Extending Procomm Plus®
Using Dynamically Linked Terminal Emulations

Table of Contents

Introduction

See the document ***Procomm Plus® 32 Interface Specification Overview and Shared Callback Functions*** for introductory information and callback functions which can be used by any DL*x*.

DLT Overview

A DLT is an emulation DLL. If a DLT is detected, by having the filetype **.dlt** and being in the same directory as **pw4.exe**, it will be loaded and queried as to its capabilities. The name of its emulation(s) and whether it requires continuous timer calls at all times are among the information it may be expected to return.

The DLL will be given **Startup**, **Reset**, and **Shutdown** calls for its internal state control. It will be provided with various resources such as buffered file I/O, virtualized Comm functions, handles to windows that it needs, an open printer DC, and a timer call. When called off the timer, the DLL will poll for characters and display them. The DLL will be called with various other commands such as print screen, capture to disk, etc. The DLL will own its display context and can use it at will.

When the **Startup()** function is called, the handles to the terminal window will be passed in and the message procedures for it will be switched by the DLL to its own handler. Multiple instances of Procomm Plus calling the DLL must be supported. One method is suggested in the sample code **simpldlt.c**. Since Procomm Plus requires a dedicated timer, it can reasonably be assumed that no more than 16 instances will be running at one time.

The current state of the interface design is presented below. The current state of the interface code is defined by the DLT-related header files:

- ♦ **gpwindlx.h**
- ♦ **pwdlx.h**
- ♦ **xdlt.h**
- ♦ **menudlx.h**
- ♦ **ddirdlx.h**
- ♦ **cbackdlx.h**
- ♦ **menudlx.h**
- ♦ **kbdredlx.h**
- ♦ **vxprtdlx.h**

- **gathrdlx.h**

Also reference the file **simpldlt.c** which contains simple but functional emulation sample code.

DLL Entry Point

The DLL will export a single entry point which will be called with a function code defining the purpose of the call, a pointer which will point to whatever is needed by the function code, and a pointer to the **pw_dlt** structure. **The pw_dlt** structure is defined in detail in **xdlt.h**, and contains all necessary information for the emulation to run and also provides a place for the DLL to communicate information back to Procomm Plus.

int EXPORTED PASCAL **DLT_Interface** (int functioncode, void * lpvoid, LPPW_DLT lppw_dlt)

The function codes and associated special pointers are defined here. If there is no special pointer defined, it will be sent as a NULL.

⇒ ***Note:*** *The pointer to the* ***pw_dlt*** *structure will not always be sent to the DLT. When it is sent, it allows the DLT to know which instance of Procomm Plus is calling it. When it is not sent, enough information will be passed in the accompanying structure that the DLT will not* ***need*** *to know which instance is calling it. Such functions are clearly marked.*

The **DLT_Interface()** call must set up in the DLL such that if it does not handle a particular functioncode it will return DLT_FUNCTION_NOT_IMPLEMENTED (defined in **xdlt.h**). This allows Procomm Plus to determine whether the DLT was able to handle the specific function code.

Procomm Plus has complete control over the keyboard, which can make keyboard control difficult for DLT's. In particular, keyboard processing occurs in the main message loop rather than in any of the window procedures. Thus, even subclassing Procomm Plus does not provide total keyboard control - only a message hook can truly wrest control. As an alternative, DLLs are provided with a set of three files which Procomm Plus reads and uses to set its keyboard mappings.

Startup (lpStartupStruct)

Startup() swaps the window procedure in **hWndTerminal** to the DLT's window procedure, then sets up the buffers and prepares to be called by **Reset()**, **Timer()**, and **TerminalPollExt()**. If **Startup()** returns FALSE, then Procomm Plus will not switch to the DLT emulation, but will remain in its current emulation.

The emulation defined in the **lpStartupStruct->nEmulationRequested** should be initialized by the .DLT. For example, if the DLT supports three different emulations then the struct will contain a 0, 1, or 2 to define which of the emulations should be initialized. The DLT should return the **lppw_dlt->iDLLMagicIndex** set to a value that allows subsequent calls to be identified as belonging to this instance of Procomm Plus.

At start-up time, various things can be set or changed in the emulation. There is a string in the **StartupStruct** into which can be copied the name of an accelerator file, which defines what is sent for each key that Procomm Plus defines as an accelerator (for definition of accelerator, see above).

⇒ ***Note:*** *For more information, refer to the code example.*

Terminal (NULL)

⇒ *Note: Starting with PROCOMM PLUS version 3.0, this function has been superseded by the TerminalPollExt() function, and thus all terminal polling code should be implemented there. For PW3.xx DLT emulations, Terminal() should just return DLT_FUNCTION_NOT_IMPLEMENTED.*

TerminalPollExt (lpTerminalPollExtStruct)

Called from inside a timer message, this function reads Comm data by calling **P_GRXGet()** and processing the incoming data appropriately. It writes responses to the port by calling **P_AddToTXBuf()** if the data should be ***paced***.

TerminalPollExt() will not be called if Procomm Plus goes into a different state during which Comm I/O by the terminal is not allowed, such as during a file transfer.

TerminalPollExt() should return when the current system tick count (obtained via **GetTickCount()**) exceeds the value passed in the **dwmsEnd** field of the **lpTerminalPollExtStruct** structure.

⇒ ***Note:*** *The DLT should never display any dialog boxes during a call to* ***TerminalPollExt()****.*

For more information, see the code example.

Timer (NULL)

Called by Procomm Plus from within a timer message. **Timer()** is intended exclusively for background processing, such as implementing software blinking on the screen. **Timer()** will always be called on every timer message between the call to **Reset()** and the call to **Shutdown().**

⇒ ***Note:*** *This function is intended for background processing only; never do any Comm I/O or dialog box displays within this function.*

Reset (NULL)

Called by Procomm Plus when dialing to reset the emulation to a known clear state. **Reset()** is also called after **Startup()** so that normal soft reset initialization processing can occur. For further information, see the code example.

ShutDown (NULL)

Called by Procomm Plus when the user has deselected the .DLT's emulation or when Procomm Plus is terminating. **ShutDown()** deallocates all buffers and resets the window handles. For more information, see the code example.

KeyChar (lpCHAR)

Arguments point to the **wParam** of a **WM_CHAR** message. The DLT may process the **WM_CHAR** any way it sees fit. Typically it would be handled by putting it in a string and calling back **P_AddToTXBuf()**. For more information, see the code example.

SpecialKeys (lpSpecialKeysStruct)

This function is available starting with Procomm Plus version 3.xx.

Called when PW senses that the user's pressed a key mapped to send text or send text with echo. Arguments point to a **SpecialKeysStruct** which contains the string mapped to the key, the length, and pause and record flags.

The pause flag says that the string is to be put into the outgoing buffer with each occurrence of PW's pause character translated into pauses. The record flag says that the string is to be recorded, if applicable, and added to the outgoing buffer. For the most part, you can pass those values, as is, right into **P_AddToTxBuf**.

Currently, there is no code example for this interface.

SnapShot_Capture (lpSnapShot_CaptureStruct)

This function is available starting with Procomm Plus version 3.xx.

Called by Procomm Plus to get a snapshot appended to the active capture file. An open file handle will be passed in and the file position will be set for the DLT to the screen contents to. For further information, refer to the code example.

SnapShot_Fax (lpSnapShot_FaxStruct)

This function is available starting with Procomm Plus version 3.xx.

Called by Procomm Plus to get a snapshot to a fax image. Procomm Plus passes in an open hDC for printing, the current default terminal font height, and information regarding the margins and current drawing position. Refer to the code example for further information.

SnapShot_Printer (lpSnapShot_PrinterStruct)

Called by Procomm Plus when a user clicks on the printer icon to get a snapshot to the printer as a bitmap or character (depending on flag). Procomm Plus passes in an open hDC for printing and the flag. For further information, refer to the code example.

SnapShot_File (lpSnapShot_FileStruct)

Called by Procomm Plus to get a snapshot of the screen to an open file. A flag determines the format, which will be one of: (1) text or (2) bitmap. The flag is kept by the DLT in its advanced setup area, which is provided to it by Procomm Plus. Refer to the code example for more information..

SnapShot_Clipboard (NULL)

Called by Procomm Plus when a user requests a snapshot of the terminal screen to the Windows clipboard. A flag determines the format, which will be one of: (1) text or (2) bitmap. The flag is kept by the DLT in its advanced setup area, which is provided to it by Procomm Plus. Refer to the code example for more information.

ReadFromDisplay (lpReadFromDisplayStruct)

Read characters from the display and return them in the buffer. If the desired length goes off the end of the line, then continue to read from the next line. If the end of the display is reached, then pad to the desired length with nulls. Characters displayed as bitmapped fonts will simply be returned as the binary character if possible, or a space if not. See the code example for further information.

⇒ ***Note:*** *The returned buffer does* ***NOT*** *have to be null terminated.*

StartCapture (lpStartCaptureStruct)

Struct contains an open Procomm Plus stream file handle. Incoming characters will be copied to this stream file as they come in. The struct also contains the field nCaptureType, which takes the values 0, 1, or 2. These are defined as:

0 - **“As seen on the screen”**, which implies that the lines are captured as they scroll off of the screen.

1 - **Filtered**, which means regular graphics characters as they come in the port plus Carriage Return, Line Feed, Form Feed, Tab, Back Space, and Vertical Form Feed. Other control characters and all Escape Sequences are stripped out and not sent to the file.

2 - **Raw**, which captures the entire download including escape sequences in such a way that it could be played back and the original screen would be recreated.

Refer to the code example for further information.

EndCapture (NULL)

Terminates capturing data to the capture file. Procomm Plus will flush and close the file handle after calling this function. For more information, see the code example.

StartReplay (NULL)

Called by Procomm Plus to initiate a file replay. Procomm Plus passes in the name of the target file. On EOF, it sends a **WM_COMMAND:ID_PLAYBACK:0** message to **pw_dlt->hWndMain**, which causes Procomm Plus to close the file. For more information, see the code example.

When the screen is at a point when it could reasonably be paused, such as when the end of the screen has been reached, the DLT should send a **WM_COMMAND:WM_USER+10:0** message to **pw_dlt->hWndMain** to cause the replay to pause. No code example is given for this action.

EndReplay (NULL)

Terminates file replay. For more information, see the code example..

PauseReplay (NULL)

Pause display of the replay file. For more information, see the code example.

ContinueReplay (NULL)

Called by Procomm Plus when the user has pressed ^Q to continue a file redisplay. For more information, see the code example.

TerminalWriteBuffer (lpTerminalWriteBufferStruct)

The struct contains an LPSTR, a row/column position on the screen and a flag specifying whether to use the given row/column position or to use the current position. The string will be inserted using the current state of the emulation. The current position will be left at the end of the string that was written.

There is no explicit mechanism defined for synchronizing the action of a **TerminalWriteBuffer** and the current state of the standard input stream. Thus, the sequence that will cause the change to the row/column position and then the actual string to be written can literally be inserted into the input stream at any point, even in the middle of an incoming escape sequence.

RequestNumOfEmuls (lpRequestNumOfEmulsStruct)

Called by Procomm Plus to determine the number of emulations supported by the DLT; the DLT should return this number in the struct. Procomm Plus then calls the DLT once for each Emulation, requesting its name, needs and capabilities. For more information, see the code example.

⇒ ***Note:*** *Procomm Plus does not pass* ***lppw_dlt*** *to this function.*

RequestEmulationName (lpRequestEmulationNameStruct)

Called by Procomm Plus to retrieve the name of an Emulation supported by the .DLT. The struct will contain the number of the emulation for which the name is being queried, a pointer to a buffer to receive the name and a maximum length of the name. The DLL will copy the name into the buffer. This name will be the default name for this emulation; it is possible that it will be changed by the user. For more information, see the code example.

⇒ ***Note:*** *Procomm Plus does not pass* ***lppw_dlt*** *to this function.*

RequestEmulationNeeds (lpRequestEmulationNeedsStruct)

Called by Procomm Plus to determine the needs of an emulation supported by the .DLT. The specified struct will contain the number of the emulation for which the needs are being queried, and a place to put the needs.

The needs are a bit flag array defining special items of interest to Procomm Plus and the DLT. Currently, there is a flag to enable the Timer call, which runs even when the emulation is shutdown, and a flag indicating that the DLL needs to store some special information, which will be done via a dialog, called the Advanced Setup Dialog (see below). For more information, see the code example.

⇒ ***Note:*** *Procomm Plus does not pass* ***lppw_dlt*** *to this function.*

RequestEmulationCaps (lpRequestEmulationCapsStruct)

This call provides a way for Procomm Plus to determine the emulation's capabilities. For example, currently implemented is a request to get a set of flags that determine which menu items in the Procomm Plus menu should be grayed. This is needed because some items such as the scrollback buffer must be grayed for DLT emulations without a scrollback buffer. For more information, see the code example.

⇒ ***Note:*** *Procomm Plus does not pass **lppw_dlt** to this function.*

DoAdvancedSetupDialog (lpDoAdvancedSetupDialogStruct)

Called by Procomm Plus when the user selects "Advanced Setup" for an emulation if the DLT has indicated via **RequestEmulationNeeds()** that it requires additional information from the user. The DLT should display a dialog box and retrieve the required information. Procomm Plus will provide a place to keep the information and a parent window handle for the dialog; the information will always be available through the PW_DLT structure. This is the preferred interface for handling special DLT specific data, such as the "unhide" and "no_attributes" data required for the BTX emulation. It also is the preferred place to keep the bitmap vs. text flags for **SnapShot_File()** and **SnapShot_Clipboard()** . For more information, see the code example.

⇒ ***Note:*** *Procomm Plus does not pass **lppw_dlt** to this function.*

ComputeScrollRange (LPCOMPUTESCROLLRANGESTRUCT)

Called by Procomm Plus when the terminal window size is changed. It queries whether a scrollbar is needed and if so, what its range would be. Please see the code example in SIMPLDLT.C function **ComputeScrollRange()** for more details.

SetupFinalSize (NULL)

Called by Procomm Plus when the terminal window size has been changed. It allows the DLT to set its internal state for the new window size. This allows the DLT to avoid handling the **WM_SIZE** message, which can be difficult when handling scrollbars. For more information, see the code example.

SetupCaret NULL)

Called when Procomm Plus's main window receives the focus. This allows the terminal to set the caret in its window. For more information, see the code example.

RemoveCaret (NULL)

Called when Procomm Plus's main window loses the focus. This allows the terminal to remove the caret. For more information, see the code example.

QueryEscape (lpEscapeStruct)

Provides an interface for special cases. EscapeStruct contains the fields:

* **nAction,** which specifies which action is being queried.

  The available values for **nAction** are:

  1 - Requests that **nReturnedValue** be filled with the number of Escape options which the DLT supports for the given **nEmulation**. For example, if **PerformEscape()** for **nEmulation** handles the "hidden" and "color" attributes, **nReturnedValue** would be set to 2.

  N - Requests that **nReturnedValue** be set to **1** if the DLT function **PerformEscape()** has meaning for the **nAction** specified by **N**; **0** otherwise. For example, if the emulation is VTX, and VTX supports **nAction** 1 ("hidden" attribute), then **nReturnedValue** would be set to **1** before returning.

* **nEmulation**, which contains the internal DLT number of the emulation being queried.

* **nReturnedValue**, which contains the value returned by **QueryEscape()**.

```
typedef struct tagEscapeStruct
{
        int     wSize;                  // Size of this struct
        int     nEmulation;             // Which DLT emulation is being accessed
        int     nAction;                // Which action to take
        int     nReturnedValue;         // Return value
}
typedef EscapeStruct *lpEscapeStruct ;
```

QueryRowColInfo (lpQueryRowColInfoStruct)

⇒ *This function is available starting with PROCOMM PLIS version 3.xx.*

PerformEscape (lpEscapeStruct)

Provides an interface for special cases. **EscapeStruct** is the same struct as is used in **QueryEscape()**. It contains the fields:

* **nAction**, which specifies the action to take. The currently defined values for **nAction** are:

  0 - Reserved.

  1 - Toggles the VTX “hidden” attribute between “hidden” and “accept”. The default after **Reset()** is “accept”.

  2 - Toggles the VTX “color” attribute between “ignore” and “accept”. The default after **Reset()** is “accept”.

* **nEmulation**, which contains the internal DLL number of the emulation being requested.

* **nReturnedValue**, which holds the return value.

  Generally, if the requested action is successful, **PerformEscape()** should return **1** in **nReturnedValue;** 0 otherwise. If **nEmulation** is not the number of the currently-running emulation, **PerformEscape()** has no meaning and it should return 0 in **nReturnedValue**.

QueryString (lpQueryStringStruct)

Provides another interface for special cases. QueryStringStruct contains an **nItem** which defines what string is desired by Procomm Plus, and a pointer to a buffer into which to place the string. The strings are typically ones that Procomm Plus needs to perform some action on behalf of the DLT. For example, when opening a screen snapshot file, Procomm Plus needs to know what filetype to use, which is only known by the DLT. Procomm Plus uses this function to discover the filetype required.

The currently defined values for “nItem” are:

0 - Reserved.

1 - Return the file extension which should be used for a **SnapShot_File**. Typically, this would be TXT for a text format snapshot, or BMP for a bitmap format snapshot. The text should be the file extension without a period, and it should be in uppercase.

TerminalChanged (NULL)

Called by Procomm Plus whenever something in the TERMINALSET structure, or some data in certain other structures, has changed. This call gives the DLT the opportunity to detect the change and respond to it. For example, if the user changes the number of rows in Terminal Setup, **TerminalChanged()** will be called, providing an opportunity for the DLT to change the number of rows it is displaying.

MenuNotification (lpMenuNotificationStruct)

Called when a **WM_COMMAND** message is received by Procomm Plus. The DLT can process the message and return **1**, or not process the message and return **0**.

Many of the messages received will be defined by the **ID_* #defines** in **menudlx.h**. Others will not. It is very important that the DLT not attempt to handle any received messages that are not defined by **ID_* #defines.**

Messages

There are two windows of interest to emulations:

- The terminal window, which is guaranteed always to be an integral size of the font in height and width and only has to be responsible for painting its own character cells as needed.
- The main window, which generally has the focus unless some other window is popped up over it

  The main window gets keystroke messages, and the mappings of function keys can be handled by setting specific keyboard maps that go along with the emulation. The regular character keys (those that produce **WM_CHAR** messages) are sent to the DLT via the **KeyChar()** interface call.

Record Mode

In order for the *Record Mode* of Procomm Plus to work well, it relies upon an emulation DLT to call it whenever a significant cursor movement event occurs. Typically, these are when a tab, form feed, vertical tab, CR, LF or absolute cursor positioning occurs.

void PASCAL **P_RecordModeChange** (void)

P_RecordModeChange() should be called when a CR or LF or TAB or absolute cursor address is seen by the main polling loop and ***(lppw_dlt->lphRecordFile)!=-1**. Calling **P_RecordModeChange()** signals that the host is potentially about to prompt the user for some input. If in fact the user types something after a **P_RecordModeChange()**, a waitfor/transmit pair will be generated in the script that is being recorded.

Overspray

There is a small area of color surrounding the terminal window before the background of the display window is displayed. This area is call the “overspray” area. This is provided to make the terminal window display appear less crowded. Without it, the characters of the terminal window jam up against the display window background, causing an unpleasant visual effect.

The color of the overspray region can be set from the DLT by calling **P_SetOverSprayColor()** and passing in a **COLORREF**. Procomm Plus uses solid colors for both text and background, and it is suggested that the DLT do the same for consistency. In order to do so, the **COLORREF** passed to **P_SetOverSprayColor()** should be the return value from the Windows API call **GetNearestColor()**. This will ensure that the requested color can be displayed without dithering. See the **Reset()** function in the sample code file **simpldlt.c** for further details.

void PASCAL **P_SetOverSprayColor** (COLORREF)

The "overspray area" is a small colored border area just inside the terminal window, surrounding the window where terminal data is displayed. The overspray area makes the terminal window display appear less crowded. Without it, the characters of the terminal window jam up against the display window background, causing an unpleasant visual effect. A DL*x* can set the color of the overspray region by calling **P_SetOverSprayColor()** and passing in a *COLORREF value*. Procomm Plus uses solid colors for its text and background; we suggest that a DL*x* do the same.

Argument:

COLORREF. The ***COLORREF*** passed to **P_SetOverSprayColor()** should be the return value from the Windows API call **GetNearestColor()**. This will ensure that the requested color can be displayed without dithering. Please see the **Reset()** function in the **simpldlt.c** sample code file for further details.

Mouse Interface

The mouse interface of a DLT should follow that of Procomm Plus. This consists of handling left-button double-clicks, click-and-drag actions with the left button and click actions with the right mouse button.

Left-button Double Clicks

Left-button double-clicks signify an instantaneous selection and sending process. Whether a single character or a word is to be selected is discovered by calling **P_GetLButtonDClickAction()**. The selected data are then sent back to Procomm Plus to be shipped out the COM port or placed in the File Clipboard by calling **P_SetMouseString()**.

⇒ ***Note: P_GetLButtonDClickAction()*** *and* ***P_SetMouseString()*** *are explained later in this document.*

Left-button Click and Drag Actions

Click-and-drag actions with the left mouse button select text on the terminal screen. The emulation should stop calling **P_GRXGet()** during a click and drag action; continued calls to **P_GRXGet()** would imply that the selected text could change even as the user selects it.

At the end of the click-and-drag action, signified by a left button up message, the emulation must determine what to do with the selected data. The action can be predefined in the

Advanced Setup for the DLT emulation, but it is preferable to present to the user a pop-up menu of available options. This is done with the Windows API **TrackPopupMenu()**.

Typical operations available to the user at the end of a click and drag action would be copying data to the COM port, copying data to the Windows Clipboard, or printing the data.

If the data is copied to the COM port or to the Windows Clipboard, the line-ends must be handled. This is done by appending the string returned from **P_GetSelectionEOLString() to** each line of data. The resultant string is then either shipped to the COM port using **P_AddToTXBuf(),** or placed into the Windows Clipboard either directly or by using **P_SetMouseString()**.

If the data is to be printed the current printer setup is available through the **P_ReadNamedStruct()** interface.

⇒ *Note: **P_GetSelectionEOLString()**is explained later in this document.*

Right-button Clicks

Right mouse clicks can pop up a menu, using the Windows API **TrackPopupMenu(),** and display the menu items that the user has specified in the *Setup, Options, RightMouse* dialog. These settings are available through **P_ReadNamedStruct()** calls.

A right mouse double-click can also send a string of cursor sequences out the port, representing the actions that a user would take in positioning the emulation's cursor at the site of the double-click using the cursor keys. This feature is convenient if the user is working in an on-line editor, as double-clicking can position the cursor in the character mode interface more rapidly than would be possible with simple keystrokes. The assembled cursor sequence should be sent out the port using **P_AddToTXBuf()**.

LPSTR PASCAL **P_GetSelectionEOLString** (void)

The returned value is a null-terminated string containing the characters specified in the ***End of line character(s)*** field in *Setup, Options, Left Mouse Button Options*. By default, this field's contents are equivalent to an Enter keypress. However, an editable field is provided for the user's convenience.

The string should be copied locally and used immediately by the DLT, as it is subject to immediate replacement. Do not write to the buffer, as this will compromise the integrity of Procomm Plus's internal data structures.

---

int PASCAL **P_GetLButtonDClickAction** (void)

The results are:

1 Send a character.

2 Send a word.

3 Send a character followed by CR.

4 Send a word followed by CR.

If the cursor is on whitespace, then the left mouse button double click does nothing.

⇒ ***Note:*** *Whitespace is defined as a blank, a graphic character, or a screen boundary. A word is defined as a contiguous sequence of non-whitespace characters surrounded by whitespace.*

Note that passing a character or word into **P_SetMouseString()** with type 1 will cause the final CR to be appended if needed. In other words, use **P_GetLButtonDClickAction()** only to determine whether a character or a word should be sent. Procomm Plus adds the terminating CR if it is appropriate.

---

int PASCAL **P_SetMouseString** (int type, LPSTR lpstr, int len)

Called with data from a mouse action.

Arguments:

int *type*. The type of mouse action which prompted the data. Values are:

0 The data is the result of a left-double-click. The data is assembled and passed to **P_SetMouseString()**. Procomm Plus adds a terminating CR if one is appropriate.

1 The data is the result of a selection. Currently only copying to the Windows Clipboard is supported. The DLT should assemble the data into a buffer, separating lines with the string returned from **P_GetSelectionEOLString().** It should then call **P_SetMouseString().**

2 The data is the result of a right double-click. Currently this sends the data out the COM port. The DLT can do the same by calling **P_AddToTXBuf()**.

Returns:

int *nRetVal.* True if the function succeeded, False otherwise. If the function failed there is no appropriate remedy, so just ignore it.

The Keyboard

Procomm Plus divides the keyboard into four regions. Each of the regions is independently set, but all are controllable. The regions are:

- **Metakeys**, which are all ALT + number keys at the top of the keyboard combinations. For example, ALT+1, ALT+SHIFT+1, ALT+CTRL+1, and ALT+SHIFT+CTRL+1 are all metakeys.

  Metakeys are defined in a metakey file (extension **.key**), which can be created using the *Metakey Editor* in Procomm Plus.

- **Accelerators**, which are all non-regular combinations of alphabetic keys. For example, ALT+a, ALT+SHIFT+a, ALT+CTRL+a, ALT+SHIFT+CTRL+a, CTRL+a, SHIFT+CTRL+a, and all ALT combinations of all keys on the keyboard are accelerators.

  Accelerators are created by **pwmkacc.c.** The programmer should edit **pwmkacc.c** to create the needed definitions, then compile and execute it to create a file **xxxxxx.acc.** The **.acc** file is included with the DLT and loaded at DLT-emulation start-up time by specifying its name in the **DLT_StartupStruct** member **szKeyboardAcceleratorFileName**. This member is returned by the DLT code in response to a **DLT_Startup** call. The **DLT_StartupStruc**t is passed as a parameter to this call, and is documented in **xdlt.h**.

- **Function Keys**, which are all combinations (except ALT) of Function Keys, White Keypad keys, and Gray (extended) keys;

  Function Keys are defined by a **.kbd** file, which can be created with the Procomm Plus *Keyboard Editor* and shipped with the DLT.

  Kbd files can also be created by modifying **makek101.c** and **makek84.c,** which are then compiled and run to create **xxxxxx.101** and **xxxxxx.84** files. The **.101** and **.84** files are then included as resources in the compiled **.dlt**.

  Regardless of the method used to create it, the keyboard resource is loaded at DLT-emulation start-up time.

- **WM_CHAR** keys, which are the normal alphabetic keys and their SHIFTed keys.

  The **WM_CHAR** keys are passed from Procomm Plus to the DLT through the **KeyChar()** interface, defined above.

  The keyboard mappings are defined in the C code files **makek101.c** and **makek84.c**, which write out a binary definition of the keyboards. You will need to study the structure of the **makek*.c** files and modify their contents to suit your needs.

  The binary files produced by compiling and running the **makek*.c** files are included into the DLL's resource with statements like the following:

  ```
  1            KBD101NEW            firstkbd.101
  ```

2	**KBD101NEW**	**secndkbd.101**
1	**KBD84NEW**	**firstkbd.84**
2	**KBD84NEW**	**secndkbd.84**

These statements allow the default 101-key and 84-key keyboard maps for each defined emulation to be retrieved from the DLT. If one is modified by the user, it will be written out and used transparently. If these resources are ***not*** included in the DLT, the results are undefined.

When metakeys, function keys, and accelerators are mapped as send text or send text with echo, their key values are not passed into the DLT. Instead, the data that these keys represent is pumped into the outgoing data stream. Text sent with echo, echoes back to the DLT through the incoming data buffer, which is retrieved using **P_GRXGet**.

If, by any chance, you need the DLT to trap data that's sent out when the user presses a mapped key, implement the **SpecialKeys()** interface, defined above. When **SpecialKeys()** is implemented, the data from the mapped key is passed into the DLT in a string. The DLT can then pump the string into the outgoing buffer using **P_AddToTxBuf()**; or, through the DLT's internal keyboard processor, by calling its own **KeyChar()** function.

IMPORTANT NOTE: If Procomm Plus is set to half duplex (local echo), characters from mapped keys won't only be pumped into the DLT using **SpecialKeys()**! In addition to the DLT getting data in half duplex mode, PW also pumps the data into the local echo queue. This means that the echoed data will show up in the incoming buffer and get retrieved when you make a call to **P_GRXGet()**. The second parameter of **P_GRXGet()**, lpSrc, contains the source of each character that's returned in the first parameter, lpChar. Simply check each character to see if it's coming in from the local echo queue and handle it appropriately.

GRX functions

Recognizing terminal escape sequences in the incoming data stream is a crucial task for a terminal emulation. When a terminal escape sequence is found, the emulation must extract it from the incoming data stream, and call the appropriate terminal control routine. Typically, an input stream parser will be looking for several different string patterns simultaneously. The "**GRX**" functions are designed to access the Procomm Plus input stream parsing engine, and to take into consideration the various Procomm Plus requirements for scrollback buffering, capture file or print logging, character translation, and *Record Mode* activities.

The "**GRX**" functions are described in ***Procomm Plus® Interface Specification Overview and***
Shared Callback Functions, but there are a couple issues particular to DLTs worth mentioning.

The **P_GRXGet()** API, in general, can be called with any value count, however during file playback and RecordMode, it should be called with a count of **1**. This is because of internal buffering logic which must assume that once the terminal has read a character, that character has been processed. This logic allows **P_GRXGet()** to handle the **P_RecordModeChar()** call, and to properly shut down playback files. The DLT can always know if RecordMode is active because ***(lppw_dlt->lphRecordFile) != -1;** for more information, see the discussion of **P_RecordModeChange()**. The DLT can always know if a file replay is in effect because the **StartReplay()** and **EndReplay()** functions are called upon starting and ending the replay file, respectively. The easiest way to handle the RecordMode and file replay situations is to always call **P_GRXGet()** with a count of **1**. This is generally acceptable from a performance standpoint because **P_GRXGet()** is relatively fast.

The **P_GatherAndMatchInputStream()** API only buffers 1K of data, so if you expect a data stream of 10K per second, you should ensure that it is called at least 10 times per second. The Procomm Plus timer is set to 55ms, so the DLT **Timer()** entry should be called 18 times per second. If there are many activities taking place within the program or on the machine as a whole, however, the **Timer()** entry to the DLT might only be called a few times per second. In such a situation, call **P_GatherAndMatchInputStream()**, process all of its characters and repeat. The loop should make sure that it eventually yields to avoid locking Windows, but it should repeat enough times to ensure that the input stream is being properly serviced. One way to do this is to run the loop until either there are no more input characters or until a certain amount of time has passed (perhaps 3 clock ticks). Clock ticks (in terms of milliseconds) are easily and quickly obtained by calling **GetTickCount()**.

Structures

The interface structures are defined in the accompanying code, particularly in the files **cbackdlx.h** and **xdlt.h**. These structure definitions are subject to change by DATASTORM and by implementors as needed. With the release of PROCOMM PLUS 2.0, however, any changes made must be done solely by addition to the end of the existing structures.

Attached Code Files

All of the conceptual details have been covered in this document, but many technical details are further explained in the listings of the code files that are attached. For example, the header files **xdlt.h** and **cbackdlx.h** are included in both Procomm Plus and the DLT and define the structures and entry points which are shared between them. The file **simpldlt.c** contains a simple but functional emulation that can be used as the basis for the DLT portion of the Procomm Plus-DLT interface.

Index of Functions