `Ch1_Emit`, `Ch2_Emit`, `Ch1_Ask_Key`, `Ch2_Ask_Key`, `Ch1_Key`, and `Ch2_Key`. These routines have the correct parameter list to enable revectoring of serial I/O functions as illustrated in the demonstration program. Each uses the module number that has been set using the Set_UART_Number function, and the channel number suggested by the function name. `Ch1_Emit` and `Ch2_Emit` queue a specified character in the outgoing FIFO for transmission. `Ch1_Ask_Key` and `Ch2_Ask_Key` test whether an input character is pending in the receiver FIFO; if so, a true (-1) flag is returned. `Ch1_Key` and `Ch2_Key` return the next pending character in the input FIFO.

Full duplex protocols (RS232 and RS422) operate with both transmitters and receivers active at all times. The half duplex RS485 protocol requires that the channel be either transmitting or receiving. The functions `RS485_Xmit_UART` and `RS485_Rcv_When_Xmit_Done` control the RS485 direction for the specified channel and module number. As expected, the former function establishes the transmit mode. The latter establishes the receive mode after all pending outgoing characters have been sent, thereby avoiding errors caused by shutting off the transmitter in the middle of a character transmission.

Additional functions allow the application to send break characters, and set and read the modem handshaking signals.

For detailed specifications on control of the communications port, refer to the UART data sheet (Texas Instruments Part# TL16C552A).

Demo Illustrates Initialization and Revectoring of Serial I/O

The demonstration program presents constants and functions that illustrate how to use the serial channels. The function named `Default_UART_Init` accepts a module number parameter, and initializes both serial channels to default values specified by the constants `DEFAULT_BITS_PER_CHAR`, `DEFAULT_STOP_BITS`, `DEFAULT_PARITY`, `DEFAULT_BAUDRATE`, `DEFAULT_PROTOCOL`, and `DEFAULT_MODEM_SUPPORT`. The values of these constants can be edited, and a more versatile initialization function can be crafted to meet the needs of your application.

The multitasking operating system on the QED Board, Panel-Touch Controller, and EtherSmart Controller allows each task to access its own distinct serial I/O channel by pointing the three primitives `Emit`, `Ask_Key` (called `?KEY` in Forth), and `Key` to the desired serial I/O handler functions. The advantage of

revectoring is that higher level serial management functions that accept and print characters and strings will then use the designated serial channel. The `Ch1_Monitor` and `Ch2_Monitor` functions in the demonstration perform the revectoring, and then call functions that perform interactive echoing via the specified serial channel on the UART Module. The `Run_Demo` function (or the main function in C) starts and runs the multitasking demonstration using both channels of the UART module. The demonstration program source code is presented in a later section.

Installing the UART Module Driver Software

The UART Module device driver software is provided as a pre-coded modular runtime library, known as a "kernel extension" because it enhances the on-board kernel's capabilities. The library functions are accessible from C and Forth.

Mosaic Industries can provide you with a web site link that will enable you to create a packaged kernel extension that has drivers for all of the hardware that you have on your system. In this way the software drivers are customized to your needs, and you can generate whatever combination of drivers you need. Make sure to specify the UART Module Drivers in the list of kernel extensions you want to generate, and download the resulting "packages.zip" file to your hard drive.

For convenience, a separate pre-generated kernel extension for the UART Module is available from Mosaic Industries on the Demo and Drivers media (diskette or CD). Look in the Drivers directory, in the subdirectory corresponding to your hardware (QED, PanelTouch, or EtherSmart), in the UART_Module folder.

The kernel extension is shipped as a "zipped" file named "packages.zip". Unzipping it (using, for example, winzip or pkzip) extracts the following files:

- readme.txt - Provides summary documentation about the library.
- install.txt - The installation file, to be loaded to COLD-started QED Board.
- library.4th - Forth name headers and utilities; prepend to Forth programs.
- library.c - C callers for all functions in library; #include in C code.
- library.h - C prototypes for all functions; #include in extra C files.

Library.c and library.h are only needed if you are programming in C. Library.4th is only needed if you are programming in Forth. The uses of all of these files are explained below.

We recommend that you move the relevant files to the same directory that contains your application source code.

To use the kernel extension, the runtime kernel extension code contained in the install.txt file must first be loaded into the flash memory of the QED Board. Start the QED Terminal software with the QED board connected to the serial port and turned on. If you have not yet tested your QED board and terminal software, please refer to the documentation provided with the QED Terminal software. Once you can hit enter and see the 'ok' prompt returned in the terminal window, type

```
COLD
```

to ensure that the board is ready to accept the kernel extension install file. Use the "Send File" menu item of the terminal to download the install.txt to the QED Board or Panel-Touch Controller.

Now, type

```
COLD
```

again and the kernel has been extended! Once install.txt has been loaded, it need not be reloaded each time that you revise your source code.

Using the Driver Code with C

Move the library.c and library.h files into the same directory as your other C source code files. After loading the install.txt file as described above, use the following directive in your source code file:

```
#include "library.c"
```

This file contains calling primitives that implement the functions in the kernel extension package. The library.c file automatically includes the library.h header file. If you have a project with multiple source code files, you should only include library.c once, but use the directive

```
#include "library.h"
```

in every additional source file that references the UART functions.

To load the optional demonstration program described above, use the "make" icon of the C compiler to compile the file named

UmodDemo.c

that is provided on the Demos and Drivers media. Use the terminal to send the resulting UmodDemo.txt file to the QED Board, and type `main` to run the program. See the demo source code listing below for more details.

Note that all of the functions in the kernel extension are of the _forth type. While they are fully callable from C, there are two important restrictions. First, _forth functions may not be called as part of a parameter list of another _forth function. Second, _forth functions may not be called from within an interrupt service routine unless the instructions found in the file named

\fabius\qedcode\forthirq.c

are followed. Also, in most cases Key and Emit functions should not be called from within interrupt service routines, because these routines call PAUSE, and use of PAUSE within an interrupt routine can halt the multitasker.

NOTE: If your compiler was purchased before June 2002, you must update the files, qlink.bat and qmlink.bat in your /fabius/bin directory on your installation before using the kernel extension. You can download a zip file of new versions at

http://www.mosaic-industries.com/Download/new_qlink.zip

The two new files should be placed in c:\Fabius\bin. This upgrade only has to be done once for a given installation of the C compiler.

Using the Driver Code with Forth

After loading the install.txt file and typing `COLD`, use the terminal to send the "library.4th" file to the QED Board. Library.4th sets up a reasonable memory map and then defines the constants, structures, and name headers used by the UART Module kernel extension. Library.4th leaves the memory map in the download map.

After library.4th has been loaded, the board is ready to receive your high level source code files. Be sure that your software doesn't initialize the memory management variables DP, VP, or NP, as this could cause memory conflicts. If you wish to change the memory map, edit the memory map commands at the top of the library.4th

file itself. The definitions in library.4th share memory with your Forth code, and are therefore vulnerable to corruption due to a crash while testing. If you have problems after reloading your code, try typing `COLD`, and reload everything starting with library.4th. It is very unlikely that the kernel extension runtime code itself (install.txt) can become corrupted since it is stored in flash on a page that is not typically accessed by code downloads.

We recommend that your source code file begin with the sequence:

```
WHICH.MAP 0=
IFTRUE 4 PAGE.TO.RAM  \ if in standard.map...
       5 PAGE.TO.RAM
       6 PAGE.TO.RAM
      DOWNLOAD.MAP
ENDIFTRUE
```

This moves all pre-loaded flash contents to RAM if the QED Board is in the standard (flash-based) memory map, and then establishes the download (RAM-based) memory map. At the end of this sequence the QED Board is in the download map, ready to receive additional code.

We recommend that your source code file end with the sequence:

```
4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
6 PAGE.TO.FLASH
STANDARD.MAP
SAVE
```

This copies all loaded code from RAM to flash, and sets up the standard (flash-based) memory map with code located in pages 4, 5 and 6. The `SAVE` command means that you can often recover from a crash and continue working by typing `RESTORE` as long as flash pages 4, 5 and 6 haven't been rewritten with any bad data.

UART Direction Control in a Multitasking System

RS485 multidrop communications are often used to establish communications with a remote instrument. Typically, a command and response protocol is used. The QED Board issues a command, usually terminated with a specified terminating character such as a carriage return, and the remote instrument then transmits a response back to the QED Board. It is the QED Board's responsibility to revert to the RS485 receive mode immediately after transmitting the terminating command character to the remote.

In multitasking RS485 applications, the time to go from transmit mode to receive mode on the RS485 link may be too long, causing characters to be dropped when the remote instrument responds to a command from the QED Board. This situation can arise because the task that controls RS485 communications is not running when the transmission of the terminating command character completes, and receive mode is not entered until the RS485 task regains control.

Of course, if a full duplex RS422 interface to the remote were available, this would solve the problem. Many instruments only support RS485, so we need a way to turn off the RS485 transmitter immediately after the last character in a command finishes transmitting.

Single Task Applications

To start the discussion, let's consider the simplest case of a one-task (non-multitasking) application. In this case, the simplest approach to RS485 communications yields good results.

With the RS485 channel in transmit mode (see the function RS485_Xmit_UART), we use CH1_Emit or CH2_Emit to send the entire command, including the terminating character. Each CH1_Emit or CH2_Emit places a character in the proper outgoing 16-byte FIFO (First-In/First-Out) buffer in the UART, and the UART automatically transmits them until the buffer is empty. All the serial driver routines are FIFO-aware, so the programmer doesn't have to worry about buffering; it's handled transparently. After emitting all the command characters, we revert to the receive mode using the command:

```
RS485_Rcv_When_Xmit_Done
```

which accepts the channel_num and module_num as input parameters. This command polls the XMIT_HAS_COMPLETED_MASK bit in the UART's line status register, PAUSEs if there are characters remaining to be sent, and changes to receive mode when the transmission is complete. This works well for a single-task system, as the PAUSE will simply loop back to the single task, and the receive mode will be entered as soon as the last character is transmitted.

Multitasking Applications

In a multitasking application we have to be more careful about time delays. For example, if there are 4 tasks running with the default 5 msec timeslice period, then each PAUSE invoked by RS485_Rcv_When_Xmit_Done will typically involve a 15 msec delay as the other 3 tasks use their timeslices. If the terminating command character is transmitted during one of these 15 msec delays, incoming characters from the remote instrument may be missed if the remote starts transmitting as soon as the terminating command character is received.

The solution is to ensure that the RS485 task maintains control while the very last character in the command string is being sent, so that the RS485 task can revert to receive mode with no time delay. To maintain multitasking efficiency, we should break the command string into two parts:

1. the command (not including terminator), and
2. the terminating character.

To facilitate this approach, an additional function has been added to the UART Module kernel extension Version 1.1. It is:

```
Loop_Until_Xmit_Done
```

which accepts the channel_num and module_num as input parameters. This function is very similar to Wait_Until_Xmit_Done, but it does not PAUSE while waiting. Using this function as described below, we are able to avoid task switches as the terminating character is transmitted and the RS485 receive mode is entered.

Assume we're using channel 1 of the UART module. We transmit the command string by repeatedly calling CH1_Emit (but we don't emit the terminating character yet). We call

```
Wait_Until_Xmit_Done
```

to wait and PAUSE until the transmission completes. Time delays due to task switches during this process should not matter, because the remote instrument is waiting for the terminating character to come in from the QED Board.

Next, call the following commands (of course, supply the proper parameters for each function):

```
DISABLE.INTERRUPTS     \ prevent task switching while sending last character
```

```
CH1_Emit                \ send the final terminating character of the command,
Loop_Until_Xmit_Done    \ don't pause, wait for terminating char to be sent
RS485_Rcv_UART          \ revert to receive mode
ENABLE.INTERRUPTS       \ re-enable interrupts
PAUSE                   \ this is optional, may boost multitasking efficiency
```

This sequence of function calls retains control while the last (terminating) character of the command is transmitted, then immediately enters the RS485 receive mode. Disabling interrupts should always be done sparingly, but is warranted in this case to ensure that no incoming characters are lost. We disable interrupts for only the time it takes to transmit one character, which at 19200 baud is about half a millisecond. As soon as `Loop_Until_Xmit_Done` is finished (meaning the character has been transmitted), `RS485_Rcv_UART` is called to immediately enter receive mode, and interrupts are re-enabled. At this point, the UART input FIFO buffer is able to accept up to 16 incoming characters before being serviced, so in some applications it may be efficient to PAUSE at this point to give other tasks a chance to run.

In summary, this approach should lead to high reliability acquisition of response data from remote RS485 instruments.

Glossary

This glossary defines important constants and functions from the driver code and demo program.

Overview of Glossary Notation

The main glossary entries presented in this document are listed in case-insensitive alphabetical order (the underscore character comes at the end of the alphabet). The keyword name of each entry is in **bold** typeface. Each function is listed with both a C-style declaration and a Forth-style stack comment declaration as described below. The "C:" and "4th:" tags at the start of the glossary entry distinguish the two declaration styles.

The Forth language is case-insensitive, so Forth programmers are free to use capital or lower case letters when typing keyword names in their program. Because C is case sensitive, C programmers must type the keywords exactly as shown in the glossary. The case conventions are as follows:

- Function names begin with a capital letter, and every letter after an underscore is capitalized. Other letters are lower case, except for capitalized acronyms such as "UART".
- Constant names and C macros use capital letters.
- Variable names use lower case letters.

Each glossary entry starts with C-style and Forth-style declarations, and presents a description of the function. Here is a sample glossary entry:

C: uchar **Key_UART** (int channel_num, int module_num)
4th: **Key_UART** (channel_num\module_num -- char)
Waits (if necessary) for receipt of a character from the specified channel and module, and returns the received character. PAUSEs while waiting. The returned byte is the next pending character in the FIFO (that is, the oldest unretrieved character in the receive FIFO). See also Ch1_Key and Ch2_Key.

The C declaration specifies that return data type before the function name, and lists the comma-delimited input parameters between parentheses, showing the type and a descriptive name for each.

The Forth declaration contains a "stack picture" between parentheses; this is recognized as a comment in a Forth program. The items to the left of the double-dash (--) are input parameters, and the item to the right of the double-dash is the output parameter. Forth is stack-based, and the first item shown is lowest on the stack. The backslash (\ ) character is read as "under" to indicate the relative positions of the input parameters on the stack. In the Forth declaration the parameter names and their data types are combined. All unspecified parameters are 16-bit integers. Forth promotes all characters to integer type.

The presence of both C and Forth declarations is helpful: the C syntax shows the types of the parameters, and the Forth declaration provides a descriptive name of the output parameter.

Glossary Quick Reference

Configuration Functions

int **Read_UART_Number**(void)
void **Set_Baud** (uint baudrate, int channel, int module)
void **Set_Data_Format** (int numbits, int stopBits, int parity, int channel, int module)
void **Set_UART_Number**(int module)
int **Set_Protocols** (int ch1_modem, int ch1_protocol, int ch2_protocol, int module)

Constants

CHANNEL1
CHANNEL2
EVEN_PARITY
HIGH_PARITY
LOW_PARITY
NO_PARITY
NOT_USED
ODD_PARITY
RS232
RS422
RS485

RS485 Direction Control

void **RS485_Rcv_UART** (int channel_num, int module_num)
void **RS485_Rcv_When_Xmit_Done** (int channel_num, int module_num)
void **RS485_Xmit_UART** (int channel_num, int module_num)

Serial I/O Primitives

int **Ask_Key_UART** (int channel_num, int module_num)
int **Ch1_Ask_Key** (void)
void **Ch1_Emit** (uchar character)
uchar **Ch1_Key** (void)
int **Ch2_Ask_Key** (void)
void **Ch2_Emit** (uchar character)
uchar **Ch2_Key** (void)
void **Emit_UART** (uchar character, int channel_num, int module_num)
uchar **Key_UART** (int channel_num, int module_num)

Utility Functions

void **End_Break** (int channel_num, int module_num)
void **Is_DTR** (int desired_state, int module_num)
void **Is_RTS** (int desired_state, int module_num)
int **Read_CTS** (int module_num)
int **Read_DCD** (int module_num)
int **Read_DSR** (int module_num)

void **Send_Break** (int channel_num, int module_num)
void **Wait_Until_Xmit_Done** (int channel_num, int module_num)

Demonstration Program Constants

DEFAULT_BAUDRATE **DEFAULT_BITS_PER_CHAR**
DEFAULT_MODEM_SUPPORT **DEFAULT_PARITY**
DEFAULT_PROTOCOL **DEFAULT_STOP_BITS**
UART_MODULE_NUM

Demonstration Program Functions

int **Default_UART_Init** (int module_num)
void **Run_Demo** (void)

Glossary Entries

C: int **Ask_Key_UART** (int channel_num, int module_num)
4th: **Ask_Key_UART** (channel_num\module_num -- flag)
Returns a flag indicating the receipt of a character on the specified serial channel. The flag is true (-1) if there is at least one character in the input FIFO of the specified channel. Otherwise the returned flag is false (0). See also Ch1_Ask_Key and Ch2_Ask_Key.

C: int **Ch1_Ask_Key** (void)
4th: **Ch1_Ask_Key** (-- flag)
Returns a flag indicating the receipt of a character on serial channel 1. The value set by the last execution of Set_UART_Number specifies the module number. The flag is true true (-1) if there is at least one character in the input FIFO. Otherwise the returned flag is false (0). The demonstration program shows how to use this function to revector serial I/O in a task. See also Ask_Key_UART.

C: void **Ch1_Emit** (uchar character)
4th: **Ch1_Emit** (char --)
This function queues the specified character in the output FIFO for transmission on serial channel 1. The value set by the last execution of Set_UART_Number specifies the module number. If the output FIFO is full, this routine waits and PAUSEs until there is room in the FIF0, then puts the specified character in the FIFO so that it will be transmitted. The demonstration program shows how to use this function to revector serial I/O in a task. See also Emit_UART.

C: uchar **Ch1_Key** (void)
4th: **Ch1_Key** (-- char)
Waits (if necessary) for receipt of a character from channel 1, and returns the received character. The value set by the last execution of Set_UART_Number specifies the module number. PAUSEs while waiting. The returned byte is the next pending character in the FIFO (that is, the oldest unretrieved character in the receive FIFO). The demonstration program shows how to use this function to revector serial I/O in a task. See also Key_UART.

C: int **Ch2_Ask_Key** (void)
4th: **Ch2_Ask_Key** (-- flag)
Returns a flag indicating the receipt of a character on serial channel 2. The value set by the last execution of Set_UART_Number specifies the module number. The flag is true true (-1) if there is at least one character in the input FIFO. Otherwise the returned flag is false (0). The demonstration program shows how to use this function to revector serial I/O in a task. See also Ask_Key_UART.

C: void **Ch2_Emit** (uchar character)
4th: **Ch2_Emit** (char --)
This function queues the specified character in the output FIFO for transmission on serial channel 2. The value set by the last execution of Set_UART_Number specifies the module number. If the output FIFO is full, this routine waits and PAUSEs until there is room in the FIF0, then puts the specified character in the FIFO so that it will be transmitted. The demonstration program shows how to use this function to revector serial I/O in a task. See also Emit_UART.

C: uchar **Ch2_Key** (void)
4th: **Ch2_Key** (-- char)
Waits (if necessary) for receipt of a character from channel 2, and returns the received character. The value set by the last execution of Set_UART_Number specifies the module number. PAUSEs while waiting. The returned byte is the next pending character in the FIFO (that is, the oldest unretrieved character in the receive FIFO). The demonstration program shows how to use this function to revector serial I/O in a task. See also Key_UART.

C: **CHANNEL1**
4th: **CHANNEL1** (-- 1)
A constant equal to 1, used as a channel specifier.

C: **CHANNEL2**
4th: **CHANNEL2** (-- 2)
A constant equal to 2, used as a channel specifier.

C: **DEFAULT_BAUDRATE**
4th: **DEFAULT_BAUDRATE** (-- n)
A constant in the demonstration program whose value sets the baud rate for channels 1 and 2 in the Default_UART_Init function. The user should set this to the desired value. Standard baud rates are 300, 1200, 2400, 4800, 9600, 19200, 38400, and 56000 baud. The suggested value in the demo source code is 19200 baud. See Default_UART_Init and Set_Baud.

C: **DEFAULT_BITS_PER_CHAR**
4th: **DEFAULT_BITS_PER_CHAR** (-- n)
A constant in the demonstration program whose value sets the number of data bits per character for channels 1 and 2 in the Default_UART_Init function. Allowed values are 5, 6, 7, or 8. The user should set this to the desired value. The suggested value in the demo source code is 8. See Default_UART_Init and Set_Data_Format.

C: **DEFAULT_PROTOCOL**
4th: **DEFAULT_PROTOCOL** (-- n)
A constant in the demonstration program whose value sets the protocol for channels 1 and 2 in the Default_UART_Init function. The user should set this to the desired value. Allowed values are given by the constants RS232, RS422, RS485, or NOT_USED; see their glossary entries. The suggested value in the demo source code is RS232. See Default_UART_Init and Set_Protocols.

C: **DEFAULT_MODEM_SUPPORT**
4th: **DEFAULT_MODEM_SUPPORT** (-- n)
A constant in the demonstration program whose value sets the modem support flag for channel 1 in the Default_UART_Init function. Allowed values are TRUE (nonzero) or FALSE (0). The user should set this to the desired value. The suggested value in the demo source code is FALSE. See Default_UART_Init and Set_Protocols.

C: **DEFAULT_PARITY**
4th: **DEFAULT_PARITY** (-- n)

A constant in the demonstration program whose value sets the parity type for channels 1 and 2 in the Default_UART_Init function. The user should set this to the desired value. Allowed values are given by the constants NO_PARITY, EVEN_PARITY, ODD_PARITY, HIGH_PARITY, or LOW_PARITY; see their glossary entries. The suggested value in the demo source code is NO_PARITY. See Default_UART_Init and Set_Data_Format.

C: **DEFAULT_STOP_BITS**
4th: **DEFAULT_STOP_BITS** (-- n)
A constant in the demonstration program whose value sets the number of stop bits for channels 1 and 2 in the Default_UART_Init function. Allowed values are 1 or 2. The user should set this to the desired value. The suggested value in the demo source code is 1. See Default_UART_Init and Set_Data_Format.

C: int **Default_UART_Init** (int module_num)
4th: **Default_UART_Init** (module_num -- error)
A function in the demonstration program that sets the protocols, data formats, and baud rates for both channel 1 and channel 2 according to a set of default parameter constants. This function passes the parameters DEFAULT_BITS_PER_CHAR, DEFAULT_STOP_BITS, and DEFAULT_PARITY to the Set_Data_Format function for each channel in the specified module. It passes the parameter DEFAULT_BAUDRATE to Set_Baud for each channel in the specified module. It passes the parameters DEFAULT_MODEM_SUPPORT and DEFAULT_PROTOCOL to Set_Protocols. If an invalid protocol combination is specified as described below, this function returns a nonzero error flag. Otherwise, a result of zero is returned. This function is provided in source form to illustrate how to initialize the UART Module. The values of the default parameters and the source code of this function can be customized to meet the needs of your application.

C: void **Emit_UART** (uchar character, int channel_num, int module_num)
4th: **Emit_UART** (char\channel_num\module_num --)
Queues the specified character in the output FIFO for transmission on the specified serial channel. If the output FIFO is full, this routine waits and PAUSEs until there is room in the FIF0, then puts the specified character in the FIFO so that it will be transmitted. See also Ch1_Emit and Ch2_Emit.

C: void **End_Break** (int channel_num, int module_num)
4th: **End_Break** (channel_num\module_num --)
Ends the break sequence that was initiated by Send_Break on the specified channel, thereby returning the specified output to the idle high (mark) state at the UART.

C: **EVEN_PARITY**
4th: **EVEN_PARITY** (-- n)
A constant (= 0x18) that, when passed as a parameter to the Set_Data_Format function, configures the channel's data format for even parity, meaning that the sum of the bits is constrained to be even.

C: **HIGH_PARITY**
4th: **HIGH_PARITY** (-- n)
A constant (= 0x28) that, when passed as a parameter to the Set_Data_Format function, configures the channel's data format for high parity, meaning that the parity bit is high at the UART. This is also known as "mark parity".

C: void **Is_DTR** (int desired_state, int module_num)
4th: **Is_DTR** (desired_state\module_num --)
Writes the specified state to the DTR (Data Terminal Ready) output on channel1. If the specified state is true (nonzero), the signal is made active (i.e., high in UART register, low at the UART output pin, and high on the RS232 cable). If the specified state is false (zero), the signal is made inactive (i.e., low in the UART register,

high at the UART output pin, and low on the RS232 cable). This signal is typically used for modem handshaking.

C: void **Is_RTS** (int desired_state, int module_num)
4th: **Is_RTS** (desired_state\module_num --)
Writes the specified state to the RTS (Request To Send) output on channel1. If the specified state is true (nonzero), the signal is made active (i.e., high in UART register, low at the UART output pin, and high on the RS232 cable). If the specified state is false (zero), the signal is made inactive (i.e., low in the UART register, high at the UART output pin, and low on the RS232 cable). This signal is typically used for modem handshaking.

C: uchar **Key_UART** (int channel_num, int module_num)
4th: **Key_UART** (channel_num\module_num -- char)
Waits (if necessary) for receipt of a character from the specified channel and module, and returns the received character. PAUSEs while waiting. The returned byte is the next pending character in the FIFO (that is, the oldest unretrieved character in the receive FIFO). See also Ch1_Key and Ch2_Key.

C: **LOW_PARITY**
4th: **LOW_PARITY** (-- n)
A constant (= 0x38) that, when passed as a parameter to the Set_Data_Format function, configures the channel's data format for low parity, meaning that the parity bit is low at the UART. This is also known as "space parity".

C: **NOT_USED**
4th: **NOT_USED** (-- n)
A constant (= 0) that, when passed as a parameter to the Set_Protocols function, indicates that the specified channel is not in use.

C: **NO_PARITY**
4th: **NO_PARITY** (-- n)
A constant (= 0) that, when passed as a parameter to the Set_Data_Format function, configures the channel's data format for no parity.

C: **ODD_PARITY**
4th: **ODD_PARITY** (-- n)
A constant (= 0x08) that, when passed as a parameter to the Set_Data_Format function, configures the channel's data format for odd parity, meaning that the sum of the bits is constrained to be odd.

C: int **Read_CTS** (int module_num)
4th: **Read_CTS** (module_num -- flag)
Returns the current state of the CTS (Clear To Send) input on channel1. Returns true (-1) if the signal is active (i.e., high in UART register, low at the UART output pin, and high on the RS232 cable). Returns false (zero) if the signal is inactive (i.e., low in the UART register, high at the UART output pin, and low on the RS232 cable). This signal is typically used for modem handshaking.

C: int **Read_DCD** (int module_num)
4th: **Read_DCD** (module_num -- flag)
Returns the current state of the DCD (Data Carrier Detect) input on channel1. Returns a true (-1) flag if the signal is active (i.e., high in UART register, low at the UART output pin, and high on the RS232 cable). Returns false (zero) if the signal is inactive (i.e., low in the UART register, high at the UART output pin, and low on the RS232 cable). This signal is typically used for modem interfacing.

C: int **Read_DSR** (int module_num)
4th: **Read_DSR** (module_num -- flag)

Returns the current state of the DSR (Data Set Ready) input on channel1. Returns true (-1) if the signal is active (i.e., high in UART register, low at the UART output pin, and high on the RS232 cable). Returns false (zero) if the signal is inactive (i.e., low in the UART register, high at the UART output pin, and low on the RS232 cable). This signal is typically used for modem handshaking.

C: int **Read_UART_Number** (void)
4th: **Read_UART_Number** (-- module_num)
Returns the module number that was set by the last execution of Set_UART_Number; see its glossary entry.

C: **RS232**
4th: **RS232** (-- n)
A constant (= 2) that, when passed as a parameter to the Set_Protocols function, configures the specified channel's data format for RS232.

C: **RS422**
4th: **RS422** (-- n)
A constant (= 1) that, when passed as a parameter to the Set_Protocols function, configures the specified channel's data format for RS422.

C: **RS485**
4th: **RS485** (-- n)
A constant (= 0x21) that, when passed as a parameter to the Set_Protocols function, configures the specified channel's data format for RS485.

C: void **RS485_Rcv_UART** (int channel_num, int module_num)
4th: **RS485_Rcv_UART** (channel_num\module_num --)
Puts the specified RS485 channel into receive mode. Turns on the receiver and turns off the transmitter of the RS485 driver chip on the specified channel. The application program is responsible for making sure that all pending outgoing characters have been transmitted before invoking this function, as characters that have not yet been transmitted will be lost when the transmitter is disabled. To avoid this problem, use the higher level function named RS485_Rcv_When_Xmit_Done.

C: void **RS485_Rcv_When_Xmit_Done** (int channel_num, int module_num)
4th: **RS485_Rcv_When_Xmit_Done** (channel_num\module_num --)
Puts the specified RS485 channel into receive mode. Waits and PAUSEs until the transmit FIFO is empty, then turns on the receiver and turns off the transmitter of the RS485 driver chip on the specified channel. This function ensures that pending outgoing characters are fully transmitted before disabling the RS485 transmitter.

C: void **RS485_Xmit_UART** (int channel_num, int module_num)
4th: **RS485_Xmit_UART** (channel_num\module_num --)
Puts the specified RS485 channel into transmit mode. Turns on the transmitter and turns off the receiver of the RS485 driver chip on the specified channel. The application program is responsible for making sure that all expected characters have been received before invoking this function, as incoming characters that have not yet been queued in the receiver FIFO will be lost when the receiver is disabled.

C: void **Run_Demo** (void)
4th: **Run_Demo** (--)
The top level function in the demonstration program. When called, it builds and activates two tasks named CH1_TASK and CH2_TASK. Each task runs the interactive monitor using a channel on the UART Module. The initialization is performed by Default_UART_Init (see its glossary entry). Each task revectors the Emit, Key and Ask_Key (also called ?KEY) primitives so that all task I/O is implemented via the UART Module. The default QED-Forth task stays active using the serial port on the QED Board. Thus, invoking this function allows you to simultaneously run three serial connections on the QED Board or Panel-Touch Controller.

C: void **Send_Break** (int channel_num, int module_num)
4th: **Send_Break** (channel_num\module_num --)
Waits and PAUSEs until the transmitter FIF0 is empty, then transmits a break on the specified serial channel. Break forces the output at the UART to the low (space) state. Use End_Break to end the break sequence.

C: void **Set_Baud** (uint baudrate, int channel, int module)
4th: **Set_Baud** (baudrate\channel\module --)
Sets the baud rate of the specified channel on the specified module. Baud rates up to 56,000 baud are supported. Standard baud rates are 300, 1200, 2400, 4800, 9600, 19200, 38400, and 56000 baud. Both the local and remote UARTs must be configured for the same baud rate.

C: void **Set_Data_Format** (int numbits, int stopBits, int parity, int channel, int module)
4th: **Set_Data_Format** (numbits\stopBits\parity\channel\module --)
Sets the data format of the specified channel on the specified module according to the specified input parameters. Sets the number of data bits per character to equal the numbits parameter, where numbits must equal 5, 6, 7, or 8. Sets the number of stop bits to 1 if stopBits = 1; otherwise sets the number of stop bits to 2. Sets the parity as specified by the parity parameter; allowed values are NO_PARITY, EVEN_PARITY, ODD_PARITY, HIGH_PARITY, or LOW_PARITY (see their glossary entries).
Notes: This function stops any in-progress break transmission.
If numbits = 5 and stopBits=2, the UART hardware implements 1.5 stop bits.

C: int **Set_Protocols** (int ch1_modem, int ch1_protocol, int ch2_protocol, int module)
4th: **Set_Protocols** (ch1_modem\ch1_protocol\ch2_protocol\module -- error)
Sets the protocols for each channel according to the specified parameters. If an invalid protocol combination is specified as described below, this function exits and returns a nonzero error flag. Otherwise, this function initializes the UART Module hardware, configuring channel 1 for the specified ch1_protocol, and channel 2 for the specified ch2_protocol. Allowed protocol parameters are the constants RS232, RS422, RS485, and NOT_USED; see their glossary entries. The ch1_modem parameter is a flag that determines whether the modem handshaking signals are enabled on channel 1. If the ch1_modem flag is true (nonzero), the handshaking is enabled; otherwise, it is disabled. The DTR and RTS handshaking outputs are set to the active state; see Is_DTR and Is_RTS. The UART hardware is configured with all FIFO's active, and interrupts disabled (polling is used to monitor the UART). If RS485 is specified on a channel, that channel is put in receiving mode. If RS232 or RS422 is specified on a channel, that channel's transmitters & receivers are enabled. An invalid protocol combination occurs: (1) if ch1_modem is true AND ch1_protocol is not RS232; or, (2) if ch1_modem is true AND ch2_protocol is RS232. See the "UART Module Hardware: Modem Handshaking Signals" section for more details.

C: void **Set_UART_Number** (int module_num)
4th: **Set_UART_Number** (module_num --)
Saves the specified module number in a variable that is accessed by the channel-specific serial I/O primitives Ch1_Emit, Ch2_Emit, Ch1_Ask_Key, Ch2_Ask_Key, Ch1_Key, and Ch2_Key. See also Read_UART_Number.

C: **UART_MODULE_NUM**
4th: **UART_MODULE_NUM** (-- n)
A constant in the demonstration program whose value equals the module number. This value must correspond to the jumper settings as shown in Table 1-1. This constant is used by the demonstration program functions Ch1_Emit, Ch2_Emit, Ch1_Ask_Key, Ch2_Ask_Key, Ch1_Key, Ch2_Key, and Run_Demo. The default value in the demonstration source code is 4.

C: void **Wait_Until_Xmit_Done** (int channel_num, int module_num)
4th: **Wait_Until_Xmit_Done** (channel_num\module_num --)
Waits and PAUSEs until the transmitter FIF0 on the specified channel is empty.

C Demonstration Program

This section presents the ANSI C version of the demonstration program source code.

```
// ********************************************************************************************
// FILE NAME:   UModDemo.c
// copyright 2002 Mosaic Industries, Inc.  All rights reserved.
// ------------------------------------------------------------------
// DATE:        5/14/2002
// VERSION:     1.1, for QED4 or Panel-Touch Controller with WildCard Carrier Board
// ------------------------------------------------------------------
// This is the demonstration code for the Dual UART Module.
// Please see the UART Module User Guide for more details.
// The UART Module kernel extension file Install.txt
// MUST be loaded into memory before this file can be loaded.
// This is an illustrative demonstration program that
// shows how to initialize the uarts for RS232 operation and run dual
// tasks using the two UART Module serial ports. Each task simply
// echoes incoming characters back to the terminal.
// When the top level function main() is running, the QED Board
// or Panel-Touch Controller is simultaneously using 3 serial ports:
// the standard primary serial port is running the QED interactive monitor,
// and each of the two serial channels on the UART Module is echoing characters.
// Using the constants and/or the Default_UART_Init function
// defined in this file, you may customize the
// baud rate and protocol settings for the UART Module ports.
//
// The QED operating system supports revectorable I/O, meaning that
// in any given task the standard C serial I/O routines such as
// putchar, puts, getchar, gets, printf, and scanf can be made to use
// any specified serial channel.  All that is required is to customize
// three functions named Key, AskKey, and Emit to the specified serial channel
// for the specified task.  This file shows how to do this
// using the functions defined in the UART Module kernel extension.
//
// MAKE SURE THAT THE UART_MODULE_NUM CONSTANT MATCHES YOUR HARDWARE JUMPER SETTINGS!!

// ------------------------------------------------------------------
// Demonstration functions defined in this file:
// UART_MODULE_NUM  // this constant MUST match hardware jumper settings!
// int Default_UART_Init( int module_num ) // demonstrates how to initialize module
// void main(void) // runs the demo program

// ------------------------------------------------------------------
// Notes:
// Disclaimer: THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT
//             ANY WARRANTIES OR REPRESENTATIONS EXPRESS OR IMPLIED,
//             INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES
//             OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
//
// ********************************************************************

// This version is for QED4 Boards or Panel-Touch Controllers with WildCard Carrier Board.

#include <\mosaic\allqed.h>     // include all of the qed and C utilities
#include "library.c"    // this is a kernel extension file;
// assume it's in the same directory as this file; if not, edit the file specification



// ****************** DEMONSTRATION PROGRAM ***************************
```

```
// The default task runs an interactive monitor as usual, using the 68HC11 UART.
// We create a second task and a third task that echo all incoming chars,
// each communicating using a serial channel on the UART Module.
// To run this demonstration, simply execute:
//       main
// You'll be running independent serial-echo tasks
// from your second and third terminals connected to the UART module.

// NOTE: YOU MUST MAKE SURE THAT UART_MODULE_NUM CONSTANT CORRESPONDS TO YOUR HARDWARE!!
#define UART_MODULE_NUM 4   // double check your hardware jumper settings!!!

// default values used in Default_UART_Init (edit these to suit your requirements):
#define DEFAULT_BITS_PER_CHAR   8
#define DEFAULT_STOP_BITS       1
#define DEFAULT_PARITY          NO_PARITY
#define DEFAULT_BAUDRATE        19200
#define DEFAULT_PROTOCOL        RS232
#define DEFAULT_MODEM_SUPPORT   FALSE

// Define and allocate RAM for the task areas:
TASK     ch1_task;  // 1 Kbyte per task area
TASK     ch2_task;  // 1 Kbyte per task area


int Default_UART_Init( int module_num )
    // initializes BOTH channel1 and channel2 on the specified uart module_num.
    // result = SUCCESS (=0) or BAD_PROTOCOL_COMBO (=1)
    // this routine demonstrates how to initialize the uarts using default settings;
    // the user should customize the parameters to suit the application.
{       // configure channel1:
    Set_Data_Format(DEFAULT_BITS_PER_CHAR,DEFAULT_STOP_BITS,DEFAULT_PARITY,1,module_num);
    Set_Baud(DEFAULT_BAUDRATE,1,module_num);
        // configure channel2:
    Set_Data_Format(DEFAULT_BITS_PER_CHAR,DEFAULT_STOP_BITS,DEFAULT_PARITY,2,module_num);
    Set_Baud(DEFAULT_BAUDRATE,2,module_num);
        // set protocols for each channel:
    return Set_Protocols(DEFAULT_MODEM_SUPPORT,DEFAULT_PROTOCOL,DEFAULT_PROTOCOL,module_num);
}


void CH1_Monitor(void)
    // infinite task loop for ch1_task, simply echoes all incoming chars on channel1
{   uchar this_char;
    UKEY = (xaddr) CH1_KEY_XADDR;   // defined in library.h
    UASK_KEY = (xaddr) CH1_ASK_KEY_XADDR;        // defined in library.h
    UEMIT = (xaddr) CH1_EMIT_XADDR;     // defined in library.h
    printf("Ready to echo incoming characters on Channel1...\n");
     while(1)
    {   this_char = _readTerminal();
         if( this_char == '\r')
             this_char = '\n';  // substitute linefeed for cr, ansi-c style
        putchar(this_char);     // automatically adds cr in front of linefeed
    }
}

void CH2_Monitor(void)
    // infinite task loop for ch2_task, simply echoes all incoming chars on channel2
{   uchar this_char = 0;
    UKEY = (xaddr) CH2_KEY_XADDR;       // defined in library.h
```

```
    UASK_KEY = (xaddr) CH2_ASK_KEY_XADDR;       // defined in library.h

    UEMIT = (xaddr) CH2_EMIT_XADDR;     // defined in library.h
    printf("Ready to echo incoming characters on Channel2...\n");
    while(1)
    {   this_char = _readTerminal();
         if( this_char == '\r')
             this_char = '\n';  // substitute linefeed for cr, ansi-c style
        putchar(this_char);     // automatically adds cr in front of linefeed
    }
}

_Q void Run_Demo(void)
    // builds and activates two forth-monitor tasks,
    // each using a separate channel on the uart module.
{   Set_UART_Number(UART_MODULE_NUM);
    if(Default_UART_Init(UART_MODULE_NUM))  // initialize the hardware
        printf("\nError: Invalid protocol combination was specified!\n");
    else
    {   printf("\nStarting UART Module Demo...\n");
        SERIAL_ACCESS = RELEASE_ALWAYS;     // ensure lots of PAUSEs in Forth task
        NEXT_TASK = TASKBASE;  // required! empty the round-robin task loop
        BUILD_C_TASK(0,0,&ch1_task);       // no heap needed
        BUILD_C_TASK(0,0,&ch2_task);       // no heap needed
        ACTIVATE(CH1_Monitor,&ch1_task);
        ACTIVATE(CH2_Monitor,&ch2_task);
        StartTimeslicer();  // enable task switching
    }
}

void main(void)
{   Run_Demo();
}
```

Forth Demonstration Program

This section presents the ANSI C version of the demonstration program source code.

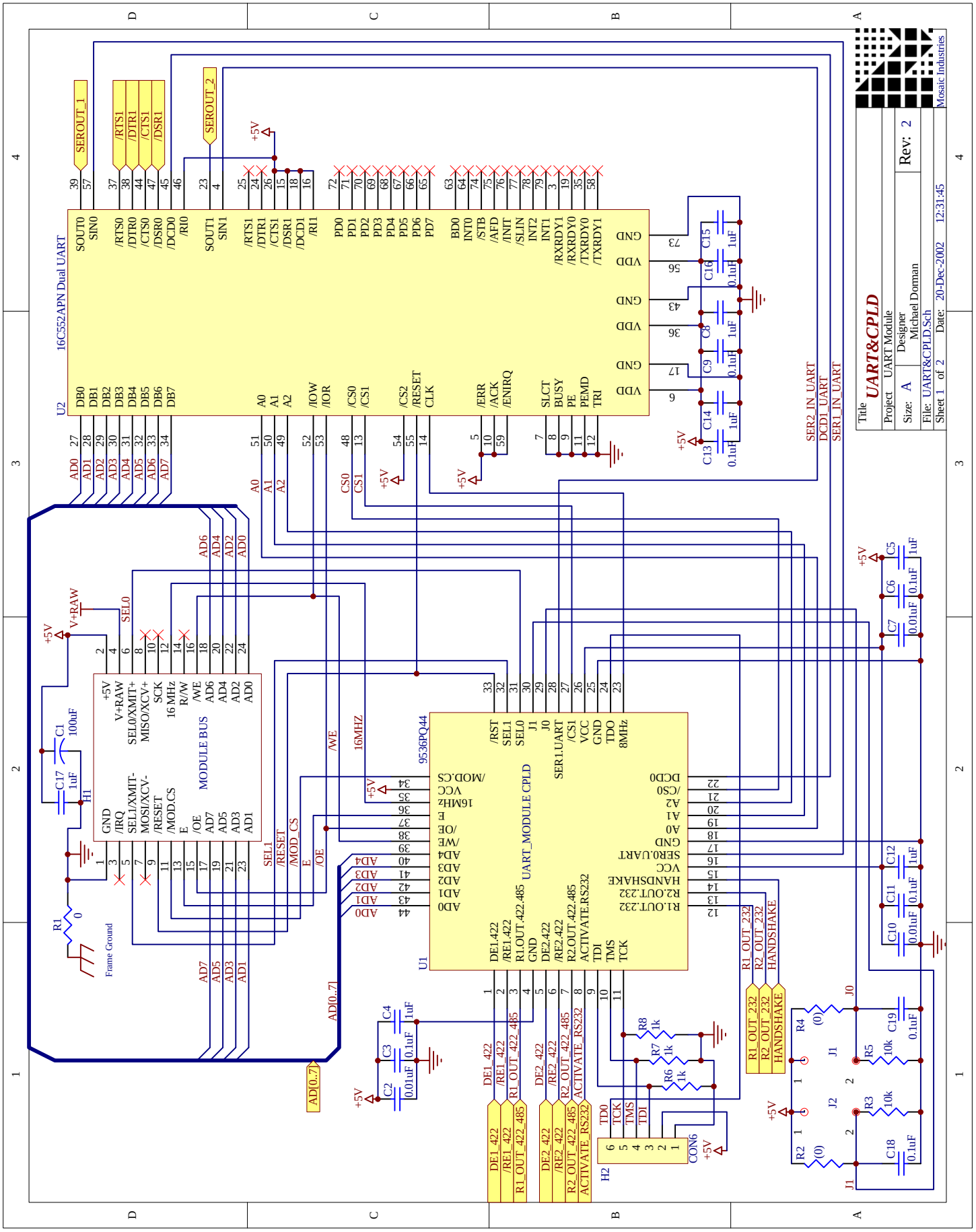
```
\ ********************************************************************************************
\ FILE NAME:   UModDemo.4TH
\ copyright 2002 Mosaic Industries, Inc.  All rights reserved.
\ ----------------------------------------------------------------------
\ DATE:        5/14/2002
\ VERSION:     1.1, for QED4 or Panel-Touch Controller with WildCard Carrier Board
\ ----------------------------------------------------------------------
\ This is the demonstration code for the Dual UART Module.
\ Please see the UART Module User Guide for more details.
\ The accompanying file named UModDvr.4th (or the corresponding kernel extension)
\ MUST be loaded before this file can be loaded.
\ This is an illustrative demonstration program that
\ shows how to initialize the uarts for RS232 operation and run dual
\ QED monitor tasks using the two UART Module serial ports.

\ When the top level function Run_Demo is running, the QED Board
\ or Panel-Touch Controller is simultaneously using 3 serial ports:
\ the standard primary serial port and each of the two serial channels
```

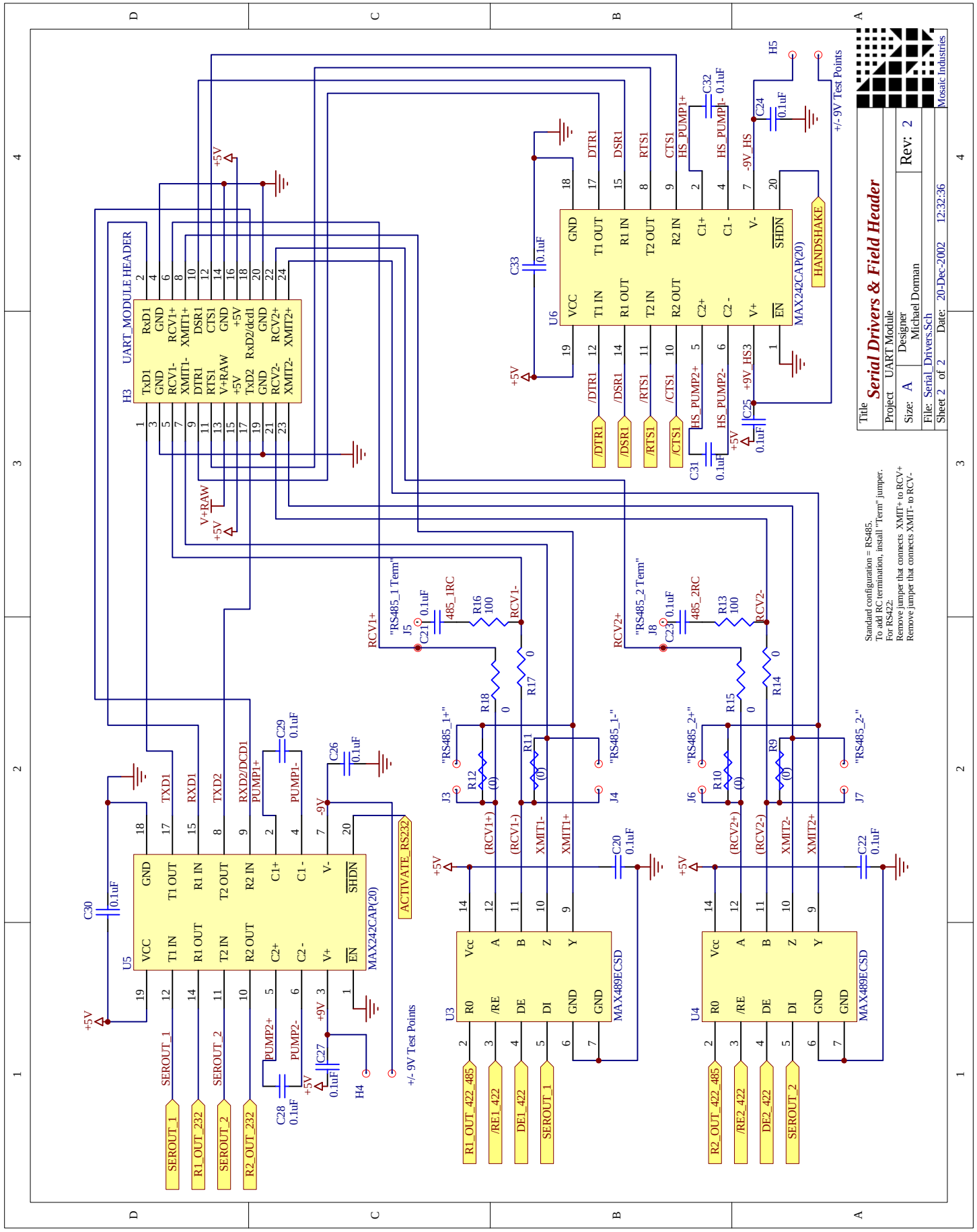
```
\ on the UART Module is running an instance of the QED-Forth monitor.
\ Using the constants and/or the Default_UART_Init function
\ defined in this file, you may customize the
\ baud rate and protocol settings for the UART Module ports.
\
\ The QED operating system supports revectorable I/O, meaning that
\ in any given task the standard serial I/O routines such as
\ CR and ." can be made to use any specified serial channel.
\ All that is required is to customize and revector (store the xcfa of)
\ three functions named Key, ?Key, and Emit to the specified serial channel
\ for the specified task.  This file shows how to do this
\ using the functions defined in the UART Module kernel extension.

\ MAKE SURE THAT THE UART_MODULE_NUM CONSTANT MATCHES YOUR HARDWARE JUMPER SETTINGS!!

\ ---------------------------------------------------------------------
\
\ Demonstration functions defined in this file:
\ UART_MODULE_NUM   \ MUST match hardware jumper settings!
\ Default_UART_Init   ( module_num -- result ) \ demonstrates how to initialize module
\ Run_Demo    ( -- )

\ ---------------------------------------------------------------------
\ Notes:
\
\ Disclaimer: THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT
\             ANY WARRANTIES OR REPRESENTATIONS EXPRESS OR IMPLIED,
\             INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES
\             OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
\
\ *********************************************************************

\ This version is for QED4 Boards, Panel-Touch Controllers, with WildCard Carrier Board.

HEX
WHICH.MAP O=
IFTRUE 4 PAGE.TO.RAM  \ if in standard.map...
       5 PAGE.TO.RAM
       6 PAGE.TO.RAM
      DOWNLOAD.MAP
ENDIFTRUE

\ set memory map on page 4 after load of UMod_Dvr.4th:
\ 800 4 DP X!  5800 4 NP X!  \ comment this out or change it if memory map is already set up

F WIDTH !      \ set width of names stored in dictionary

ANEW UDemo_Code   \ define forget marker for easy re-loading


\ ****************** DEMONSTRATION PROGRAM ***************************

\ The default task runs FORTH as usual, using the 68HC11 UART.
\ We create a second task and a third task that also run FORTH,
\ each communicating using a serial channel on the UART Module.
\ To run this demonstration, simply execute:
\       RUN_DEMO
\ You'll be running FORTH from your standard terminal
\ and you'll be running independent
\ FORTH tasks from your second and third terminals connected to the UART module.
```

```
DECIMAL      \ compile this section in decimal base

\ NOTE: YOU MUST MAKE SURE THAT UART_MODULE_NUM CONSTANT CORRESPONDS TO YOUR HARDWARE!!
4 CONSTANT UART_MODULE_NUM   \ double check your hardware jumper settings!!!

\ default values used in Default_UART_Init (edit these to suit your requirements):
8         CONSTANT DEFAULT_BITS_PER_CHAR
1         CONSTANT DEFAULT_STOP_BITS
NO_PARITY   CONSTANT DEFAULT_PARITY
19200      CONSTANT DEFAULT_BAUDRATE
RS232      CONSTANT DEFAULT_PROTOCOL
FALSE      CONSTANT DEFAULT_MODEM_SUPPORT

: Default_UART_Init   ( module_num -- result )
   \ initializes BOTH channel1 and channel2 on the specified uart module_num.
   \ result = SUCCESS (=0) or BAD_PROTOCOL_COMBO (=1)
   \ this routine demonstrates how to initialize the uarts using default settings;
   \ the user should customize the parameters to suit the application.
   \ CAUTION: if decimal baud rates are hard-coded into this routine, make sure that
   \ this routine is compiled in decimal base.
   LOCALS{ &module }
   \ configure channel1:
   DEFAULT_BITS_PER_CHAR DEFAULT_STOP_BITS DEFAULT_PARITY CHANNEL1
   &module      ( numbits\numStopBits\parity_code\channel_num\module_num -- )
   Set_Data_Format                    ( -- )
   DEFAULT_BAUDRATE CHANNEL1 &module       ( baud\channel_num\module_num -- )
   Set_Baud                         ( -- )
   \ configure channel2:
   DEFAULT_BITS_PER_CHAR DEFAULT_STOP_BITS DEFAULT_PARITY CHANNEL2
   &module      ( numbits\numStopBits\parity_code\channel_num\module_num -- )
   Set_Data_Format                    ( -- )
   DEFAULT_BAUDRATE CHANNEL2 &module       ( baud\channel_num\module_num -- )
   Set_Baud                         ( -- )
   \ set protocols for each channel:
   DEFAULT_MODEM_SUPPORT DEFAULT_PROTOCOL DEFAULT_PROTOCOL
   &module         ( Ch1_modem_support\Ch1_protocol\Ch2_protocol\module_num -- )
   Set_Protocols                     ( -- result )
;

HEX      \ variable area MUST be in common memory! ie., USE.PAGE, or HEX 8E00 0 VP X!
400 V.INSTANCE:     CH1_TASK    \ 1 Kbyte per task area
400 V.INSTANCE:     CH2_TASK     \ 1 Kbyte per task area

: CH1_Monitor   ( -- )         \ infinite task loop for CH1_TASK
     CFA.FOR CH1_EMIT UEMIT X!   \ revector this task's serial routines to use channel1
     CFA.FOR CH1_Ask_KEY  U?KEY X!
     CFA.FOR CH1_KEY   UKEY  X!
     CR ." Starting CH1_Monitor..."
     QUIT                          \ call the infinite-loop FORTH monitor
;

: CH2_Monitor   ( -- )         \ infinite task loop for CH2_TASK
     CFA.FOR CH2_EMIT UEMIT X!   \ revector this task's serial routines to use channel2
     CFA.FOR CH2_Ask_KEY  U?KEY X!
     CFA.FOR CH2_KEY   UKEY  X!
     CR ." Starting CH2_Monitor..."
     QUIT                          \ call the infinite-loop FORTH monitor
;
```

```
: Run_Demo    ( -- )
   \ builds and activates two forth-monitor tasks,

   \ each using a separate channel on the uart module.
   UART_MODULE_NUM Set_UART_Number   \ set global variable, must match hardware
   UART_MODULE_NUM Default_UART_Init   ( -- result )   \ initialize the hardware
   IF   CR ." Error: Invalid protocol combination was specified!" CR
   ELSE
      RELEASE.ALWAYS SERIAL.ACCESS !      \ ensure lots of PAUSEs in Forth task
      (STATUS) NEXT.TASK !   \ empty the task loop
      0\0 0\0 0\0 CH1_TASK BUILD.STANDARD.TASK
      0\0 0\0 0\0 CH2_TASK BUILD.STANDARD.TASK
      CFA.FOR CH1_Monitor CH1_TASK ACTIVATE
      CFA.FOR CH2_Monitor CH2_TASK ACTIVATE
      START.TIMESLICER
   ENDIF
;

4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
6 PAGE.TO.FLASH
STANDARD.MAP
SAVE
```

Hardware Schematics


16C552APN Dual UART
U2
MODULE BUS
H1
9536PQ44
UART_MODULE CPLD
U1
Frame Ground
SEROUT_1
SEROUT_2
SER2_IN_UART
DCD1_UART
SER1_IN_UART
Title
UART&CPLD
Project UART Module
Size: A
Designer Michael Dorman
Rev: 2
File: UART&CPLD.Sch
Sheet 1 of 2
Date: 20-Dec-2002 12:31:45
Mosaic Industries


Serial Drivers & Field Header
UART Module
Michael Domman
Rev: 2
Mosaic Industries
Standard configuration = RS485.
To add RC termination, install "Term" jumper.
For RS422:
Remove jumper that connects XMIT+ to RCV+
Remove jumper that connects XMIT- to RCV-
UART_MODULE HEADER
MAX242CAP(20)
MAX489ECSD
HANDSHAKE
+/- 9V Test Points
ACTIVATE_RS232
"RS485_1 Term"
"RS485_2 Term"
485_1RC
485_2RC
HS_PUMP1+
HS_PUMP2+
HS_PUMP2-
-9V_HS
+9V_HS3
SEROUT_1
SEROUT_2
R1_OUT_232
R2_OUT_232
R1_OUT_422_485
R2_OUT_422_485
/RE1_422
/RE2_422
DE1_422
DE2_422
/DTR1
/DSR1
/RTS1
/CTS1