Chapter 1

OVERVIEW

1.1 FEATURES

The LiteComm-TP Toolbox(Tm) is a set of powerful routines designed to provide easy access to the full capabilities of the PC's asynchronous communications ports. The LiteComm-TP ToolBox supports fully interrupt-driven and buffered communications support on COM1 through COM4 simultaneously. Now you can quickly incorporate sophisticated communications support in your applications without having in-depth knowledge of how the hardware functions.

LiteComm-TP is implemented as a set of 4 units for the basic product, with additional units providing the protocol-engine capability. The protocol engines are a part of the registered version of the package.

The LiteComm ToolBox was originally developed in the C language for use in CAD/CAM applications that required the ability to have PC compatible systems communicating with a variety of devices. LiteComm-TP extends the same capabilities to the PASCAL programmer.

1.2 THE SHAREWARE CONCEPT

Shareware is a "try before you buy" means of software distribution. If you find a shareware product useful, pay the registration fee, and let the authors know that you support their efforts.

Information Technology is a member of the Association of Shareware Professionals (ASP). ASP wants to make sure that the shareware principle works for you. If you are unable to resolve a shareware-related problem with an ASP member by contacting the member directly, ASP may be able to help. The ASP Ombudsman can help you resolve a dispute or problem with an ASP member, but does not provide technical support for members' products. Please write to the ASP Ombudsman at P.O. Box 5786, Bellevue, WA 98006 or send a Compuserve message via easyplex to ASP Ombudsman 70007,3536.

LITECOMM-TP (Tm) TOOLBOX for Turbo-PASCAL

Chapter 2

LICENSE, WARRANTY AND REGISTRATION

2.1 LICENSE

The LiteComm-TP ToolBox, is distributed as a shareware product. We urge you to register your copy today. See the registration form at the end of this manual.

Information Technology, Ltd, grants to registered users a nonexclusive, perpetual license to the LiteComm-TP ToolBox, subject to these terms and conditions:

1. You must treat your copy of the LiteComm-TP Toolbox as you would a book. You may install the LiteComm-TP ToolBox on more than one machine, but you may use only one copy at a time. If you desire, site licenses are available at a reduced cost. You may make as many copies of the LiteComm-TP ToolBox as you require for the sole purpose of backup.

2. You may incorporate portions of the LiteComm-TP ToolBox into products that you develop without the payment of additional royalties or license fees. You must include the statement 'Portions Copyright 1987, 1988, Information Technology, Ltd' in your product's documentation.

3. You may copy and redistribute the shareware portion of the LiteComm-TP ToolBox, commonly known as LCOMMTP.ARC, but you may not modify in any way, the contents of the shareware package.

4. Information Technology grants to ASP-approved vendors only the right to charge a duplication fee, not to exceed $8.00 for providing a copy of the shareware version of the product. No other individual or vendor is permitted to charge a fee for providing such a copy without the express, written consent of Information Technology, Ltd,

5. You may not redistribute, in any form, the source code for the LiteComm-TP ToolBox. Further, you may not translate the source code for the LiteComm-TP ToolBox into any other language without the express, written consent of Information Technology, Ltd.

6. Information Technology reserves the right to change both the LiteComm-TP ToolBox or its documentation without prior notice, with no obligation to you, the licensee.

7. You agree that any disputes arising from this license will be subject to the laws of the state of Rhode Island.

8. You agree to hold the developer and distributors of the LiteComm-TP ToolBox harmless for any damages, either direct or consequential, that might arise from the use of this product.

9. You acknowledge that the LiteComm-TP ToolBox, libraries, source code, and documentation are the copyrighted property of Information Technology, Ltd.

10. By your use of the LiteComm-TP ToolBox, you acknowledge that you have read, and understand the terms and conditions of this license.

2.2 WARRANTY

The LiteComm-TP ToolBox is distributed as-is and without warranty, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Information Technology, Ltd does warrant the distribution media for a period of 30 days. During that period, Information Technology, Ltd will replace the distribution media or provide a refund at its option.

2.3 REGISTERING YOUR COPY

Registration of your copy of the LiteComm-TP ToolBox provides you with several benefits:

1. Puts you on our mailing list for low-cost updates, enhancements, and alert bulletins when they occur.

2. Gives you access to telephone support. Sorry, but we cannot provide support by telephone to unregistered user's of the ToolBox. Unregistered users can leave EMAIL on Compuserve to 70166,1152 and on GEnie to I.TECH. We will respond to EMAIL on an as-available basis.

3. Helps to further the shareware concept.

You can order directly from us or from the Public (Software) Library 1-800-2424-PSL (for orders only. For information call 1-713-665-7017) or by writing PSL; P.O.Box 35705; Houston, TX 77235-5705. MC/Visa Accepted.

If you want to order directly from us, print the file REG.FRM and follow the instructions there.

2.4 NOTE

LiteComm-TP is a package undergoing continuing development. Registered users of the product receive, in addition to the fully-functional base package, units that provide protocol engines supporting XModem, XModem-1K, and YModem protocols.

We plan to follow these with similar engines for CompuServe B, Telink, and other protocols. These engines, as they are released, will only be made available to registered ToolBox users. The shareware version, as enhanced but without the protocol engines, will continue to be offered.

LITECOMM-TP (Tm) TOOLBOX for Turbo-PASCAL

Chapter 3

Serial Port Fundamentals

3.1 The 8250 UART family

This portion of the manual provides you with some details about the working of the 8250 (and related) UART'S, the basic component of your system's serial port. Some close compatibles use enhanced versions of this chip, such as the Intel 16450. LiteComm is known to work successfully with such devices. If you have questions about the kind of serial port you are using, refer to the manufacturer's documentation. If your serial port does not use an 8250 or similar chip, LiteComm will not function.

3.2 Purpose of the port

The purpose of the serial port is to convert information from the form in which it is used within your system, to a form that can be easily used outside your system. Modern computers, by design, are parallel in nature. By this we mean that information travels through the computer's circuitry as whole units or as multiples of whole units. In the IBM PC and related systems, information travels as bytes (8 bits at a time), as words (16 bits at a time), or, on 80386 based systems, as double words (32 bits at a time).

Within the computer, such arrangements are convenient and fast. But when the computer must transfer information to an external device, the problem of data path width is introduced. To provide a true, parallel path between the computer and external devices, there would have to be, at a minimum, enough data lines or circuits between the two to satisfy the data path. For most modern computer systems, this would mean a minimum of 8 data lines, not counting any additional control information that might also be required. For certain newer systems, the requirement might be for as many as 32 data lines. In effect, it might be necessary to have several different versions of a device, dependent upon the data path width of the computer to which it is connected.

The purpose of a serial port is to convert the information from its internal, parallel form, to a more common, external form and back again. By using such an approach, we simplify the interconnection of devices, reducing information to its lowest common denominator, the bit. And it allows us to transfer information 1 bit at a time,

using a single data path, between devices. The real beauty of this approach is that, by 'agreeing' on how this external form appears, each device can hide the details of how it works, and still accomplish the required task.

3.3 Internal Details

In this section we discuss the fundamental working of serial ports as they are implemented of the IBM PC and close compatible systems. It is not essential that you understand this material thoroughly to be successful with LiteComm. However, it may help to answer some of the more important questions that may arise as you proceed with your development.

3.3.1 The Interrupt Connection

The PC is an interrupt driven system. This is a sophisticated way of saying that the PC can 'pay attention' to a number of internal devices without the necessity of having to check on them periodically.

When we describe interrupts to clients, we use the school room as a metaphor. Think of a teacher lecturing to a group of students during a class. The teacher knows that, on occasion, one or more students may not understand the material that is being presented. So the teacher has the choice of either asking each student periodically if they understand or permitting the students to raise a hand to ask a question. As you can imagine, stopping the class to 'poll' each student will waste valuable time, particularly if no one wants to ask a question. Not only is it wasteful of time, but the last student polled may have forgotten the question he or she wanted to ask by the time the teacher gets to him. If the teacher permits students to 'interrupt' his lecture by raising a hand, much less time is wasted, but the teacher has to be careful to identify each raised hand by name and answer the question quickly and accurately, lest some student forget his question or looses interest altogether.

The internal working of most modern computers is identical to the teacher that permits hand raising. The computer focuses on the task at hand, stopping only to identify and pay attention to devices when they signal that they require this attention. So much for the hardware end of things. The PC has this same capability. But at least some of the work that has to be done requires software in a general purpose computer, otherwise the computer wouldn't be general purpose.

The serial port on the PC is no less capable of asking for attention from the computer, at least from the standpoint of hardware. But, for whatever the reason, MS-DOS and PC-DOS do not provide the needed software to exploit the full capabilities of the serial port. OS/2 does provide such support, we are told, but, at

least for the foreseeable future, its an MS-DOS world. On PC's and true compatibles, the only support for the serial port that is provided is through the system's ROM BIOS. And that support is only for the polled mode of operation.

The method by which 80x86 family systems, of which the PC is a part, is elegant in its simplicity. When a device needs the attention of the system, its asserts a control signal and identifies itself with a number ranging from 0 to 255. On the basic PC, the numbers used actually range from 8 through 15 decimal. The identification is translated to a memory location by multiplying the identification by 4, and the system simulates a special form of a call to the routine whose address is stored at that location. Since it is impossible to predict when a device will require attention, the full address of the routine is stored, both segment and offset, hence the 4.

Once the routine, called the Interrupt Service Routine or ISR is invoked, it has a duty to save the state of the system when the interrupt occurred, take care of the interrupt as quickly as possible, and return control to the interrupted process. But it must also be aware that, while it is doing its work, other, more important devices may require attention, too.

One such device that is likely to require attention is the system clock which ticks roughly 18 times per second. In part, the PC makes provision for this by prioritizing the interrupt scheme. The ISR must allow for this by re- enabling the interrupt control system as rapidly as it is practical to do so. The PC's interrupt structure, if left undisturbed, will prevent interrupts of the same or lower priority from occurring. To help your organize your thoughts, the standard identification for the first two serial ports on the system are 12 (0C) and 11 (0B) for ports 1 and 2 respectively.

As you can see, dealing with the PC's interrupt structure is not for the faint of heart. It requires a significant amount of knowledge, and close attention to detail. With LiteComm, these details have already been taken care of for you. You are free to focus on your application, treating the serial port in much the same way that you would any DOS file.

3.3.2 The Programmable Port Registers

The 8250 port, and its close relatives, are fully programmable. You are fortunate that LiteComm has already taken care of the intricacies of this programming. But some additional information about each of the registers used in the serial port may be of use when you are attempting to communicate with an external device. For the sake of this discussion, we use the basic register numbers, although the register number that is employed in programming is actually the register number referenced below used as an offset to a base port number. In the case of COM1 (port 1), this base is 3F8(hexadecimal).

3.3.2.1 register 0 - transmit/receive

In normal operation, individual characters are read from register 0 when the become available, and are written to register 0 when the transmitter portion of the 8250 is ready to accept a character.

3.3.2.2 register 0 - baud rate selection

During initialization, register 0 is used as part of the mechanism that sets baud rate. During this process, register 0 and its companion register 1 are used to specify the baud rate divisor (not the actual baud rate). The baud rate divisor is a value which, when divided into a given, preset constant, yields the desired baud rate. To use registers 0 and 1 to set the baud rate, access to this mode must be first enabled by writing a value of 80H to register 3, the line control register. Once access is enabled, the least significant byte (LSB) of the divisor is written to register 0; the most significant byte is written to register 1. Access to the normal modes of registers 0 and 1 are re- enabled by writing any value less than 80H to the line control register. Of course, only certain values less than 80H would be meaningful (see the line control register description below).

3.3.2.3 register 1 - interrupt enable

Values written to register 1 control which conditions will cause the 8250 to interrupt the system. There are four possible conditions that can cause interrupts:

1. A character has been received (RDI)

2. The transmitter is ready to send a character (TDI)

3. An error or BREAK signal has been detected (ERI)

4. A modem status signal has changed (MSI)

The designations, in parentheses, are for our purposes only. They are not 'standard' designations. To enable a particular type of interrupt, you must set the corresponding bit in a byte to a 1, then write the byte to register 1. To reset (ignore) the condition, set the corresponding bit to 0. The diagram that follows shows the bit positions the correspond to the various conditions described above.

7	6	5	4	3	2	1	0
N/A	N/A	N/A	N/A	Modem Status	Error/ Break	Xmit Ready	Recv Char

Figure 3.1: Register 1 Bit Definitions

3.3.2.4 register 1 - baud rate selection

See the description under register 0, baud rate selection.

3.3.2.5 register 2 - interrupt identification

Register 2, in normal operation, acts as a companion to register 1. Register 1 determines the conditions that can cause an interrupt. Register 2 is used to determine which condition actually caused the interrupt, when more than one condition has been enabled. Only least significant 3 bits of the register are actually employed. See the diagram of register 2 below.

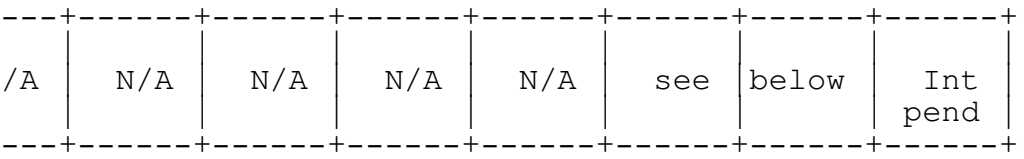

Figure 3.2: Register 2 Bit Definitions

Since it is possible, even likely, that more than one condition may occur at the same time, bit 0 is used to determine whether all conditions that currently exist have been handled. When bit 0 has a value of 0 (yes zero), there are conditions waiting to be handled. When bit 0 has a value of 1, all outstanding conditions have been handled. Bits 2 and 1 taken together identify the actual cause of the interrupt.

Again, because of the multiple conditions which may occur, the 8250 presents the conditions in a prioritized order. When bits 2 and 1 have a value of 3 (the most important), an ERI condition has been identified. The actual error is determined by reading the line status register(register 5). Reading this register also clears the condition.

When a value of 2 is present, an RDI condition has occurred, and a character should be read. from port 0. If the character is not read quickly enough, a data overrun error may occur, indicating that a character was lost.

When bits 2 and 1 have a value of 1, a TDI condition has occurred and a character may be written to register 0.

A value of zero in bits 2 and 1(least important) indicates that one or more of the modem status lines (so called) have changed. The condition is cleared by reading the contents of the modem status register, register 6.

3.3.2.6 register 3 - line control

The line control register provides the means for setting those values that affect the way in which the serial port appears to the outside world. It is through this register that character length,

parity, and other significant values are established. Indirectly, register 3 also plays a role in setting the speed (baud rate) of the port. (See the description of registers 0 and 1 above.)

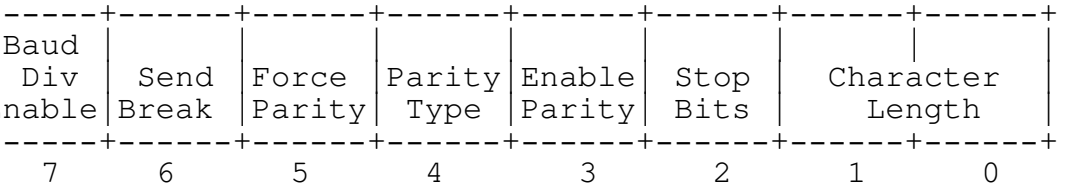

Figure 3.3: Register 3 Bit Definitions

3.3.2.7 register 4 - modem control

The modem control register permits control of the two modem-related signals that the serial port generates as an output. The signals are RTS and DTR.

These two signals are called 'handshaking' signals, since they, in part, help a connected device determine the state of the connection. You should be aware that although these signals were originally designated to be used in a specific fashion, manufacturers of specific devices have used them to meet their own needs. Your success or failure in dealing with any specific device may depend, in part, on your understanding of how the device's manufacturer uses these signals. LiteComm provides you the means for manipulating these signals in a variety of ways.

You will notice in the register 4 diagram, below that some additional positions are identified.

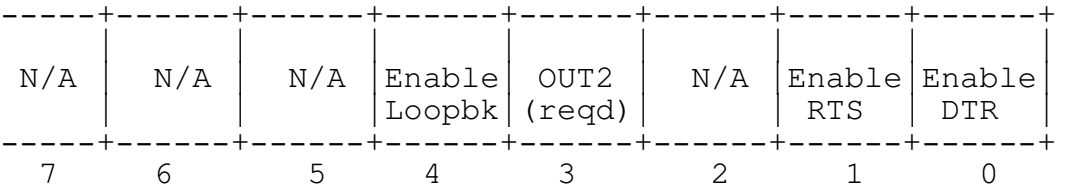

Figure 3.4: Register 4 Bit Definitions

LiteComm controls all of these additional positions for your benefit. Only one deserves mention, the position labeled OUT2. It is necessary for this position to have a value of 1 for the serial port to function as an interrupting device. Since LiteComm relies on interrupts to perform it's job, it insures that this position is always set correctly.

3.3.2.8 register 5 - line status

The line status register is read normally when an ERI condition occurs. Each bit of the character returned when the port is read has significance, as shown in the accompanying diagram. Using the appropriate functions in LiteComm, you can interrogate the value in this register, and test for the various conditions using the LiteComm- provided definitions. Note that, due to the special

nature of the BREAK signal, LiteComm treats this one condition as a separate entity.

Time Out	Shift Reg Empty	Hold Reg Empty	Recd Break	Frame Error	Parity Error	Data OvrRun	Char Recd
7	6	5	4	3	2	1	0

Figure 3.5: Register 5 Bit Definitions

3.3.2.9 register 6 - modem status

Just as the serial port can generate certain 'handshaking' signals, it can also read, and report on the status of similar signals that are generated by an external device. In their original form, these signals had special significance when a terminal was connected to a modem. We refer you to our comments, above, about present day use of the handshaking signals.

One special note is appropriate here. The modem status register actually provides two types of information. The most significant 4 bits (see the diagram) show the current state of the 4 covered signals. The least significant 4 bits indicate which, if any, of the signals have changed state (from zero to one, or vice-versa), since the last time the register was interrogated. LiteComm updates its internal tables with this value in real-time, and reports the results when asked to do so. You can test the signals individually or in combination using the LiteComm-provided definitions.

DCD carr detect	RI ring indic	DSR data set rd	CTS clr to send	DELTA DCD	DELTA RI	DELTA DSR	DELTA CTS
7	6	5	4	3	2	1	0

3.4 The LiteComm Connection

Figure 3.6: Register 6 Bit Definitions

In the design of LiteComm, we have purposely 'hidden' many of the underlying details we presented above. In many cases, you will have little use for this additional information. This is particularly true of most of the applications with which come into contact. In fact, in the majority of applications, you will probably open the port or ports, set the necessary parameters and modem control signals, and do nothing more than read and write characters using one or more of the LiteComm functions. The beauty of LiteComm's design is that its high degree of granularity doesn't force you to pay the price of dragging along functions that you are not using.

The information that we presented above will help you when it is necessary to communicate with a device that requires special handshaking considerations, such as a cash drawer. You may also need some of the information we presented if you intend to use serial ports beyond COM2 (serial port 2).

Finally, by presenting the information that we have supplied, we hope to gain a more informed user. Communications programming is not the black art that some would have you believe, although it can easily seem that way at times. Of all of the calls we receive who need questions answered, more than 75 per cent could have been answered by the caller himself with a more thorough understanding of the underlying concepts and rules.

3.5 TOOLBOX NOTES AND WARNINGS

Before you can send or receive information on a serial port using the ToolBox, you must use the open function to enable the line. This function initializes the 8250 UART with the correct parameters, and introduces the UART into the interrupt structure of the PC. The ToolBox will detect, and report, any errors that you may make in selecting the port or specifying the initial parameters. The ToolBox cannot and will not detect an attempt to open a nonexistent serial port.

The ToolBox interfaces directly with the interrupt structure of the PC. It is critical before exiting a program that has opened a serial port that the serial port is closed with the close function. Because a program may terminate abnormally, the open function installs an exit routine that will automatically close any open ports. Good programming practice demands, however, the your program should close the ports explicitly. By so doing, you may avoid problems in the future if we find it necessary to remove the auto-close functionality. Further, the auto-close functionality drops all modem handshaking signals absolutely, while an explicit close can decide whether or not to drop these signals. You should review Borland's documentation about installing exit routines in the Turbo PASCAL Reference, and you should review the documentation for CommClose for further information about the features of that procedure.

Failure of the open function can be the result of either improper parameters to the open function, or insufficient memory available to allocate the requested buffers and related control structures for the port. Memory for the transmit and receive buffers as well as the port control block are allocated from the heap. It is your responsibility to insure that adequate memory is available for this purpose.

The 8250 serial chip and its descendants will not transmit information until, at a minimum, the DTR (Data Terminal Ready) signal is asserted. The ToolBox will, at your option, assert both

the DTR and RTS signals when you open the port. If you do not select this option you must use the SetModem function to assert (raise) this signal. In addition, some modems and other devices may require you to assert the RTS (Request To Send) signal before they will respond to data. The use of this, and other handshaking signals is HIGHLY hardware-dependant. The ToolBox provides all the functionality necessary for you to implement virtually any handshaking scheme that might be required.

Due to the use of all available interrupt modes of the 8250, one user has discovered an unusual set of circumstances that can be troublesome. If the 8250 chip detects an error condition, such as a parity error, framing error, or data overrun error, it causes an interrupt to which the ToolBox will respond. If these errors occur frequently enough, the ToolBox code will spend too much time handling the errors, and lose characters as a result, causing additional errors. If you encounter a situation in which your application appears to behave erratically, especially at higher speeds, investigate the following table.

Table 3.1: Possible Error Conditions

- Is the cabling to the other device sound and solidly connected.

- Are any of the signals in the cable 'floating' or are they all properly terminated.

- Is the other device known to be functioning properly. We have encountered situations in which a serial port on some devices tend to be sloppy in terms of voltage levels, bit timings, and similar problems. Any or all of these situations can cause the erratic operation to which we referred.

Unless you are very familiar with the interrupt structure of the PC, do not attempt to manipulate the interrupt enable flag outside of the ToolBox. The ToolBox sets and clears the interrupt enable flag at appropriate times and assumes that it has sole control over the flag.

LITECOMM-TP (Tm) TOOLBOX for Turbo-PASCAL

Chapter 4

LITECOMM-TP HISTORY

4.1 VERSION 3.0

Version 3 of LiteComm-TP was a departure from the precedent established in previous version. In versions 1 and 2, LiteComm-TP was implemented strictly in Turbo-PASCAL. The advantages to a strictly high-level language implementation are obvious...as is the drawback of speed. In version 3.0, the kernel interrupt handlers were re-implemented in assembly language. In addition, version 3 contained some fundamental function additions that will be of use to those attempting to develop bulletin board systems. In addition, we feel that these routines will serve as a tutorial on how to approach certain communications problems.

With version 3, we also included interrupt chaining for COM3 and COM4, that users of the C version of LiteComm have enjoyed since version 2.

4.2 NEW IN VERSION 5

There will be no version 4 of the LiteComm-TP ToolBox. We have opted to bring out the next revision as Revision 5 to maintain consistency with the C version.

Version 5 features significant enhancement to the kernel interrupt handlers and supporting code, resulting in higher baud rates on older, slower systems. In addition, the tighter code should result in smaller applications.

And with version 5, we have also upgraded and enhanced the XModem and YModem protocol engines with additional capabilities. If you are interested in the protocol engines, see the separate documentation for them. Note that the protocol engines are a part of the registered version of LiteComm-TP only.

Due to a lack of interest in Windowed XModem, we have decided to drop support for that engine, in favor of the other, more advanced protocols. For your convenience, we will continue to provide the source code for the WXModem engine, but we cannot answer questions or support it.

We had originally planned to include a ZModem protocol engine as part of version 5. While that engine is nearing completion, we did not want to hold up the newest version of the main package. Nor did we want to release something like ZModem until we are satisfied with its performance. ZModem is coming in the next major release.

Chapter 5

BEYOND COM2

5.1 THE TOOLBOX METHODOLOGY

In the design of the original PC, and in subsequent variations such as the PC/AT, there were only provision for two serial ports. Many manufacturers of add-in products, both serial ports and internal modems have added the capability to support 1 or more additional ports beyond the COM2 limit. Generally, this can cause problems in the PC since there is no room in the interrupt request scheme for additional levels of interrupts, and there are no designated interrupt vectors for other additional ports.

The ToolBox approach to resolving these issues is to permit the programmer a degree of control over the parameters that govern the interrupt mechanism for COM3 and COM4. Specifically, these parameters are:

1. The interrupt request (IRQ) bit that is used to mask the 8259 interrupt controller.

2. The interrupt vector number (not address) to which the port is attached.

3. The base i/o register for the port itself. Of course, it is assumed that the port is based upon the 8250 UART or compatible device.

Before you attempt to use COM3 and/or COM4, you must determine from the port's documentation, the appropriate values for these three parameters. As distributed, the ToolBox assumes the following:

Table 5.1: COM3 and COM4 Default Settings

	COM3	COM4
IRQ Bit	4	3
Vector #	0Ch	0Bh
Base Reg	3E8h	2E8h

You may change any or all of these values by using the PortChange function described below, but only before you open the port with CommOpen. Once the port has been opened, PortChange is ineffective, and PortChange will not work at all on COM1 or COM2.

At present, LiteComm is not compatible with multiport boards such as the Digiboard. The structure of these boards generally require additional programming to be used effectively. If you want to use such a board with LiteComm, please contact us directly for information on custom modifications to LiteComm. We have performed such modifications for other LiteComm users.

5.2 CAUTIONS

There is an intimate relationship between the IRQ setting and the interrupt vector to which it relates. In the PC, this relationship is controlled, in part, by the 8259 interrupt controller that is set during BIOS initialization.
In brief, the BIOS settings for the PC (and most close compatibles) establish IRQ0 as being vector number 08h, and subsequent IRQ levels at increasing vector numbers. These vector numbers (or types in INTEL terms) act as a cpu-directed call table to locations in the lowest 1K of system memory. We can alter how the system responds to a given interrupt by replacing or changing the values in the associated vector position to point to a routine which we supply.

COM3 and COM4 share two critical parameters with COM1 and COM2 respectively, the IRQ bit and the interrupt vector number. If you use COM3 and COM4 with the default IRQ and Vector values, and you have a COM1 or COM2 installed in your system, LiteComm will chain (share) the interrupt vector. Otherwise you must change both the IRQ and Vector using the PortChange facility. Please remember, the ability for your add-on ports to handle such a change is highly hardware dependent, so check your port's documentation carefully. Failure to do so will result in loss of data at best, and a system lockup at worst.

Judging from the questions asked by some users of LiteComm-TP, there is evidently some misunderstanding about using ports beyond COM2, and how this all relates to your hardware. Before you can successfully use COM3 or COM4, you must consider the following:

1. Does the hardware permit a change to the base port and/or the interrupt vector to which the port responds. Some expansion cards will support changing one and not the other, giving rise to potential hardware conflicts and lost data.

2. Does the hardware permit reassignment of the IRQ priority. Some expansion cards permit you to alter the IRQ priority, some won't. Suffice it to say from the previous discussion the any change to the IRQ priority must allow a corresponding change to the interrupt vector number. Without this capability, reprogramming of the 8259 chip would be required.

3. In fact, many add-on cards permit this dual change simply by making a single switch or jumper setting. Unfortunately, the documentation for these cards generally assume that you are aware of the dual nature of the IRQ vector relationship, and may leave you with the impression that you are changing one and not the other. In most circumstances, this is not the case.

The point to all of this is that LiteComm-TP can only provide as much support as the hardware permits, or is capable of responding to. If you wish to use other than the default base port, interrupt vector, or irq priority for COM3 or COM4, then your expansion card must be capable of supporting the new values; in other words, these values are all hardware-provided, and are recognized by the LiteComm-TP software. If your hardware does not permit changing a value, LiteComm-TP cannot improve the situation.

We should, at this point, add one final caution about how interrupt priorities function, and their relationship to the irq bit the you may select. The standard PC permits 8 interrupt priority levels, with the programmable timer having the highest priority, and the parallel printer port having the lowest priority. When an interrupt occurs, the interrupt controller (8259 chip) masks out all other interrupts from the priority of the interrupting device and all lower priority devices.
As an aid to making COM3 and COM4 "fit", LiteComm-TP supports interrupt chaining for the COM3 and COM4 ports. If you use COM3 or COM4, when an interrupt occurs, the kernel will attempt to determine if the interrupt was caused by the supported port or from some other source.

If the kernel determines that the supported port did not cause the interrupt, an automatic chain to the original interrupt handler for that interrupt level (IRQ level) will take place, allowing you to "patch in" or share the available interrupt vectors.

If you intend to use other than the provided defaults, be sure that you understand the interrupt mechanism. The use of the automatic chaining described above can be particularly troublesome under some circumstances, resulting in loss of interrupts and, potentially, a system crash.

DO NOT attempt to mix the ToolBox functions with other seemingly-related functions (such as the serial port BIOS routines in Turbo PASCAL). At least two users have attempted to only use the receive portions of LiteComm, while resorting to the BIOS functions to send characters or adjust port parameters such as baud rate. The results, at best, have been failure of the user's application to function, and, at worst, total system lockup. This mix of functions is NOT supported and must not be used. If you attempt such a mix, we cannot help you.

One final caution is in order. One or two users have attempted to trace through the interrupts as they occur using debuggers. This

is a risky proposition at best since most debuggers work by tapping into, and disturbing, the interrupt mechanism. If you feel you must use a debugger, try to stay away from the kernel routines of LiteComm, or use a hardware-based debugger such as Periscope.

5.3 OTHER GENERAL NOTES AND WARNINGS

In the discussion of the various functions which follow, you should assume that any references to the 'port' variable refer to a variable or constant that may take on a value of from 1 to 4. No other values are acceptable, and will be rejected.

While we feel that LiteComm-TP is written in the most efficient way possible, commensurate with good programming practice, we cannot be responsible for variations caused by hardware configurations or other factors beyond our control. LiteComm-TP has been tested, and is known to perform on, the IBM PC-AT and several compatible systems such as the Zenith and Wyse equivalents. LiteComm-TP has not been tested in environments in which other software, most significantly TSR (terminate and stay resident) modules exist. Some TSR programs that "steal" interrupts for their own purposes present an unfavorable environment to other programs that rely on the interrupt structure of the computer.

Should you experience erratic behavior with LiteComm-TP in a TSR-type situation, try executing your application without the TSR module being present. If the problems you have experienced disappear, suspect the TSR module.

Conversely, LiteComm-TP provides an excellent vehicle for supporting TSR programs that you may write. Since the package is fully re-entrant, your only concern need be with those aspects of TSR programs are of normal concern, e.g. the non-reentrant nature of DOS. LiteComm-TP never uses DOS functions and may therefore be safely used in a TSR environment.

5.4 USE WITH MULTITASKING ENVIRONMENTS

Some users have made attempts, with varying degrees of success, at using LiteComm in conjunction with multitasking environments such as Quarterdesk's DesqView. Use of LiteComm in such an environment is certain to be affected by the way in which the multitasking monitor behaves with respect to interrupts.

While we recognize that DesqView has achieved a certain measure of popularity among so-called power users, LiteComm was not explicitly designed for such an environment, and its performance may suffer as a result.

5.5 NOTES ON RING DETECTION

Several users have reported difficulty in consistently detecting a ringing telephone by checking the state of the RI (Ring Indicator) signal. The problem seems to be highly dependent on the type of modem that is being used since this signal is provided by the modem, NOT the serial port. If the duration of the signal is too short, the program may 'miss' the signal as the modem toggles it on and off. One workaround that has been used successfully is to check the DeltaRI bit that can be obtained from the ModemStatus function, rather that the RI bit itself. The DeltaRI bit will be set when the RI bit comes on and again when the RI bit goes off. This is the method we employ in the CheckForCall function.

LITECOMM-TP (Tm) TOOLBOX for Turbo-PASCAL

Chapter 6

PACKAGE CONTENTS

Your distribution diskette contains several files that are important to you. All distribution diskettes, at a minimum, include the following files in the diskette's root directory:

Table 6.1: Basic Diskette Contents

READ.ME	LATEST INFORMATION ABOUT LITECOMM-TP
LTCOMM.ARC	SHAREWARE VERSION AND DOCUMENTATION FILES
LTUNITS.ARC	FULLY FUNCTIONAL UNIT FILES

If you registered for the source code modules, the diskette contains 2 additional source code archives.

LITECOMM-TP SOURCE CODE	LTSRC.ARC
XMODEM ENGINE SOURCE CODE	LTXMSRC.ARC

6.1 INSTALLATION INSTRUCTIONS

In the following discussion, we assume that your regular unit files are contained in a directory called \TP.
To install the unit files used with LiteComm-TP, perform the following steps:

1. CD \TP

2. ARC E A:LTUNITS

Since Turbo PASCAL permits you the flexibility of having a separate subdirectory for units, you should execute the above instructions in whatever directory you use for units.

If you are installing only the units, this completes the installation procedure. If you have registered for the package's source code, we recommend that you create a separate subdirectory. The example below assumes that you will use a directory named COMM

to hold the LiteComm-TP and XModem source code modules. To install the LiteComm-TP source code, do the following:

1. MD \COMM

2. CD \COMM

3. ARC E A:LTSRC *.*

4. ARC E A:LTXMSRC *.*

We strongly urge that you use the recommended approach in handling the source code to avoid naming conflicts that might arise.

Chapter 7

PROCEDURE AND FUNCTION REFERENCE

In the following pages, we provide the detailed information about each of the available LiteComm-TP ToolBox functions and procedures. Each definition includes, at a minimum, a summary of how the function or procedure is referenced, in which unit the function or procedure is found, a description of what the function or procedure does, and an indication of those values, if any, that might be returned.

Where appropriate, we include additional documentation about the function. Some definitions include examples, in the sense of code fragments illustrating the function's usage. More importantly, some definitions include additional notes and warnings as well as references to other functions within the package.

We have made every effort to insure that the documentation of the functions is complete and accurate. The style and manner of the documentation assumes that the reader is thoroughly familiar with the elements of PASCAL syntax and common conventions.

7.1 UNIT USAGE

To assist you in developing your own applications, you will need to know the following information.

Unit	Uses
LctKrnl	DOS
LctSupp	DOS, LctKrnl
LctHayes	DOS, LctSupp
LctBBs	DOS, CRT, LctKrnl, LctSupp
LTXMKrnl	DOS, LctSupp
LTXModem	DOS, LctSupp, LTXMKrnl

PortChange function

UNIT — LctKrnl

FUNCTION — Changes one or more of the critical parameters for port COM3 or COM4.

DECLARATION — PortChange(CPort:integer; NewBase:word; NewIrq, NewVector:byte)

RESULT TYPE — boolean

REMARKS — This function must be used before the port is opened to be effective. To leave any of the parameters at its default value, make the corresponding entry 0. Note that vector is a vector number, not an address or pointer.

The irq parameter should not be taken to be the irq (interrupt request number), but rather the irq mask. For example, the correct value for irq4 is NOT 4, but a byte in which bit 4, using INTEL's bit numbering, is set to a value of 1. Thus, to use irq priority 4 as the irq for either COM3 or COM4, you would specify $10 as the value of irq when calling PortChange.

If you intend to change the default irq settings, you MUST also make a corresponding change to the vector number. See the preceding section about using COM3 and COM4 for additional details. Failure to follow this rule may make the port appear to be nonfunctional.

The PortChange function does NOT check to determine that you have provided both an IRQ mask AND a new vector number.
PortChange returns a value of TRUE if the change was successful, false otherwise.

EXAMPLE

```
Var
  Newbase : word;
  Newirq : byte;
  NewVector : byte;
Begin
  Newbase := $03E8;
  Newirq := $10;
```

```
  NewVector := $0C;

  if PortChange(3, Newbase, Newirq, NewVector) then
    Writeln('Port 3 Changed')
  else
    Writeln('Error changing Port 3');
  end;
```

CommOpen function

UNIT — LctKrnl

FUNCTION — Prepares the specified port for use by the other functions

DECLARATION — CommOpen(CPort, Baud:integer; Parity:char; Databits, Stopbits, InSize, OutSize:integer; RaiseMdmCtl:BOOLEAN)

RESULT TYPE — boolean

REMARKS — Opens the specified port for use and attaches an interrupt handler to DOS for the port. The function allocates buffers for input and output of the specified sizes, and sets the port to the parameters specified. The minimum value for InSize is 128; the minimum size for OutSize is 64. A port opened in this manner must be closed using CommClose before program termination to avoid the possibility of a system crash.

CommOpen sets aside an additional 512 bytes per open port. This additional memory is used as the stack for the port's interrupt handler while interrupts are being processed. This approach help avoid the possibility of stack overflows that might occur under some conditions.

CommOpen installs an exit procedure to protect DOS from problems that might arise if a program using LiteComm fails. However, we recommend that you always close a port opened with CommOpen by calling CommClose explicitly in your program to gain maximum control over the port.

The parameters passed to the function are discrete values, and must be drawn from the following lists:

Baud	any value that your communication equipment, e.g. your modem, will support.
Parity	E, O, N, M, S E - Even O - Odd N - None M - Mark S - Space
Databits	5, 6, 7, 8
StopBits	1, 2

If RaiseMdmCtl has a value of TRUE, CommOpen will automatically raise the handshaking signals DTR and RTS. If RaiseMdmCtl has a value of FALSE, the programmer must raise these signals by calling SetModemSignals. In general, use the second form to gain full control over the handshaking signals if needed, use the first form in those cases where the control of these two signals is noncritical.

A return of TRUE indicates the port has been successfully opened and is ready for use. A return of FALSE indicates an error occurred, either as the result of an invalid parameter, or insufficient heap space available to allocate the buffers and control structures for the port.

EXAMPLE

```
Var
  Baud, Databits, Stopbits : integer;
  Parity : char;
  Insize, Outsize : integer;

begin
  Baud := 2400;
  Parity := 'E';
  Databits := 7;
  Stopbits := 1;
  Insize := 256;
  Outsize := 256;

  if CommOpen(1, Baud, Parity, Databits, Stopbits,
Insize, Outsize, TRUE) then
    Writeln('COM1 available for use')
  else
    Writeln('Error opening COM1');
```

The equivalent code to the above is shown below

```
  if CommOpen(1, Baud, Parity, Databits, Stopbits,
Insize, Outsize, FALSE) then
    Writeln('COM1 is now open')
  else
    Writeln('Error opening COM1');
  if SetModemSignals(1, (DTR OR RTS)) then
    Writeln('Port is ready to transmit')
  else
    Writeln('Unable to set modem signals');
```

SEE ALSO SetModemSignals

CommClose procedure

UNIT LctKrnl

FUNCTION Closes a port that has been opened by the CommOpen function

DECLARATION CommClose(CPort:integer; DropMdmCtl:BOOLEAN)

REMARKS This function is the companion to CommOpen and, in effect, performs the opposite action. CommClose detaches the kernel interrupt handler from the port, and reconnects the previous interrupt handler. CommClose also release dynamically allocated memory used for buffering and control structures. If DropMdmCtl has a value of TRUE, the port is closed absolutely; both the DTR and RTS signals are dropped. If DropMdmCtl has a value of FALSE, the port is closed conditionally and both DTR and RTS are left in their current state.

Since CommOpen installs an exit procedure, you are not required to explicitly close an open port. However, if you do not use an explicit close, you will lose control over the handling of DTR and RTS. The built-in exit procedure always uses the absolute form of the close.

CommSetup function

UNIT — LctKrnl

FUNCTION — Provides the capability of changing the parameters for an open port, without breaking the connection or closing the port.

DECLARATION — CommSetup(CPort, Baud:integer; Parity:char; Databits, Stopbits:integer)

RESULT TYPE — boolean

REMARKS — The CommSetup function is a subset of the CommOpen function and the remarks made in the description of CommOpen apply. This function is useful if you wish to change the basic communication parameters of the specified port that has already been opened without CommClose'ing the port and breaking the connection.

SEE ALSO — CommOpen

BytesInInput function

UNIT — LctSupp

FUNCTION — Returns the number of characters currently available in the input buffer (BytesInInput) or the number of untransmitted characters in the output buffer (ByteInOutput).

DECLARATION — BytesInInput(CPort:integer)
BytesInOutput(CPort:integer)

RESULT TYPE — integer

REMARKS — May be used to determine the number of characters currently in the input (BytesInInput) or output (BytesInOutput) buffers for the port. In the event of an error (bad port), a value of -1 is returned.

ModemStatus function

UNIT — LctKrnl

FUNCTION — Returns the last know status of the modem control lines for the specified port.

DECLARATION — ModemStatus(CPort:integer)

RESULT TYPE — byte

REMARKS — Use this function to determine the last known state of the modem-supplied handshake signals. These may be tested using the values included in the unit, using PASCAL's bitwise AND operator.

The byte value returned can be viewed as consisting of two sub-fields, the current signal state (found in bits 4-7 of the byte), and the signal change(DELTA) indicators(found in the bits 0-3 of the byte). ModemStatus always returns the current state of the signals in bits 4-7. Bits 0-3 will reflect which, if any, of the signals has changed.

Whenever this function is called, both subfields are returned, and represent the current state of the individual signals. The DELTA settings may be all reset, if no signals have changed since the last call to the function. The signals which are tracked are CTS, DSR, RI, and DCD.

To determine which signals, if any, have changed use the DeltaXXX bits returned. For example, if CTS has changed, the DeltaCTS bit will be set. The actual CTS value (on or off) will be found in the CTS bit of the returned byte.

In the event of an error, a byte of $00 is returned.

HINT — Detecting a ringing phone (using the RI signal) can be tricky and timing dependent. One nearly foolproof method that we have used is to examine the DeltaRI value, not the RI value. The DeltaRI value is set and reset as the telephone starts and stops ringing. The RI value is set and cleared independently, and you may miss the fact that the

phone is ringing if you don't examine the value at the right time.

EXAMPLE

```
Var
  CStat : byte;

begin
  CStat := ModemStatus(1);
  if CStat and DeltaDCD = DeltaDCD then
    if CStat and DCD <> DCD then
      Writeln('Remote is off-line, carrier lost');
```

BreakRecd function

UNIT — LctKrnl

FUNCTION — Returns a value of true if a BREAK signal has been received from the serial port since that last call to the function.

DECLARATION — BreakRecd(CPort:integer);

RESULT TYPE — boolean;

REMARKS — This function returns a value of TRUE if a BREAK character has been received since the last call to the function.

EXAMPLE

```
if BreakRecd(2) then
  Writeln('Break Signal detected on Port 2');
```

ErrorStatus function

UNIT — LctKrnl

FUNCTION — Return the last known error status for the specified port.

DECLARATION — ErrorStatus(CPort:integer)

RESULT TYPE — byte

REMARKS

Returns the last known state of the serial port's error status bits, encoded in a byte. These may be tested using the constants defined in the unit in conjunction with PASCAL's bitwise AND operator. The applicable values that may be checked are OverRun, BadParity, and, BadFrame.

1. OverRun - failure to fetch a character from the port before the next character was received. Usually caused by a problem in the interrupt handler.

2. BadParity - One or more characters were received in which the parity of the character(s) did not match the current parity setting of the port. Can be caused by line noise (electrical interference), poor connections, or a variety of other reasons.

3. BadFrame - A framing error has occurred. A character was received that had too few or (more likely) too many bits. Usually caused bye line noise.

Break detection, i.e. the receipt of a BREAK character, is handled by the BreakRecd function(qv). In the event of an error, a byte of $00 is returned.

Once the error status bits have been read in this fashion, they are reset to $00, and will remain so until the next error occurs. Since this process happens asynchronously, it is not possible for your application to determine which character created the error, only that the error occurred.

EXAMPLE

```
Var
  EStat : byte;

begin
  EStat := ErrorStatus(2);
  if EStat and OverRun = OverRun then
    Writeln('Receive Character Over Run');
```

SetModemSignals function

UNIT — LctKrnl

FUNCTION — Allows the programmer to set the individual modem control lines for the specified port.

DECLARATION — SetModemSignals(CPort:integer; NewSet:byte)

RESULT TYPE — boolean

REMARKS — Set one or more of the modem control signals. Because of the need to always have OUT2 set for interrupt support, the function always provides the correct setting for this bit.

The value of NewSet is bitwise OR'ed with the current set of values to produce a new modem control setting. The value of NewSet DOES NOT replace the current values. Use the constants supplied in the unit to obtain the correct values. These include DTR and RTS.

Many applications will not require this, and its companion, functions, if the permit CommOpen to raise the DTR and RTS signals. More sophisticated applications may be required to control either or both of the signals to provide handshaking with an external device.

EXAMPLE

```
begin
  (* raise both DTR and RTS *)
  if SetModemSignals(1, (DTR OR RTS)) then
    Writeln('Port is ready to transmit')
  else
    Writeln('Unable to set modem signals');
```

SEE ALSO — ClearModemSignals, FlipModemSignals

ClearModemSignals function

UNIT LctKrnl

FUNCTION Allows the programmer to clear (reset) the individual modem control lines for the specified port.

DECLARATION ClearModemSignals(CPort:integer; NewSet:byte)

RESULT TYPE boolean

REMARKS Clears (resets) one or more of the modem control signals. Because of the need to always have OUT2 set for interrupt support, the function always provides the correct setting for this bit.

The compliment of NewSet is bitwise AND'ed with the current set of values to produce a new modem control setting. The value of NewSet DOES NOT replace the current values. Use the constants supplied in the unit to obtain the correct values. These include DTR and RTS.

EXAMPLE

```
begin
  if ClearModemSignals(1,  RTS) then
    Writeln('RTS for Port 1 has been dropped')
  else
    Writeln('Unable to clear RTS');
```

SEE ALSO SetModemSignals, FlipModemSignals

FlipModemSignals function

UNIT LctKrnl

FUNCTION Allows the programmer to compliment(toggle) the individual modem control lines for the specified port. This function only has value if you are attempting to implement some form of hardware handshaking with another device. The absolute form

of CommClose will automatically lower both RTS AND DTR, which is generally adequate for many applications.

DECLARATION FlipModemSignals(CPort:integer; NewSet:byte)

RESULT TYPE boolean

REMARKS Complements(toggles) one or more of the modem control signals. Because of the need to always have OUT2 set for interrupt support, the function always provides the correct setting for this bit.

The value of NewSet is bitwise XOR'ed with the current set of values to produce a new modem control setting. The value of NewSet DOES NOT replace the current values. Use the constants supplied in the unit to obtain the correct values. These include DTR and RTS.

EXAMPLE
```
begin
  (*
  ** change the RTS modem control signal to its
other
  ** state (if raised, lower it, if lowered raise
it)
  *)
  if FlipModemSignals(1, RTS) then
    Writeln('RTS for Port 1 has been changed')
  else
    Writeln('Unable to change RTS');
```

SEE ALSO SetModemSignals, ClearModemSignals

EnableXon function

UNIT LctKrnl

FUNCTION Enable or disable the semiautomatic flow control features of LiteComm-TP

DECLARATION EnableXon(CPort:integer; XonFlag:boolean)

RESULT TYPE boolean;

REMARKS If XonFlag is TRUE, turns on semiautomatic XON-XOFF flow control function. If XonFlag is FALSE (the

default setting), automatic flow control is disabled.
When enabled, the kernel will automatically transmit an XOFF if and when the input buffer is approximately 2/3 full and will automatically recognize an XOFF sent by the other device. If the other device transmits an XOFF, the kernel will refuse to send any characters until the condition is cleared, either by receipt of an XON, by calling the XOffRecd function, or by disabling XON-XOFF altogether.

The XOFF recognition is implemented in the kernel. As a result, the transmit buffer will continue to accept input from your program until the buffer fills completely, even though the information will not be sent. Once the matching XON is received, the contents of the transmit buffer will be sent rapidly to the other device. It is possible that the rate with which characters are sent when this occurs may cause problems for the other device, depending on its ability to handle the data flow.

If the kernel has sent an XOFF, it is the programmer's responsibility to transmit XON when conditions warrant. Use the XoffSent function to tell if an automatic XOFF has been sent by the kernel.

If you intended to implement a protocol that might include the XON-XOFF characters, be sure to disable the automatic flow control. Failure to do so may result in a system hang.

SEE ALSO XoffRecd, XoffRecd

XoffRecd function

UNIT LctKrnl

FUNCTION Reports whether or not the kernel has detected an XOFF from the other device

DECLARATION XoffRecd(CPort:integer)

RESULT TYPE boolean

REMARKS Returns TRUE if an XOFF has been received, FALSE otherwise. If an XOFF has been received, the port's internal flag will be reset, and transmission to the other device will be permitted. If an XON is received from the other, the port's flag will also be reset, permitting further transmissions to occur.

If you use flow control and the other device never sends an XON after sending an XOFF, a system hang is possible. You may wish to call XoffRecd periodically to test for this condition, and to cause your port to resume transmitting data.

SEE ALSO EnableXon, XoffSent

XoffSent function

UNIT LctKrnl

FUNCTION Reports whether or not the kernel has automatically sent an XOFF to the other device.

DECLARATION XoffSent(CPort:integer)

RESULT TYPE boolean

REMARKS Returns TRUE if the LiteComm kernel has sent an XOFF to the other device, FALSE otherwise. If an XOFF has been sent, the port's flag will be reset. You must send an XON character to the other device to permit transmissions to proceed.

SEE ALSO EnableXon, XoffRecd

LctGet function

UNIT LctSupp

FUNCTION Returns an available character from the ports input buffer.

DECLARATION LctGet(CPort:integer; var Ch:byte)

RESULT TYPE boolean

REMARKS Places the next available character in the input buffer for the port in the variable Ch. The function returns a value of TRUE if there is a character available, FALSE if there is no character available or on an error. The contents of Ch are undefined when the return is FALSE.

If you specified other than N (No Parity) when the port was opened, you may have to reset (make zero) the parity bit before you use the character.

EXAMPLE
```
BEGIN
  IF LctGet(CPort, Ch) THEN
    Ch := Ch AND $7F;       (* Reset the Parity Bit
*)
```

LctPeek function

UNIT LctSupp

FUNCTION Permits you to look at the next character in the ports input buffer without removing if from the buffer.

DECLARATION LctPeek(CPort:integer; var Ch:byte)

RESULT TYPE boolean

REMARKS Places the next available character in the input buffer for the specified port into the Ch variable, but does not remove the character from the buffer. This allows the application to look-ahead by one character in a nondestructive fashion. Returns FALSE if the port is not active, or if there are no characters in the port's buffer, TRUE otherwise. The contents of Ch are undefined when the result is FALSE.

The comment made regarding parity setting and the use of the LctGet function also applies to LctPeek.

SEE ALSO LctGet

LctPut function

UNIT LctSupp

FUNCTION Places a character in the port's transmit buffer to be sent when the port is ready.

DECLARATION LctPut(CPort:integer; Ch:byte)

RESULT TYPE boolean

REMARKS Returns TRUE if successful. Note that this does not guarantee that the character has been sent, only that no errors were detected, and there was space in the transmit buffer to hold the character. Returns FALSE if the port is not active, or if there no room in the port's buffer.

Characters are sent from the transmit buffer when the system has the time to send them, assuming that all conditions for transmission are satisfied

GetStream function

UNIT LctSupp

FUNCTION Gets a stream of N characters from the port's input buffer

DECLARATION GetStream(CPort:integer; var Buff; BCnt:integer)

RESULT TYPE integer

REMARKS Reads a stream of, at most, BCnt characters from the serial port's input buffer into the Buff array. Returns the count of characters actually transferred, or -1 if an error occurs.

NOTE that Buff is an array of characters or bytes, not a string, although you may treat a string variable like an array, as shown below. The

comments made about parity in the LctGet function description also apply the the GetStream function.

EXAMPLE

```
Type
  MaxStr = string[256];

Var
  StrBuff : MaxStr;
  RecdLen : integer;

begin
  RecdLen := GetStream(2, StrBuff[1], 256);
  if RecdLen <M=> 0 then     { error or no char }
    StrBuff[0] := 0
  else
    StrBuff[0] := Chr(RecdLen);
end;
```

SEE ALSO LctGet

PutStream function

UNIT LctSupp

FUNCTION Places a stream of, at most, N characters in the port's transmit buffer.

DECLARATION PutStream(CPort:integer; var Buff; BCnt:integer)

RESULT TYPE integer

REMARKS Buff is an array of character or byte, not a string, although it is possible to specify a string variable, using the same approach as outlined for the GetStream function. PutStream returns the number of characters actually placed into the buffer. Note that this does not guarantee that the characters have been sent. A value of 0 will be returned if any error occurs, or if there no room in the port's buffer.

EXAMPLE

```
VAR
  Buff : ARRAY[1..128] OF BYTE;
  LeftToSend : INTEGER;
  ReallySent : INTEGER;
  StartPos   : INTEGER;

BEGIN
```

```
  LeftToSend := 128;                (* set up for
Full Length *)
ReallySent :=  0;
  StartPos := 1;
  WHILE LeftToSend > 0 DO
  BEGIN
    ReallySent := PutStream(CPort, Buff[StartPos],
LeftToSend);
    IF ReallySent > 0 THEN
    BEGIN
      StartPos := StartPos + ReallySent; (* adjust
start byte *)
      LeftToSend := LeftToSend - ReallySent;
    END;
  END (* while *);
```

SEE ALSO PutStream

Buffer Flushing functions

UNIT LctSupp

FUNCTION Provides several high level buffer management functions to control the contents of the port's transmit and receive buffers

DECLARATION

```
function PurgeTxBuff(CPort:integer)

function PurgeRxBuff(CPort:integer)

procedure FlushUntilMatch(CPort:integer; Ch:byte)

procedure FlushNBytes(CPort:integer; N:integer);
```

RESULT TYPE boolean for PurgeTxBuff, PurgeRxBuff

REMARKS The PurgeRxBuff and PurgeTxBuff functions remove all characters from the port's receive and transmit buffers respectively and discard them; untransmitted characters in the transmit buffer are NEVER sent; unprocessed characters in the receive buffer are lost. Both functions return a value of TRUE if no errors were encountered, FALSE otherwise. An empty buffer is NOT considered an error.

The FlushUntilMatch procedure will continually dispose of received characters until the character

Ch is received. The procedure will return when the character Ch is detected, or when there are no more characters in the port's input buffer.
The FlushNBytes procedure removes, at most, N characters from the port's receive buffer.

SendBreak function

UNIT LctKrnl

FUNCTION Send a true Break signal

DECLARATION SendBreak(CPort:integer)

RESULT TYPE boolean

REMARKS SendBreak generates a BREAK signal using a particular characteristic of the 8250 UART to generate an accurate BREAK at any baud rate. BREAKs generated in this manner are timed based upon the baud rate at which the 8250 is currently initialized. This function may or may not work correctly with other than the actual 8250 UART.

Returns TRUE if successful. Returns FALSE if an error is detected.

CheckEvent Function

UNIT LctBBS

FUNCTION Returns a value of TRUE is the Event Timer specified in the function call has not expired. Returns a value of FALSE if the specified Event Timer has expired. Do not attempt to use CheckEvent against a variable that was not initialized bye NewEvent. The results are unpredictable, and may result in an apparent system hang.

DECLARATION CheckEvent(EventVal : Event);

RESULT TYPE boolean;

REMARKS The event timer specified by EventVal must have been set using the NewEvent function

SEE ALSO NewEvent

NewEvent Function

UNIT LctBBS

FUNCTION Initializes an event timer to a value suitable for use with CheckEvent. The event timer created in this fashion can time events up to 32767 seconds in duration.

DECLARATION NewEvent(Seconds : integer);

RESULT TYPE Event;

REMARKS When used in conjunction with CheckEvent, the event timer can be used to time events that span days, months or years. Actually, it should be termed a timeout timer, since CheckEvent checks to see if the period specified by Seconds has elapsed.

EXAMPLE

```
var
  InputEvent : Event;
  Ch : byte;

begin
  InputEvent := NewEvent(15);
  while CheckEvent(InputEvent) do
  if LctGet(Port, Ch) then
    exit;
  WriteLn('No Input Received in 15 seconds');
end;
```

CheckForCall Function

UNIT LctBBS

FUNCTION 'Listens' to the specified port to see if the telephone is ringing. If the phone is ringing, waits for up to 30 seconds for a successful connection to be established.

DECLARATION CheckForCall(CPort : integer);

RESULT TYPE integer;

REMARKS This function will return a value of -1 if the phone is not ringing, or if the modem fails to respond to the call within the 30 second period allowed. In all other cases the function returns the result code that was returned by the modem itself. It is the programmer's responsibility to correctly recognize and react to the various codes. In the case of a failure of the modem to respond, this function will automatically attempt to disconnect and reset the modem.

The function assumes that the modem has been set up to use numeric result codes, and that the S0 register (number of rings before answering) has not been set to zero. The function ResetModem sets the correct values to match these assumptions. CAUTION - do not attempt to use this function on ports not connected to a modem. The function examines the modem control status lines and may behave in a unpredictable fashion if not connected to a modem.

EXAMPLE

```
var
  ModemResult : integer;

begin
  repeat
  ModemResult := CheckForCall(CPort);
  if ModemResult = -1 then
    Delay(1000);  (* settling time*)
  until ModemResult <> -1;
  Writeln('Modem reply to call was ',
ModemResult:2);
end;
```

SEE ALSO Disconnect, ResetModem

Disconnect Procedure

UNIT LctBBS

FUNCTION Causes the modem to disconnect from the caller.

DECLARATION Disconnect(CPort : integer);

REMARKS Disconnects the modem by dropping the DTR (Data Terminal Ready) modem status signal for 1 second. This action will cause most modems to drop carrier and force the phone on-hook. Please not that if the modem has been optioned with DTR permanently set on or ignored, this procedure will have no effect.

ResetModem Function

UNIT LctBBS

FUNCTION Returns the modem to a known set of parameters, suitable for use with the related functions in this unit. See the typed constants MODEMSET0 through MODEMSET2 in the interface portion of the unit.

DECLARATION ResetModem(CPort, : integer);

RESULT TYPE integer;

REMARKS The modem is reset to a known state, including, but not limited to 1) answer on the first ring, 2) numeric result codes, 3) extended code set. The function returns the result of the reset operation (a modem response code) or -1 if the modem fails to respond. This function and related functions are suitable for use only with HAYES-type modems. It is the programmer's responsibility to interpret the result code returned.

SEE ALSO GetModemReply

GetModemReply Function

UNIT LctBBS

FUNCTION Returns the modem's response to the last set of instructions that were issued to the modem, in numeric form.

DECLARATION GetModemReply(CPort : integer);

RESULT TYPE integer;

REMARKS This function expects the modem to be returning numeric result codes (set ResetModem) of up to 2 digits. The function will react to 2 digits returned or the first <CR> returned, whichever occurs first within a 1 second timeout period. In the case that the modem does not respond in the timeout period, the function returns a value of -1. In no case does the function attempt to evaluate the response...this is left to the programmer.

SEE ALSO ResetModem.

HAYES MODEM FUNCTIONS

UNIT LctHayes

FUNCTION Provides support for various aspects of modems the support the Hayes(Tm) command set.

DECLARATIONS

```
const
  NUMRES = 0     { numeric result codes}
  WRDRES = 1     { word result codes }
  SPKOFF = 0     { speaker off }
  SPKON  = 1     { speaker on until CD }
  SPKSPC = 2     { speaker always on }
  ONHK   = 0     { go on-hook (hang up) }
  OFFHK  = 1     { go off-hook (lift receiver) }
  OFFHKS = 2     { go off-hook, don't close relay }
  BASIC  = 0     { basic result set }
  EXSET1 = 1     { extended results, set 1 }
  EXSET3 = 2     { extended results, set 3 }
  EXSET4 = 3     { extended results, set 4 }

type
  TelNumStr = string[20];

procedure SetType(NType : byte)

procedure SetSet(NSet : byte)
```

```
function RetType : byte

function RetSet : byte

function ModemCodesOn(CPort : integer):boolean

function ModemCodesOff(CPort : integer):boolean

function ModemWordResponse(CPort : integer):boolean

function ModemDigitResponse(CPort :
integer):boolean

function RepeatModemCommand(CPort :
integer):boolean

function ModemSpeaker(CPort : integer; Mode :
byte):boolean

function SetModemRegister(CPort, Reg, NValue :
integer) : boolean

function GetModemRegister(CPort, Reg, NValue :
integer) : boolean

function ModemHalfDuplex(CPort : integer) : boolean

function ModemFullDuplex(CPort : integer) : boolean

function ModemEchoCmd(CPort : integer) : boolean

function ModemNoEchoCmd(CPort : integer) : boolean

function ModemHookMode(CPort : integer; HMode :
byte) : boolean

function ModemCarrierOn(CPort : integer) : boolean

function ModemCarrierOff(CPort : integer) : boolean

function ModemWordResponse(CPort : integer):boolean

function ModemDigitResponse(CPort :
integer):boolean

function RepeatModemCommand(CPort :
integer):boolean

function ModemSpeaker(CPort : integer; Mode :
byte):boolean

function SetModemRegister(CPort, Reg, NValue :
integer) : boolean
```

```
function GetModemRegister(CPort, Reg, NValue :
integer) : boolean

function ModemHalfDuplex(CPort : integer) : boolean

function ModemFullDuplex(CPort : integer) : boolean

function ModemEchoCmd(CPort : integer) : boolean

function ModemNoEchoCmd(CPort : integer) : boolean

function ModemHookMode(CPort : integer; HMode :
byte) : boolean

function ModemCarrierOn(CPort : integer) : boolean

function ModemCarrierOff(CPort : integer) : boolean

function ModemCodeSet(CPort : integer; NewSet :
byte) : boolean

function ModemPulse(CPort : integer) : boolean

function ModemTone(CPort : integer) : boolean

function ModemDial(CPort : integer; TelNo :
TelNumStr) : boolean
```

REMARKS

The ModemCodeSet function allows you to change the set of codes that are returned by the modem when an action is specified.

ModemDial instructs the modem to dial the number contained in TelNo. Do not include the dialing (ATD) prefix, or the trailing <CR>. These are provided by the function. You may include non-numeric characters as the contents of TelNo are not checked. The dialing is done in the last known, pulse or tone, mode. If you use the Modempulse or ModemTone functions, then dialing will be done in the mode that was last correctly enabled. If you have not exercised these functions, then dialing occurs in the modems default or power-up mode.

The ModemHalfDuplex and ModemFullDuplex functions place the modem into local echo and remote echo modes respectively.

The GetModemRegister function requests that the modem return the current value of S-register Reg. Reg must be in the range 0 to 13. Use the GetStream, or similar function, to retrieve the modem's response. Specifying a register outside the 0 to 13 range will cause a return of FALSE.

SetModemRegister is the companion to GetModemRegister, with the same restrictions. Sets the S-register Reg to the value contained in NValue. If NValue contains -1, then the register is reset to its default (power-up) value. The function checks the value to be certain that it is within the limits specified for the particular register, and returns a value of FALSE if the value is outside the predefined limits.

ModemCarrierOff enables modem carrier detect, but disables the modems carrier signal. The ModemCarrierOn companion enables both carrier detect and the modems carrier signal. When ModemCarrierOff is used the modem will receive data but will be unable to send data.

The ModemNoEchoCmd and ModemEchoCmd functions determine whether commands sent to the modem are echoed back to the sending program. With echo turned off, the modem will continue to accept commands, but will not try to redisplay the command's characters.

ModemHookMode allows you to control the current status of the modem's telephone line connection. See your modem's documentation and the above constants for additional information.

The ModemRepeatCommand function instructs the modem to repeat the last command sequence executed. Generally, this function is of greatest value in redialing numbers that are busy, although its use is not restricted to that.

The way in which your modem responds to commands is determined, in part, by the ModemWordResponse and ModemDigitResponse functions. If you call ModemWordResponse, then modem responses use the English language response codes, e.g. CONNECT or OK. Calling ModemDigitResponse instructs the modem to respond with code numbers only from the currently selected response set, see the ModemCodeSet function above.

You may use the functions ModemCodesOn and ModemCodesOff to specify whether you want your modem to send back response codes when it receives a command string. In a sense, these act as companions to the EchoCmd functions above.

Use the ModemSpeaker function to control the modem's internal speaker, if it has one. See the above constants for the applicable codes.

The RetSet and RetType functions return, respectively, the last known command set (ModemCodeSet) and last known result type (ModemWordResponse, ModemDigitResponse). The RetSet and RetType functions are only of value when used in conjunction with the companion functions.

GENERAL REMARKS Several considerations are in order if you intend to use the Hayes ToolBox functions.

1. You are responsible for checking the return codes from the modem once you have given modem a command. To make the task easier, we suggest that you turn OFF command echo (so that you don't have to worry about separating commands from responses) and turn ON numeric responses (to make the interpretation of result codes easier and faster).

2. Be sure that you allow adequate time between commands for the modem to process the command and respond. Failure to observe this rule may result in commands being misinterpreted or "lost". You can monitor the number of characters in the receive buffer to help you with the timing. Or alternatively, check the response after each command. The latter approach is more in line with what we believe to be good programming practice.

RETURN VALUES All functions return a value of FALSE if a port or other error is detected, TRUE otherwise.

Index

Contents

Figures

Tables