ithaca®
a product of TRANSACT

Ithaca® Printer Windows Driver API
(Applications Programming Interface)

PN 100-08038, Revision C, November 2008
Proprietary and Confidential

Change History

Rev A	Initial release	September 2007
Rev B	New functions added: Direct I/O, download firmware, and statistics	August 2008
Rev C	Correction for min buffer size for status	November 2008

Disclaimer

Copyright

Trademarks

Table of Contents

This page intentionally left blank

Transact Windows Driver Interface

Who Should Read This Guide?

This document provides information useful to applications programmers and original equipment manufacturers (OEM) who will develop applications for all Ithaca® printers.

What is Included In This Guide?

This document describes extended functionality to the standard Windows printer applications programming interface (API) that is specific for Ithaca® printers.

Refer to the appropriate Programmer's Guide or OEM Integration Manual for your specific printer for complete information on the mechanical, electrical, and command language requirements of each printer, which is not covered in this supplement. For further technical information, visit Transact's on-line support center at **www.transact-tech.com**.

About the Ithaca Windows Printer Driver

This interface definition is provided give the application programmer additional functionality and control useful in developing applications for point of service (POS), banking, and gaming printers. The need for an extension comes from the fact that our printers include functionality that the Windows operating system does not include for a "standard" (normally office / page) printer. Some of these functions include:

- Activate a cutter
- Open a cash drawer
- Accept an inserted form for printing/validation
- Eject an inserted form after printing/validation
- Clear the power cycled status
- Feed a specified number of lines
- Get printer statistics
- Get detailed status

This extended interface and the ability to send raw data to a printer represent a way to control the printer via applications programming. It provides synchronous status returns from the printer as well as additional functionality such as firmware download. Since print jobs sent via the standard Windows spooler service are a potentially concurrent source of output to the printer, the interface has to wait for those jobs when handling the additional functions.

These functions can be invoked in the same way that the standard Windows Graphics Device Interface (GDI) function calls are used to generate and check on a print job**.**

Note: There is no standard definition for the precise nature of how extensions should be defined and implemented in our industries. It is unlikely that other printers will behave as expected if our extended driver is used to operate them.

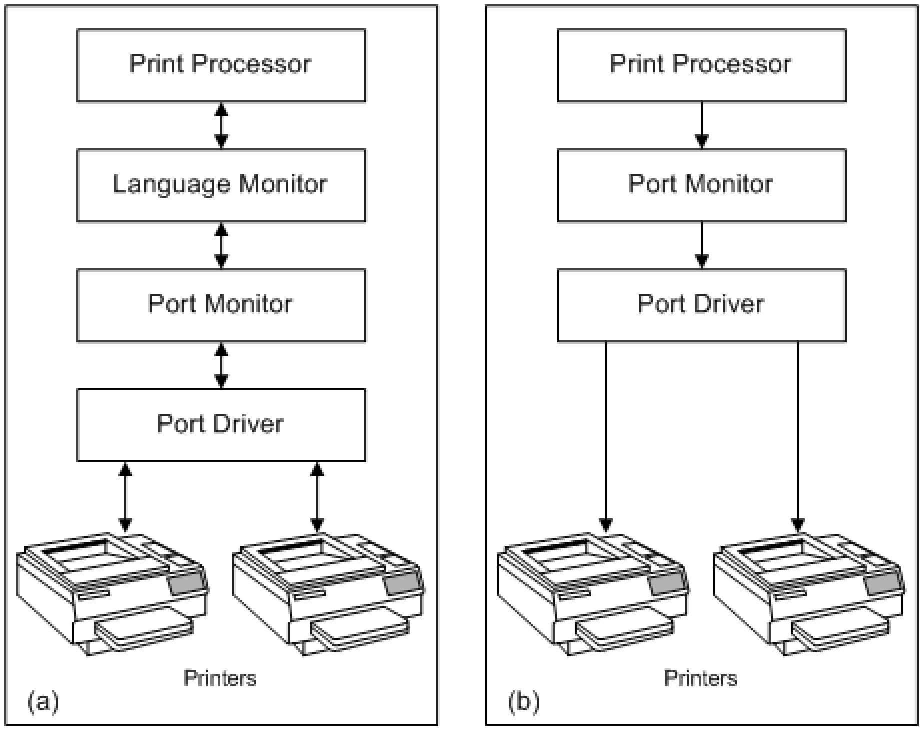
In general, extended functions that return no data can be invoked in the context of a "StartDoc" and "EndDoc," as used for normal print jobs. If additional functionality for receiving status is also desired, this functionality should be invoked *outside* the context of a true "start document" and "end document" segment, with the GetPrinterData() function called to perform the IO. This may require that a large print job be broken up into to smaller subsets, and the extended function called in-between these subsets. Consult the program flow flowchart on page 4 for an overview of an example sequence.

Note: If a Transact printer's Properties window Advanced tab has spooling selected to start printing after the last page is spooled (vs. start immediately), then it might be necessary to enclose GetPrinterData calls within a start document – end document sequence or to set the access value at the time a printer is Opened; see the note later in this document.

Ithaca Printer Extended Interface

The Ithaca printer extended interface is implemented in a module defined in the Windows printer architecture as the print job “Language Monitor”. In this case “language” does not refer to human dialects but to printer command sets.

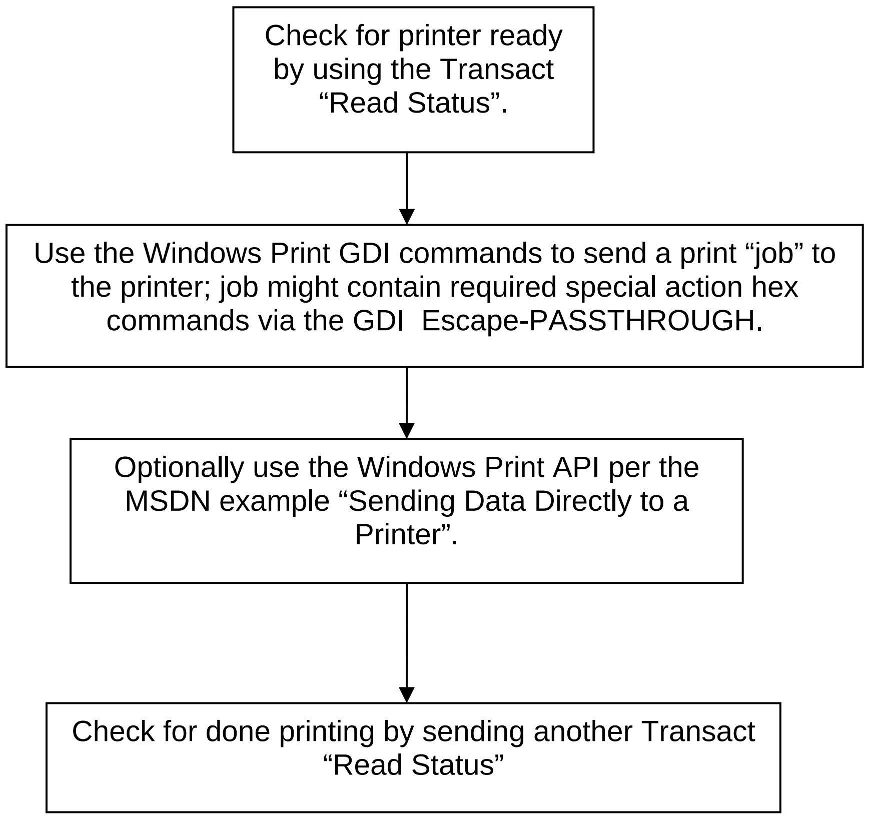
The following diagram, taken from Microsoft documentation, shows the difference between a printer installed (a) with, and (b) without a language monitor. The Print Processor is the largest module here and includes the print spooling function and the Graphical User interface widow, “Printers and Faxes” available as a selection from the Control Panel window.



Before you can use the extended printer interface functions, you must install the standard Windows provided printer drivers that are part of the appropriate Transact provided Windows Print Driver install package. The install package includes a Language Monitor along with the specific printer model “mini driver” that works in conjunction with the Windows printing system.

Note: When you distribute your application you will need to distribute and install the Windows Print Driver install directory with your application. Most install package tools will take care of this for you.

Typical Program Flow Flowchart


Check for printer ready by using the Transact "Read Status".
Use the Windows Print GDI commands to send a print "job" to the printer; job might contain required special action hex commands via the GDI Escape-PASSTHROUGH.
Optionally use the Windows Print API per the MSDN example "Sending Data Directly to a Printer".
Check for done printing by sending another Transact "Read Status"

Transact extended definitions

The following definitions are used as the API for Transact functions; they are based on documentation taken directly from the MSDN. Application coding is done to the standard Windows Print spooler service GDI-based interface. Our functions extend those of Microsoft and are defined by printer specific pValueNames. These Transact values are defined as constant strings which can be copied directly from this document. All other values are left as-is and thus eventually passed directly to a standard Windows port monitor.

GetPrinterData

The **GetPrinterData** function retrieves configuration data for the specified printer or print server.

Windows 2000/XP: Calling **GetPrinterData** is equivalent to calling the **GetPrinterDataEx** [http://msdn2.microsoft.com/en-us/library/ms535646.aspx] function with the *pKeyName* parameter set to "PrinterDriverData".

```
DWORD GetPrinterData(
  HANDLE  hPrinter,    // handle to printer or print server
  LPTSTR  pValueName,  // value name
  LPDWORD pType,       // data type
  LPBYTE  pData,       // pointer to data buffer
  DWORD   nSize,       // size of data buffer
  LPDWORD pcbNeeded    // bytes received or required );
```

Parameters

hPrinter

[in] Handle to the printer or print server for which the function retrieves configuration data. Use the **OpenPrinter** [http://msdn2.microsoft.com/en-us/library/ms536027.aspx] or **AddPrinter** [http://msdn2.microsoft.com/en-us/library/ms535500.aspx] function to retrieve a printer handle.

pValueName

[in] Pointer to a null-terminated string that identifies the data to retrieve.

For printers, this string is the name of a registry value under the printer's "PrinterDriverData" key in the registry. Note that Transact, keys for the Language Monitor are stored in a Print\Monitors section and are not retrieved by this function.

For printers with the Transact Language Monitor installed, uniquely named pValueName strings are used to invoke the extended functions.

pType

[out] Pointer to a variable that receives the type of data retrieved. The function returns the type specified in the **SetPrinterData** [http://msdn2.microsoft.com/en-us/library/ms535657.aspx] or **SetPrinterDataEx** [http://msdn2.microsoft.com/en-us/library/ms535693.aspx] call when the data was stored.

Note that for the API this value is ignored by the LM when one of the *pValueNme* strings is a Transact extended function; 0 (NULl) is suggested.

pData

[out] Pointer to a buffer that receives the data. For Transact extended functions, the size of this buffer is at least 8 bytes long for status and at least 256 bytes for statistics.

nSize

[in] Specifies the size, in bytes, of the buffer pointed to by *pData*.

pcbNeeded

[out] Pointer to a variable that receives the size, in bytes, of the received data. If the buffer size specified by *nSize* is too small, the function returns ERROR_MORE_DATA, and *pcbNeeded* indicates the required buffer size.

For Transact extended functions, *pcbNeeded* indicates the size, in bytes, of data from the printer.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS. If the function fails, the return value is an error value.

Remarks

GetPrinterData in non-API use retrieves printer-configuration data set by the **SetPrinterDataEx** [http://msdn2.microsoft.com/en-us/library/ms535693.aspx] or **SetPrinterData** [http://msdn2.microsoft.com/en-us/library/ms535657.aspx] function.

GetPrintData in non-API use may trigger a Windows call to **GetPrinterDataFromPort**. (See the Windows Development Kit, formerly Driver Development Kit, for more information about **GetPrinterDataFromPort**.) The latter function may write to the registry. If it does there may be side effects like triggering an update/upgrade printer event ID 20 in the client, if the printer is shared in a network.

For Microsoft defined *pValueName val*ues please refer to MSDN documentation.

Transact Extension to GetPrinterData

The above method has been extended using the defined parameters, with values defined by Transact, for the Language Monitor ("LM") to first interpret the pValueName parameter and take special actions for values defined in this document. All other parameter values are passed to a standard, Microsoft supplied, port monitor..

An application invokes these method extensions by providing parameter values given below when calling GetPrinterData.

Transact I/O pValueName Functions

Defined Constants

```
/* These pValueName parameter values should be used to construct the string used to
specify GetPrinterData actions proprietary to Transact printers after the Windows Print
Driver is installed: */

TSTR Const TAReadStatus          = "TAReadStatus";
TSTR Const TAGetStatistics       = "TAGetStatistics";
TSTR Const TADirectIO            = "TADirectIO";
TSTR Const TASetDownloadMode     = "TASetDownloadMode";
TSTR Const TAFirmwareDownload    = "TAFirmwareDownload";
```

ReadStatus

This function returns the status bytes defined in the Printer Programmer's Guide PcOS command: Inquire all Printer Status. Refer to that document for your printer model for a definition of the meaning of each bit in the returned data. Response to this call will return the latest available printer status, and the number of status bytes available will be returned in pcbNeeded.
Parameters:

```
  HANDLE   hPrinter,     // Handle returned from opening the printer
  LPTSTR   pValueName,   // string “TAReadStatus”, 0 terminated
  LPDWORD  pType,        // = 0 (NULL)
  LPBYTE   pData,        // I/O data buffer address (such as &pData[0])
  DWORD    nSize,        // >=10
  LPDWORD  pcbNeeded     // the provided buffer should receive several
bytes
```

Note and Example

Note: Depending on how the mini-driver was installed or what process is monitoring the printer, it is often necessary to specify required access rights when invoking the Open printer method, which precedes asking for a printer’s status.

Here is a C language example code which shows a status read before and after sending raw data to the printer:

```
#include <Windows.h>
#include <StdIO.h>

// ********************************************************************
// RawDataToPrinter - sends binary data directly to a printer
//
// Params:
//   szPrinterName - NULL terminated string specifying printer name
//   lpData        - Pointer to raw data bytes
//   dwCount       - Length of lpData in bytes
//
// Returns: TRUE for success, FALSE for failure.
//
BOOL RawDataToPrinter(LPTSTR szPrinterName, LPBYTE lpData, DWORD
dwCount)
{
HANDLE     hPrinter;
DOC_INFO_1 DocInfo;
DWORD      dwJob;
DWORD      dwBytesWritten;

TBYTE statBuff[20];
DWORD statLen = sizeof(statBuff);

PRINTER_DEFAULTS hPrnDef = {0}; // pDatatype and pDevMode of structure
are set to NULL

hPrnDef.DesiredAccess = PRINTER_ACCESS_ADMINISTER | PRINTER_ACCESS_USE
; // set desired access rights

// Need a handle to the printer.
if(! OpenPrinter(szPrinterName, &hPrinter, &hPrnDef))
{
      printf(“OpenPrinter abort error %d”, GetLastError());
      return FALSE;
}
// Report on the status
```

```
DWORD err = GetPrinterData(hPrinter, "TAReadStatus", NULL, statBuff,
sizeof(statBuff), &statLen);
printf("TAReadStatus read result %d; len %d:", err, statLen);
for (WORD i = 0; i < statLen; i++) printf("%X ", statBuff[i]);
printf("\n");     // Show the returned status in hex

// Fill in the structure with info about this "document."
      DocInfo.pDocName = TEXT("My Document");
      DocInfo.pOutputFile = NULL;
      DocInfo.pDatatype = TEXT("RAW");
      // Inform the spooler the document is beginning.
      if((dwJob = StartDocPrinter(hPrinter, 1, (LPBYTE)&DocInfo)) == 0)
      {
            printf("StartDocPrinter abort error %d", GetLastError());
            ClosePrinter(hPrinter);
            return FALSE;
      }
      // Start a page.
      if(! StartPagePrinter(hPrinter))
      {
            printf("StartPagePrinter abort error %d", GetLastError());
            EndDocPrinter(hPrinter);
            ClosePrinter(hPrinter);
            return FALSE;
      }
      // Send the data to the printer.
      if(! WritePrinter(hPrinter, lpData, dwCount, &dwBytesWritten))
      {
            printf("WritePrinter abort error %d", GetLastError());
            EndPagePrinter(hPrinter);
            EndDocPrinter(hPrinter);
            ClosePrinter(hPrinter);
            return FALSE;
      }
      // End the page.
      if(! EndPagePrinter(hPrinter))
      {
            printf("EndPagePrinter abort error %d", GetLastError());
      EndDocPrinter(hPrinter);
            ClosePrinter(hPrinter);
            return FALSE;
      }
      // Inform the spooler that the document is ending.
      if(! EndDocPrinter(hPrinter)
      {
            printf("EndDocPrinter abort error %d", GetLastError());
            ClosePrinter( hPrinter );
            return FALSE;
      }
      // Check to see if correct number of bytes were written.
      if(dwBytesWritten != dwCount)
      {
            printf(TEXT("Wrote %d bytes instead of requested %d
bytes.\n"), dwBytesWritten, dwCount);
            ClosePrinter(hPrinter);
            //return FALSE;
      }
      else printf("Entire file data was sent to printer.\n");
      // Show current printer status.
      statLen = sizeof(statBuff);
      err = GetPrinterData(hPrinter, "TAReadStatus", NULL, statBuff,
sizeof(statBuff), &statLen);

      printf("TAReadStatus After print result %d; len %d:", err,
statLen);
      for (WORD i = 0; i < statLen; i++) printf("%X ", statBuff[i]);
printf("\n");

      // Tidy up the printer handle.
      ClosePrinter( hPrinter );
      return TRUE;
}
```

GetStatistics

This function returns the printer statistics data defined in the PcOS diagnostic commands . Refer to appendix A for a definition of the returned data, which varies per printer model. If the return buffer length is insufficient to return all the statistics values, pcbNeeded will indicate the size of the buffer needed. If successful, each statistic value is returned as 4 contiguous bytes of data and the number of bytes available in pData are returned in pcbNeeded. If the printer is in an error state (cover open, paper out etc), the GetStatistics call returns with a delay of several seconds and a return buffer of zeros. Example code follows.

Parameters:

```
  HANDLE   hPrinter,     // Handle returned from opening the printer
  LPTSTR   pValueName,   // string “TAGetStatistics”, 0 terminated
  LPDWORD  pType,        // NULL
  LPBYTE   pData,        // I/O data buffer address (such as &pData[0])
  DWORD    nSize,        // >= 256
  LPDWORD  pcbNeeded     // the buffer should receive <= 256 bytes
```

```
#include <winspool.h>
void CCLMTestDlg::OnBnClickedGetstats()
{

      DWORD Type = 0;
      DWORD nCount = 0;
      BYTE RetBuf[1024];
       int nReadSz = 140;

      // Invoke the GetPrinterData for GetStatistics and display the return value
      //
       int nError = GetPrinterData(hPrinter, "TAGetStatistics", &Type, RetBuf,
                                        nReadSz, &nCount);
      // If statistics are available Display the results
      //
```

```
        If (nCount > 0)
                DisplayStats(RetBuf);
        return;

}
```

DirectIO

This function sends DirectIO data to the printer. The data string followed by a ":" and each byte separated by a " ", is appended, to the pValueName. pData should be of sufficient size to hold the return data from the printer. nSize indicates the size of the pData buffer sent by the user. If pData is of insufficient size, pcbNeeded indicates the number of bytes needed to hold the printer return data. If successful pData contains the response returned by the printer and pcbNeeded indicates the number of bytes returned. An example of a DirectIO call follows:

Parameters:

```
  HANDLE   hPrinter,    // Handle returned from opening the printer
  LPTSTR   pValueName,  // string "TADirectIO[|| Direct Out]", 0
terminated
  LPDWORD pType,        // = 0 (NULL)
  LPBYTE   pData,       // I/O data buffer address (such as &pData[0])
  DWORD   nSize,        // the number of bytes to send to the printer
  LPDWORD pcbNeeded     // the expected length of bytes to receive;
this will indicate the actual number received, which may be 0.
```

```
#include <winspool.h>
void DoDirectIO()
{
        // Send down the user entered data as part of the "TADirectIO" string
        // This is for "ENQ 21", hex value 05 15 command
        CString strDirectIO = "05 15";


        // Tokenize the user command and append each of the tokens
        //
        int nStart = 0;
        CString strTokens = " ";
        CString strToSend = "TADirectIO";

        strToSend.Append(":" +strDirectIO);

        DWORD Type = 0;
        DWORD Needed = 0;
        DWORD nCount = 0;
        BYTE RetBuf[1024];
        CString strRes = "";
        int nReadSz = 1000;

        LPSTR strData = strToSend.GetBuffer(strToSend.GetLength());
        // Invoke the GetPrinterData for directIO and display the return value
        //
        int nError = GetPrinterData(hPrinter, strData, &Type, RetBuf, nReadSz,
                                    &nCount);
        strToSend.ReleaseBuffer();
        DisplayResult(RetBuf);
}
```

Printer Firmware (FW) Download

A pair of functions is used to download a firmware file into the printer. If the printer's current FW does not have a 2nd level loader and download mode command, the printer should be manually set in "bootloader" mode and the "TASetDownloadMode" function can be skipped. In each case pcbNeeded indicates the status of the call made. If the function is successful, ERROR_SUCCESS is returned,else the value returned indicates the error that occurred.

Set Download Mode

Parameters:

```
  HANDLE   hPrinter,    // Handle returned from opening the printer
  LPTSTR   pValueName,  // :string "TASetDownloadMode", 0 terminated
  LPDWORD  pType,       // = 0 (NULL)
  LPBYTE   pData,       // I/O arbitrary buffer address (such as
&pData[0])
  DWORD   nSize,        // the size of the arbitrary buffer, such as 4
  LPDWORD pcbNeeded     // address of the length received; for download
mode value returned will be 0.
```

This first part requests to printer to ender download mode. As this may result in resetting the established port for communication (equivalent to unplugging and then plugging in again), the connection must be reacquired before the Language Monitor is able to write to the printer – this is accomplished by closing and reopening the printer and then invoking the next function.

Firmware Download

The name of the firmware file is appended to the pValueName parameter. If the download succeeds the printer resets at the end of the firmware download. If the firmware download fails the printer lights indicate a firmware download error. An example of code that invokes these calls follows:

Parameters:

```
  HANDLE   hPrinter,    // Handle returned from opening the printer
  LPTSTR   pValueName,  // string "TAFirmwareDownload[|| [file path
name]", 0 terminated
  LPDWORD  pType,       // = 0 (NULL)
  LPBYTE   pData,       // I/O data arbitrary buffer address (such as
&pData[0])
```

```
  DWORD   nSize,        // the number of bytes to send to the printer
  LPDWORD pcbNeeded     // the expected length of bytes to receive; for
download this is ignored and when returned it is set to 0.
```

```
DWORD DownloadFirmware(CString fwFileName)
{
        DWORD pType = 0;
        DWORD nSize = 0;
        DWORD pcbNeeded = 0;
        BYTE retBuf[1];


        // Invoke the GetPrinterData for SetDownloadmode

        GetPrinterData(hPrinter, "TASetDownloadMode", &Type, RetBuf, 0,
        &pcbNeeded);

        // If startdownload mode successful, close the printer and reopen
        //
        if (pcbNeeded == ERROR_SUCCESS)
        {
                ClosePrinter(hPrinter);
                DoPrinterOpen();
        }
        else
        {
                return pcbNeeded;
        }

        // Append the firmware filename to the pValueName
        //
        CString strDownload = "TAFirmwareDownload";
        strDownload += fwFileName;
        LPSTR pValueName = strDownload.GetBuffer();
        GetPrinterData(hPrinter, pValueName, pType, retBuf, nSize,
                                                &pcbNeeded);
        strDownload.ReleaseBuffer();
        return pcbNeeded;
}
```

A DoPrinterOpen for a GUI edit box that accepts the printer name can be similar to the following:

```
void TestDlg::DoPrinterOpen()
{

        PRINTER_DEFAULTS PrnDefs;

        PrnDefs.DesiredAccess = PRINTER_ALL_ACCESS;
        PrnDefs.pDatatype = 0;
        PrnDefs.pDevMode = NULL;
        // retrieve the name from an edit box
        CString strPrnName;
        m_TxtName.GetWindowText(strPrnName);
        if (strPrnName.GetLength() == 0)
        {
              ShowErrorDlg("Invalid Printer Name", 0);
              return;
        }
        if (OpenPrinter((char*)(LPCTSTR)strPrnName, &hPrinter, &PrnDefs))
              bPrnOpen = TRUE;
        else
        {
              CString errMessage = "Open Printer Failed";
              ShowErrorDlg(errMessage, GetLastError());
        }

}
```

Appendix A: Statistics Returns

All statistics are returned as arrays of 4 byte count values, each value in Intel (‘little endian”) byte order. Please check with your compiler and modify the “unsigned int” type definition if needed to get 4 byte variables (DWORD is the same as unsigned int for MS Visual Studio on 32 bit operating systems)

iTherm 280:

```
typedef struct  CFG_STAT280
{
      unsigned int reserved;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Base_Flash_Erases;
      unsigned int Ext_Flash_Erases;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;
      unsigned int OverTemps;
      unsigned int Cutter_ReHome;
      unsigned int Jam_Detect_L1;
      unsigned int Jam_Detect_L2;
      unsigned int Missed_TOF;
      unsigned int Cash_Drawer_Opens;
      unsigned int Config_Faults;
      unsigned int Spare27;
      unsigned int Spare28;
      unsigned int Spare29;
      unsigned int Spare30;
      unsigned int RAM_Code_Mismatch;
};
```

An example of a declaration for the buffer to be supplied to the GetPrinterData function is:

CFG_STAT280 stat280;

The parameters used for GetPrinterData are then &stat280 and sizeof(stat280).

Note: The Language Monitor uses an incrementing 4 bye counter to fill the statistics buffer, so that alignment and type definitions need to be changed if your compiler does not produce the same address incrementing for the references as defined above.

PJ1000:

```
typedef struct CFG_STAT1000
{
      unsigned int Total_Heads;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Head_ReIndex;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;

};
```

PJ1500 / PJ1580 / PJ1600 / PJ1680

```
typedef struct  CFG_STAT1500
{
      unsigned int Total_Heads;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Head_ReIndex;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;
      unsigned int Slips_Inserted;
      unsigned int RAM_Code_Mismatch;
      unsigned int Config_Faults;
      unsigned int FlashFileFault;
      unsigned int Ext_Flash_Erases;
      unsigned int Spare24;
      unsigned int Spare25;
      unsigned int Spare26;
      unsigned int Val_Line_Feeds;
      unsigned int Spare28;
      unsigned int Spare29;
      unsigned int Spare30;
      unsigned int Spare31;
};
```

International 280i

```
typedef struct  CFG_STATIN280
{
      unsigned int reserved;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Base_Flash_Erases;
      unsigned int Ext_Flash_Erases;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;
      unsigned int OverTemps;
      unsigned int Cutter_ReHome;
      unsigned int Jam_Detect_L1;
      unsigned int Jam_Detect_L2;
      unsigned int Missed_TOF;
      unsigned int Config_Faults;
      unsigned int Cash_Drawer_Opens;
      unsigned int FlashFileFault;
      unsigned int Spare28;
      unsigned int Spare29;
      unsigned int Spare30;
      unsigned int RAM_Code_Mismatch;
};
```

Epic 430

```
typedef struct  CFG_STAT430
{
      unsigned int reserved;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Base_Flash_Erases;
      unsigned int Ext_Flash_Erases;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;
      unsigned int OverTemps;
      unsigned int Cutter_ReHome;
      unsigned int Jam_Detect_L1;
      unsigned int Jam_Detect_L2;
      unsigned int Missed_TOF;
      unsigned int Config_Faults;
      unsigned int Cash_Drawer_Opens;
      unsigned int FlashFileFault;
      unsigned int Spare28;
      unsigned int Spare29;
      unsigned int Spare30;
      unsigned int RAM_Code_Mismatch;
};
```

BJ2500

```
typedef struct  CFG_STAT2500
{
      unsigned int reserved;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Head_ReIndex;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;
      unsigned int Slips_Inserted;
      unsigned int RAM_Code_Mismatch;
      unsigned int Config_Faults;
      unsigned int FlashFileFault;
      unsigned int Ext_Flash_Erases;
      unsigned int External_Resets;
      unsigned int Software_Resets;
      unsigned int PLL_Resets;
      unsigned int Val_Line_Feeds;
      unsigned int Spare28;
      unsigned int Spare29;
      unsigned int Spare30;
      unsigned int Spurious_IRQ;
};
```

Epic 630:

```
typedef struct  CFG_STAT630
{
      unsigned int reserved;
      unsigned int Cover_Opens;
      unsigned int Paper_Outs;
      unsigned int Line_Feeds;
      unsigned int Characters_Printed;
      unsigned int Cash_Drawer1;
      unsigned int Cash_Drawer2;
      unsigned int Standby_Cycles;
      unsigned int Power_Up_Resets;
      unsigned int Watchdog_Resets;
      unsigned int Base_Flash_Erases;
      unsigned int Ext_Flash_Erases;
      unsigned int Auto_Cutter_Cycles;
      unsigned int Init_Requests;
      unsigned int Error_Vectors;
      unsigned int Auto_Cutter_Faults;
      unsigned int Power_On_Time;
      unsigned int System_Active_Time;
      unsigned int OverTemps;
      unsigned int Cutter_ReHome;
      unsigned int Jam_Detect_L1;
      unsigned int Jam_Detect_L2;
      unsigned int Missed_TOF;
      unsigned int Cash_Drawer_Opens;
      unsigned int Config_Faults;
      unsigned int Spare27;
      unsigned int Spare28;
      unsigned int Spare29;
      unsigned int Spare30;
      unsigned int RAM_Code_Mismatch;
};
```

[end of document]