

**Procomm Plus® Interface Specification Overview and  
Shared Callback Functions**

# Table of Contents

<b>INTRODUCTION</b>	<b>3</b>
<b>DLL EXTENSIONS</b>	<b>3</b>
<b>DLX INTERFACE OVERVIEW</b>	<b>3</b>
<b>DATA STRUCTURES</b>	<b>3</b>
<b>DLX ENTRY POINTS</b>	<b>4</b>
<b>CALLBACK FUNCTIONS</b>	<b>4</b>
<b>Callback Functions Based On WIN 3.x SDK And Standard C Library Functions</b>	<b>5</b>
<b>General Purpose Callback Functions</b>	<b>6</b>
<b>GRX functions</b>	<b>8</b>
<b>INDEX OF FUNCTIONS</b>	<b>13</b>

## Introduction

Although Procomm Plus is the most full-featured Windows communications program available today, there may be certain situations where a technologically-sophisticated user needs additional capabilities that are not provided in the standard product. Procomm Plus was developed to be easily extended through the addition of user-written DLL's. In fact, some of the features of the standard product were developed using the DLL procedures described in this document.

## DLL Extensions

Procomm Plus recognizes three distinct types of DLL extensions; collectively, these are referred to as DLx's. DLx's are used to extend Procomm Plus's capabilities in three areas:

- ◆ **File Transfer Protocols** can be added to Procomm Plus using Dynamically Linked Protocols (DLP's). One example is the INDFILE.DLP, which provides the IND\$FILE protocol within Procomm Plus 2.xx, 3 and 4.xx.
- ◆ **Terminal Emulations** can be added using Dynamically Linked Terminals (DLT's). The RIPscrip 1.54 emulation is a DLT, and the Minitel, Prestel and BTX terminals shipped with the various Procomm Plus 2 and 3 International versions are all .DLT's.
- ◆ **Connection support** can be added using Dynamically Linked Connections (DLC's), which allow redirection of the Windows Comm calls. Examples of these files include the PWTELNET.DLC and PW3NCSI.DLC, which redirects the communications calls to a network communications server.

When Procomm Plus is first executed, it searches its installation directory for any files with .DLP, .DLT or .DLC filename extensions. If any are found, they are loaded, initialized and added to Procomm Plus's list of supported terminal emulations, protocols, or connection types.

## Dlx Interface Overview

Procomm Plus maintains several data structures which are important to developers of DLP's, DLC's, and DLT's, which are collectively known as DLx's. These data structures, in conjunction with callback functions into Procomm Plus and the DLx entry points, provide partial extensibility of Procomm Plus.

## Data Structures

The data structures required for interfacing DLx's and Procomm Plus are defined in the #include files which accompany any specific DLx specification. These structure definitions are subject to change with future versions of Procomm Plus. If it becomes necessary for a developer to change any of these structures, new fields should be appended to the end of the existing structures, rather than inserted into the middle. This will insure maximum compatibility with future versions of Procomm Plus.

In general, any DLx must be capable of handling multiple instances of Procomm Plus. This means that all data that is specific to an instance of the caller of the DLx must be maintained as instance data. Typically, this is done by extending the currently-available mechanisms provided by the DLx interfaces.

Each call to a DLx from an instance of Procomm Plus includes a pointer into a data structure maintained by that particular instance. An area within the data structure is made available to the DLx, allowing it to maintain its own pointer to an allocated data structure for any purpose.

⇒ **Note:** *The point is that each DLx is responsible for maintaining a separate state for each instance of a calling application, and that the calling application will make some of its local data available to the DLx in order to help it do so.*

For example, in the DLT interface the main data structure is called a **PW\_DLT**. It contains a variable, **iDLLMagicIndex**, which the DLT is free to use as it needs. Typically, the DLT would maintain a static table of pointers to instance data and use **iDLLMagicIndex** to specify which pointer to use.

A simpler interface is provided for a DLP. The **PROTOSSET** structure contains a byte field called **temp[66]**, which can be used in any way by the DLP. In particular, the DLP can map a structure onto this data which contains all of the DLP's instance data. If the instance data is longer than 66 bytes, the first 4 bytes can be interpreted as a pointer to an allocated instance data structure.

The callback function, **P\_ReadNamedStruct()**, allows a DLx to access Procomm Plus's internal state structures, which are stored in the parameter (PW4.PRM) file. The access is fairly complicated, and is best understood by looking at **ddirdlx.h**, which defines the structures returned by **P\_ReadNamedStruct()**, and at the **SET** structures in **pwdlx.h**, which are included in the definitions of the named table structures from **ddirdlx.h**.

## DLx Entry Points

When Procomm Plus first begins executing, it performs a search in its installation directory for all files with a suffix of **.dlp**, **.dlc**, or **.dlt**. Each DLx file type has at least one entry point which can be called by Procomm Plus. For example, the DLT files have a single entry point which is used as a function dispatcher. Procomm Plus calls the DLT entry point with a function code requesting the DLT to perform some action. DLP files on the other hand have four entry points: different places in Procomm Plus code call the appropriate DLP entry point to execute a particular function.

⇒ **Note:** *Please refer to the appropriate DLx documentation for specific details concerning their entry points.*

## Callback Functions

All of Procomm Plus's callback functions are available as exports. Some are also available through callback pointers in the **PW\_CALLBACK\_STRUCT** structure, which is defined in **cbackdlx.h**. This file also contains typedefs which **greatly** simplify calling these functions through the callback structure.

However, callback functions *should* be called directly, just as one makes a call into other Windows DLLs. This is done by linking with **pw.lib**, then simply calling the functions; prototypes are available by including **cbackdlx.h**.

Some of the code contains examples of calls indirected through the **PW\_CALLBACK\_STRUCT** structure, but these are obsolete. Direct calls are both faster and easier to code.

### **Callback Functions Based On WIN 3.x SDK And Standard C Library Functions**

Almost all of the Comm functions from the Windows SDK have been duplicated in Procomm Plus, and have been exported to make them accessible by DLx's. The names of the exported SDK functions are created by prefixing the SDK function name with P\_ and appending a W. Thus, **ReadComm** becomes **P\_ReadCommW**, and **P\_ReadCommW** takes the same arguments and returns the same return code as **ReadComm**. There are a few exceptions documented in **cbackdlx.h**. The following functions are based on their SDK counterpart:

```
int PASCAL P_ReadCommW (COMM_HANDLE nCid, LPSTR lpBuf, int nSize);
```

```
int PASCAL P_WriteCommW (COMM_HANDLE nCid, LPSTR lpBuf, int nSize);
```

```
int PASCAL P_FlushCommW (COMM_HANDLE nCid, int nQueue);
```

```
int PASCAL P_SetCommBreakW (COMM_HANDLE nCid);
```

```
int PASCAL P_ClearCommBreakW (COMM_HANDLE nCid);
```

```
int PASCAL P_GetCommStateW (COMM_HANDLE nCid, DCB *lpDCB);
```

```
int PASCAL P_SetCommStateW (DCB *lpDCB);
```

```
int PASCAL P_EscapeCommFunctionW (COMM_HANDLE nCid, int nFunc);
```

```
int PASCAL P_GetCommErrorW (COMM_HANDLE nCid);
```

⇒ **Note:** *The SDK function has status. That is, this function does not follow the SDK.*

```
int PASCAL P_GetCommErrorX (COMM_HANDLE nCid, COMSTAT *lpstat);
```

⇒ **Note:** *This function exactly corresponds to the SDK's **GetCommError ()** function.*

## General Purpose Callback Functions

---

BOOL PASCAL **P\_AddToTXBuf** (LPSTR *string*, int *cnt*, unsigned char *do\_pause*, unsigned char *do\_record*)

**P\_AddToTXBuf ()** is generally used to transmit characters out the port. It handles character pacing if necessary, pausing if necessary and any needed indirection by calling **WriteCommW ()**.

Typically, emulation code would call **P\_AddToTXBuf ()** with **do\_pause** FALSE. **do\_record** should be TRUE when sending the user's keystrokes, but FALSE if the emulation is responding to a host command for information.

### Arguments:

**string.** A pointer to the string to be added.

**cnt.** The count of characters in the string.

**do\_pause.** True if every pause character in the string (available via a pointer) should be converted into a 1/2 second pause before shipping the next character out the port.

**do\_record.** True if the string should be considered user output.

### Returns:

TRUE on success, FALSE on fail (if there is insufficient room in the transmit buffer). If FALSE is returned, then NONE of the characters in string were buffered, and the entire string should be sent again at a later time.

---

void PASCAL **P\_DisplayStatusMessage** (LPSTR *msg*, BOOL *BlinkAndBeep*)

This function is very similar to **P\_StatusLineDirect ()**, except it allows a means of beeping and blinking the status message.

⇒ **Note:** If you're setting *BlinkAndBeep* FALSE, it would be more efficient to use **P\_StatusLineDirect ()** instead.

### Arguments:

**msg.** A ptr to a string to be displayed on the Procomm Plus Status Line, or NULL to clear the Status Line

**BlinkAndBeep.** TRUE to beep and cause the message to blink, FALSE to display the message normally.

---

LPVOID PASCAL **P\_ReadNamedStruct** (int *Table*, int *Index*)

**P\_ReadNamedStruct ()** provides access to Procomm Plus's internal state structures which are stored in the parameter file. The access is fairly complicated, and is best understood by looking at the included files **ddirdlx.h**, which defines the structures returned by **P\_ReadNamedStruct()**, and at the **\*SET** structures in **pwdlx.h**, which are included in the definitions of the named table structures from **ddirdlx.h**.

The values should be treated as read-only, as Procomm Plus could become internally inconsistent if any values were changed without going through the standard interface (which is Setup). If you **do** need to change the values, all entries can be accessed through Aspect, which properly type checks and maintains internal consistency.

**Arguments:**

**Table.** A **#define** from the top of **ddirdlx.h**, which specifies which Named Table (that is, which structure in **ddirdlx.h**), will be accessed.

**Index.** Specifies which entry in the given Named Table to access. A (-1) means to retrieve the entry currently in use. For example, to retrieve the Named Table of the ANSIBBS emulation, the index would be 4 (look in the popup list of emulations in Setup:Terminal), but if you want the Current emulation, regardless of which emulation is in use, then pass in a (-1).

**Returns:**

A pointer to the requested structure on success; otherwise, a NULL pointer. In general, the pointer must be explicitly cast to the type of pointer to DDStruct, where DDStruct is one of the DD\*\*\*\* structures defined in **ddirdlx.h**.

⇒ **Note:** To iterate through all named tables, start with an index of 0, and loop with index++ until the return value is NULL.

---

void PASCAL **P\_StatusLineDirect** (LPSTR *msg*)

When called, this function causes Procomm Plus to display the **msg** argument on the Procomm Plus Status Line. If a message needs to be displayed to the user, use the Procomm Plus Status Line. Status Line messages are not queued; only one message can be displayed at a time.

⇒ **Note:** Most Procomm Plus strings are defined in a resource file instead of a **.c** or **.h** file. This is to facilitate translating Procomm Plus into different languages. If you use status messages defined in an **.RC** file, you must do a **LoadString ()** before calling **P\_StatusLineDirect ()**.

⇒ **Warning!** Do not use Message Boxes, please! They cause problems with reentrancy because Procomm Plus continues to run off the timer while the Message Box is up!

**Argument:**

**msg.** A ptr to a string to be displayed on the Procomm Plus Status Line, or NULL to clear the Status Line

**GRX functions**

Recognizing specific character sequences in the incoming data stream is often a necessary requirement when implementing DLx's, especially terminal emulations. The "GRX" functions, described below, are designed to access the Procomm Plus input stream parsing engine, and to take into consideration the various Procomm Plus requirements for DLx activities.

---

```
int PASCAL P_GRXRegisterString(LPSTR lpString, int Count, BYTE bFlags,
    DWORD dwTime, LPFN_GRXCALLBACK lpfn)
```

This function registers a string to be used for pattern matching against the input stream. If the pattern is matched or a timeout time is reached, the callback function specified will be called within the context of Procomm Plus.

**Arguments:**

**LPSTR lpString.** Points to a constant string to be searched for in the input stream.

**int Count.** The number of characters in the **lpString** target pattern.

**BYTE bFlags.** A set of bitflags defining how the string is to be matched and what will happen after it is matched. The bitflag values are:

**GRX\_CASE ( 0x01 )** The string matching algorithm is case sensitive.

**GRX\_FLUSH ( 0x02 )** Once matched, the matched string will be put into the **GRX** buffer, so that it will be seen by **P\_GRXGet()**. If this flag is not set, then the string, once matched, will **not** be placed in the **GRX** buffer. In other words, the string will effectively be stripped from the input stream.

**GRX\_IMMEDIATE ( 0x04 )** The callback function is called immediately upon matching. At that time, there may be characters still in the **GRX** buffer. If this flag is not set, the callback function will not be called until the **GRX** buffer has been

emptied. In essence, the **gatherer** stops processing and waits until its queue is empty before calling the callback function.

**GRX\_TIMEOUTONLY ( 0x08 )** The callback function will be called only after the timeout specified in the **DWORD dwTime**. In other words, even if the **lpString** is matched the callback function will not be called, but the timer is reset if **lpString** is matched.

**GRX\_XLATE ( 0x10 )** The input streams will be stripped via the current emulation's "**Strip 8th Bit**" flag, and the result run through the input translate table before being tested for matching.

**DWORD dwTime**. If **dwTime** is zero, an infinite wait is implied. If not zero, **dwTime** represents a timeout value in milliseconds, which is the amount of time to wait before calling the callback function. If a match is found before the time specified, the callback function is also called. If **bFlags & GRX\_TIMEOUTONLY (0x08)**, then the callback function is called **only** after the timeout value. Thus, it would be a logical error to have **bFlags & GRX\_TIMEOUTONLY (0x08)** and **dwTime = 0**.

**LPFN\_GRXCALLBACK lpfn**. A pointer to the callback function. The callback function is **pascal**, takes an **int**, an **LPSTR**, and two more **ints**, and returns an **int**. It must be callable from the Procomm Plus context, so it should probably be exported and in a DLL. Its return value is either 1, to leave the string registration in place, or 0, to automatically unregister the string. Its arguments are:

**int**, which is the handle of the registered string.

**LPSTR**, which points to the matched string.

**int**, which is the length of the matched string.

**STRDB\_MATCHED ( 1 )** if the string matched or **STRDB\_NOMATCH (-3)** if it did not match (for example, if the function is being called because of a timeout).

**Returns:**

**int handle** to the registered string. This value is used to reference the registration of the string in any subsequent calls to the **GRX** subsystem. A handle equal to -1 indicates an error occurred when attempting to register the string.

---

```
int PASCAL P_GRXRegisterSpecialString (LPSTR lpString, int Count,
    LPSTR lpSpecial, BYTE bFlags, DWORD dwTime, LPFN_GRXCALLBACK lpfn)
```

This function should be used if you want to match all strings of the form **a?b** where **?** represents a single character. More complex pattern matching schemes such as regular expressions are not supported.

**Arguments:**

These arguments are the same as those for **P\_GRXRegisterString()** except for the **LPSTR *lpSpecial*** argument, which points to a string of the same length as the ***lpString***, and which has a byte corresponding to each byte in the ***lpString***. Each of the bytes in the ***lpSpecial*** has its 0th bit set, or not set. If the 0th bit is 1 then the pattern matcher does not care what is in that position in the ***lpString***. If the 0th bit is 0, then the pattern matcher does care what is in that position.

**Returns:**

**int *handle*** to the registered string. This value is used to reference the registration of the string in any subsequent calls to the **GRX** subsystem. A handle equal to -1 indicates an error occurred when attempting to register the string.

---

```
void PASCAL P_GRXUnregisterString (int handle)
```

**P\_GRXUnregisterString()** must be called when the caller no longer needs the string. Failure to do so will eventually lead to a performance degradation in Procomm Plus itself.

**Arguments:**

**int *handle*** is the value returned from **P\_GRXRegisterString()** or **P\_GRXRegisterSpecialString()**.

---

```
void PASCAL P_GRXEnableString (int handle, BOOL bEnable)
```

**Arguments:**

**int *handle*** is the value returned from **P\_GRXRegisterString()** or **P\_GRXRegisterSpecialString()**. If **BOOL *bEnable*** is TRUE, matching on the string will be enabled; if FALSE, it will be disabled.

After the initial call to **P\_GRXRegisterString()** or **P\_GRXRegisterSpecialString()**, the matching is ON by default. To turn it off you must explicitly call **P\_GRXEnableString( *handle*, FALSE)**. The string may be unregistered by **P\_GRXUnregisterString()** whether it is enabled or not.

---

BOOL PASCAL **P\_GRXQueueEmpty** (void)

**Returns:**

TRUE if the **GRX** buffer is empty.

⇒ **Warning:** *It may happen that there is data waiting in the input Windows Comm buffer even though this function returns TRUE. It is important to periodically call **P\_GatherAndMatchInputStream()** so that the input Comm buffer can be read and processed. Only after such a call is the return value from **P\_GRXQueueEmpty()** valid.*

---

int PASCAL **P\_GRXGet** (LPSTR *lpChar*, LPSTR *lpSrc*, int *count*)

**P\_GRXGet()** should be called to read characters from the input stream parser. Although **P\_ReadCommW()** will read characters directly from the COM port, **P\_GRXGet()** should be used or Procomm Plus will not be able to parse the read characters - this can cause unpredictable and probably unpleasant results.

**Arguments:**

**lpChar.** A pointer to a buffer to receive the characters.

**lpSrc.** A pointer to a buffer to receive the source of characters, or NULL if you do not need that information. The **lpSrc** values will parallel the values in **lpChar**; COM port is **COM\_PORT\_STREAM** (0x1), Local Echo is **LOCAL\_ECHO\_STREAM** (0x2) and File Replay is **PLAYBACK\_STREAM** (0x4)

**count.** The maximum number of characters that should be received. This is the size of the buffer - no terminating null is used.

**Returns:**

**int nChars.** The number of characters actually retrieved and placed into the buffers pointed to by **lpChar** and **lpSrc** (if non-NULL).

---

void PASCAL **P\_GatherAndMatchInputStream** (BYTE *Filter*)

Call **P\_GatherAndMatchInputStream()** to allow the Procomm Plus gatherer to gather characters and buffer them. **P\_GRXGet()** and **P\_GRXQueueEmpty()** can then be sensibly called. **P\_GatherAndMatchInputStream()** can be called as often as needed. It can simply be called off the timer, or it can be called only after **P\_GRXGet()** fails or **P\_GRXQueueEmpty()** returns TRUE.

**P\_GatherAndMatchInputStream()** 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.

**Arguments:**

**Filter.** Defines which input streams to read from. The flags are bit flags, with the value of COM port defined in **P\_GRXGet**. Typically, one would call this function with the value **ALL\_STREAMS = (COM\_PORT\_STREAM | LOCAL\_ECHO\_STREAM | PLAYBACK\_STREAM)**.

## Index of Functions

<b>P_GRXQueueEmpty</b> .....	<b>10</b>
<b>P_AddToTXBuf</b> .....	<b>5</b>
<b>P_ClearCommBreakW</b> .....	<b>5</b>
<b>P_DisplayStatusMessage</b> .....	<b>6</b>
<b>P_EscapeCommFunctionW</b> .....	<b>5</b>
<b>P_FlushCommW</b> .....	<b>5</b>
<b>P_GatherAndMatchInputStream</b> .....	<b>11</b>
<b>P_GetCommErrorW</b> .....	<b>5</b>
<b>P_GetCommStateW</b> .....	<b>5</b>
<b>P_GRXEnableString</b> .....	<b>10</b>
<b>P_GRXGet</b> .....	<b>10</b>
<b>P_GRXRegisterSpecialString</b> .....	<b>9</b>
<b>P_GRXRegisterString</b> .....	<b>8</b>
<b>P_GRXUnregisterString</b> .....	<b>10</b>
<b>P_ReadCommW</b> .....	<b>5</b>
<b>P_ReadNamedStruct</b> .....	<b>6</b>
<b>P_SetCommBreakW</b> .....	<b>5</b>
<b>P_SetCommStateW</b> .....	<b>5</b>
<b>P_StatusLineDirect</b> .....	<b>7</b>
<b>P_WriteCommW</b> .....	<b>5</b>