NetPod

Software Developers Guide

Version 1.04

June 2001

Basic Concepts
Examples
Program Sequences
Configuration Management
Data Acquisition Techniques
NetPod Driver Description

Figures

1 Introduction

1.2 Purpose of this manual

This manual describes the possibilities of interaction between the NetPod data acquisition systems and the different Windows operating systems programming languages. It shows the structure that an applications program must follow to acquire and process real time information in a multi-user, multi-instrument environment across a series of different types of communication network and also, a list of functions and variables available to undertake these tasks. Any operating system that supports TCP/IP communications protocol can access data from the instruments. This manual is limited to the software support for the Microsoft Windows Operating Systems.

1.3 The Advantages Of Using NetPod Software

NetPod software has been designed to acquire data from multiple instruments operating at different acquisition rates in a multi-user environment. The programming examples have been presented to enable software development to be undertaken in the Microsoft Windows 95, 98 and NT operating systems. Software can be easily created for single as well as multi-user applications using many instruments. Finally, multi-user multi-instrument operations using the high speed Ethernet networks. The NetPod libraries support full TCP/IP communications protocol that enables the instruments to be integrated into most types of Ethernet systems.

1.4 Difficulties encountered using Windows Operating Systems for data acquisition operations.

Microsoft Windows is the most popular operating systems for personal computers. It provides an easy to use programming environment supported by many different programming languages from different venders. Programming large-scale data acquisition and control applications can be complicated, the reasons for this are:

- Microsoft Windows 95, 98 and NT are real time operating systems.

- The Windows operating systems enable applications to undertake actions at any time. This means that applications can be interrupted for an unknown amount of time. There is no maximum delay allocated to the interruptions and it is not possible to know in advance, how much time will be spent to undertake a specific task.

2 Instrument Operations

2.2 Introduction

The following chapter summarises the data communication operations of the NetPod and has been included to demonstrate how to select, and use the various communication ports available within instrument.

NetPod contains up to three separate communication ports. 10Base2/T ports for high-speed Ethernet communications, RS232 for direct serial port operations and finally an RS485 port for low speed long distance multi-instrument operations. Occasionally, the RS485 port is changed for an RS422 device.

2.3 Data Transmission – Communication Port Selection

The instrument can be set to transmit data to an operator over any of the communication ports. There are no software commands to select the chosen communications port, instead, the operation is carried out automatically by the instrument.

Figure 2 - Instrument Layout

Up on powering the instrument, the microprocessor system within NetPod monitors the communication ports for traffic. The instrument watches for Ethernet packets broadcast from a LAN or control signal changes on the serial ports caused by connection of the instrument to host computer or industrial network.

NetPod assigns the communication port for data transmission to the one that receives the first items of data or control signals. For example, on connecting the instrument to a LAN via a 10Base2 coax, NetPod will detect the Ethernet packets and assign the 10Base2 coax port for communications.

The only way to change the active communication port is to switch off the instrument and restart connected to the next interface.

2.4 Data Access Control

Figure 2, shows how data is accessed and control of the instrument is carried out by the driver software. The only actions of interest to the developer are function calls to the NetPod.dll. This piece of software is used for all operations allowed to be undertaken within third party software.

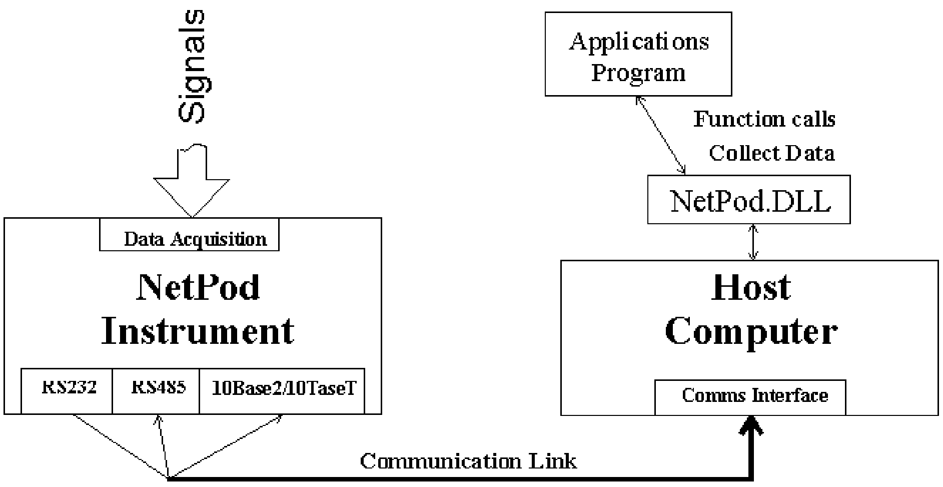


Figure 3 - Driver Software Operations

The developer has no control/responsibility over the selection of the communications port that is going to be used for data transmission and is only allowed to send and receive instructions and receive data from the driver software. These instructions can be used to set all of the features of the instrument and receive data from across a network.

The NetPod system has been carefully designed to minimise the complexity of the code that is required to be learnt before any applications can be developed. The developer does not have to consider how data is gathered or control details are passed to the instruments, only that certain function calls need to be applied for these tasks to be carried out. The NetPod.dll automatically assigns communications to the required port and takes care of all management and control tasks required for the network. This frees the developer to concentrate on the user interface and tasks associated for the project and not on the intricacies of low level communications control.

The **NetPod.dll** allows multi-tasking applications to be developed. You can have databases and analysis programs running simultaneously on the same computer. Section 3.2, Page 11 details how the comms. ports are assigned.

3 Software Development Tips

The following chapter describes the sequence of operations to follow in order to develop NetPod applications. The chapter has been included to help you understand the function calls available within netpod.dll and how they are used.

3.2 Communications

The applications you will develop will be general purpose and function correctly no matter how the instrument has been configured and used. All you have to do, as the developer is concentrate on the higher level operations such as, setting sample rates, reading configuration details etc. You do not need to understand Ethernet or serial port operations to acquire data.

3.3 Hardware Features – Understanding the data.

All of the instrument configuration details is stored within each instrument on a series of flash EEPROMs. Each processor card stores global details such as Sample rate, Ethernet and IP addresses, digital port status etc. Global details are all those parameters that define the system operations.

Each analogue card stores its own calibration details within it’s own EEPROM such as, pre-amp gain, Input range, Type Thermocouple etc. Since the channel modules contain their own configuration details they can be swapped, put into storage etc. without any requirement to reconfigure or calibrate.

The use of EEPROM within the instrument has removed the requirements to maintain or configure any set-up files on the computers using the units for data acquisition.

3.4 Initialisation

Before you can do anything with NetPod you should:

1. Scan for the instruments across the communication interfaces. This is the only occasion you have to select which port should be used for comms.
2. Start the podmng.exe taskbar software.

   Scan Network – Ethernet Operations
   Scan Port – RS485/RS232 serial port operations.

Once you have identified the instruments you must then interrogate them for their configuration settings i.e. sample rates, gain settings etc. You require this information in order to understand the data that will be sent to you from the instruments.

3.5 *Multi-Instrument Operations*

<u>Single User Mode</u>

Multi-instrument operations can be undertaken in **single user mode** using the RS-485 network or Ethernet LAN.

Applications using an RS-485 network are slow, this due to the polling operations of the instruments when connected to this type of network. You will only be able to use sample rates in the order of 10 Hz.

For applications using Ethernet LAN you have access to the full operational flexibility of NetPod. You can have many instruments operating at different sample rates across many different types of LAN.

For multi-instrument operations you should:

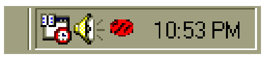
1. *Ensure that there are no NetPod acquisition operations.*
2. *Scan network to create a list of instrument PID numbers. These numbers uniquely identify each instrument.*
3. *For each PID:- obtain sample rate and channel configuration details. This information is required to understand the instrument data.*

For applications were instruments may be added to, or removed in order to meet project requirements you should:

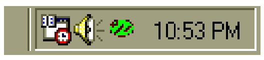
1. *Stop data acquisition operations.*
2. *Scan for new instruments each time the data acquisition operations are interrupted by a user.*
3. *Compare current instrument list with the details from the new scan.*
4. *Update records.*

Once you have stopped data acquisition from the instruments you will observe that the taskbar software icon change from green to red.

Figure 4 - Podmng (taskbar) software



Data Acquisition Operations - Stopped



Data Acquisition Operations - Active

Note. You cannot read instrument configurations when the instruments are transmitting data.

3.6 Multi-user Operations

Multi-user operations are only available when the instruments are connected to a LAN. Data is passed directly onto a network by each instrument. Every user can access data using the netpod.dll.

Each individual user can read and process data or pass it to applications programs. A **password protection system** is available within the podmng.exe software to prevent configuration changes. The protection system restricts the user to viewing data only, and is ideal for applications where unattended data processing is required. It can be used to prevent unauthorised instrument configuration changes.

The protection has two modes:

Admin(user) – Full access to configuration.
Operator – Restricted access. No configuration changes.

The password protection system within podmng.exe package has priority over any commands you can issue from an application program. You cannot change instrument configurations if the protection system is in ***Operator*** mode.

As each new user activates the podmng.exe software, the other applications are automatically informed that a new PC is receiving data. The new podmng.exe application reads the latest instrument configuration settings from the master program. As well as informing the new podmng.exe application of the latest configuration settings, the master program informs each driver of any configuration changes and checks to ensure they have been received.

3.7 Configuration Commands

When you issue configuration change commands to the instruments in a multi-user environment, netpod.dll automatically passes the changes to each user using the LAN. This feature has been developed to remove any requirement to maintain and update set-up files on each users machine.

NetPod does not have any configuration files to maintain. The driver software simply downloads the instrument configurations from the instruments and stores this information within netpoid.dll buffers. When you switch of the computer you loose the configuration details.

To issue configuration commands:

1. *Stop data acquisition operations.*
2. *Select the instrument to be configured. Use instrument PID number. For assigning parameters to all instruments simultaneously use PID =0.*
3. *Write parameters to the chosen instrument.*
4. *Restart acquisition operations.*

Note. When new configuration details are written to the instruments they are automatically passed to the driver software in operation for each user.

3.8 Sample Rates

For multi-instrument operations you can set the sample rates for the individual unit sample rates to be at any speed supported by the chosen instrument. You must remember that during the analysis of multi-rate data you must apply the correct parameters to the information you are processing.

3.9 Real-time Data

Information is accessed from the data buffers within netpod.dll in a channel-by channel basis. It is important that you have an up to date list of the instruments and channel configurations in order that you process the acquired data correctly.

You can read information in any order. You are not restricted to reading channel 1 of Pod 1 and working through a specified sequence. You are free to ignore channels if they are not required for processing operations.

Operations to follow to read data:

1. *Ensure you have an up to date instrument and channel configuration details.*
2. *Assign data buffers to read and store the incoming data.*
3. *Data acquisition operations are active i.e. task bar icon is green*

There are two functions available for reading data:

1. Simple read function is used to read a single data item from each channel.
2. Channel read function is used to obtain a block of raw data from a specified analogue input.

Details of these functions can be found in **Section 5.3**

When reading data you not have to worry about how it is acquired and passed to the host PC. The same function calls are used no matter if data is transmitted across Ethernet or serial ports.

Continuous Operations

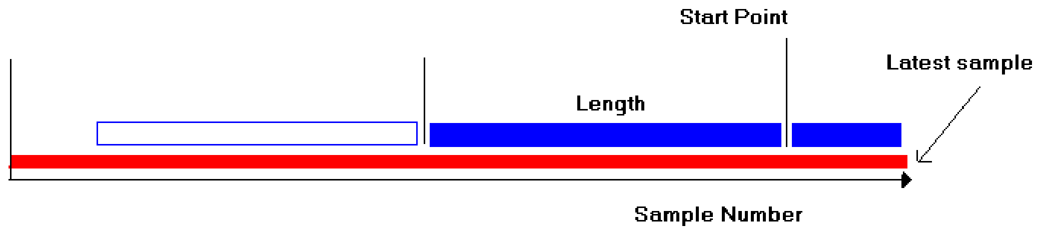
The following section describes how Keynes Controls undertakes continuous data acquisition operations within the podmng.exe application. It has been included to help you understand and develop your own techniques.

Data is acquired using the channel acquisition functions. Each sample as it is acquired is given a sample number. This sample number starts at 0 and is only incremented when data is being acquired. Each channel has its own sample number and this number is only incremented at the sample rate of the instrument from which the channel is associated. For example, if you are reading data from an instrument acquiring information at a sample rate of 100 Hz, then the sample number will increment at 100 counts a seconds.

The sample number is a 64 Bit integer (Quad word). For development languages that do not support quad word integers, the use of double precision floating point number will suffice.

Using the **NP_ChannelBufRead function** request a block of data for a specified channel. For this function you must define a starting point for the sample number and always keep track of the number of samples you have acquired. Note the function **NP_ChannelBufRead** reads data in blocks. To obtain information regarding the latest sample number you must examine the data stream using the **NP_GetBufParam** function**.**

Figure 5 - Data Acquisition Operations



You will observe in Figure 5 how data is read into applications programs. Information is continually gathered and appended to the data stream for each [redacted]st [redacted]. The length of the data block t[redacted]er defined data arrays.

To obtain a continuous trace the applications program must keep calling the **NP_ChannelBufRead** function with updated start samples ensuring that the latest sample required is just less than latest sample acquired by the hardware.

3.10 Digital Port Operations

The digital ports used within NetPod can be assigned as digital inputs or outputs. The type of port to which a channel can be configured depends solely up on the type of interface installed.

Digital output ports can be used to supply TTL level signals and also to switch the relay and triac units.

To read digital input port levels you should:

1. Ensure data acquisition operations are active.
2. Digital port configurations are known.
3. Complete list of instrument PID numbers has be gathered.
4. Select instrument PID – choose instrument whose I/O levels are to be read.

5. Assign flag = NP_Digital
6. Call function **NP_SimpleRead**

To read digital information call function
NP_SimpleRead with flag = NP_DIGITAL

Note. The result should be a pointer to a 32bit integer to receive the digital data (single shot)

or

Use the **NP_ChannelBufRead** with flag = NP_DIGITAL
results must be an array of 32bit integers (digital trace)

3.11 Multiplexer Control

The analogue signals from the sensors connect directly to the inputs of the multiplexer (as shown below). Digital signals are used to select the sensor input that is passed through to the analogue output port. The digital port utilises three digital inputs to select any one of the eight channels available within the MUX. Note, the three channels gives $2^3 = 8$ combinations of different signals. The output from the MUX is passed directly to an analogue input within NetPod.

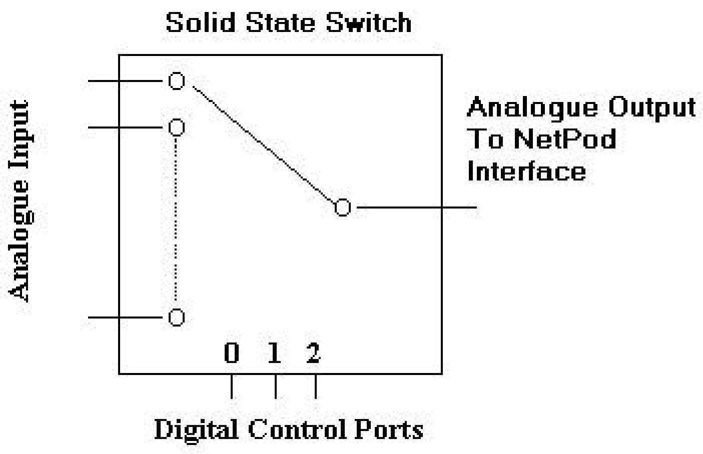


Figure 6 - Multiplexer Operations

The multiplexer utilises a solid state switch unit and takes its digital driver signals from the digital I/O available from NetPod. On switching a multiplexer channel, you must allow time for it to settle. This settling time is much faster for a solid state MUX than those available from a mechanical device.

NetPod analogue interfaces utilise digital filter operations to provide anti-alias filtering, because of this, on passing a new signal from a multiplexer output to an analogue interface; you have to allow the digital filters time to reach their new steady state i.e. have no effect from the previous data obtained from an earlier MUX position. For NetPod you must allow 5 sample periods to occur after switching the MUX before utilising the new data for analysis.

Once you have switched the multiplexer to a new channel and the filters have settled, you are free to acquire data at the maximum rate of the instrument.

For operations where you have to switch continually from 1 channel to the next, you can only achieve 1/5 the maximum sample rate of the instrument.

MUX Operations – Continual Channel switching

To read data from a MUX follow the sequence of operations below:

1. Ensure data acquisition operations are active
2. Set the digital output ports chosen to drive the Multiplexer to the correct binary output pattern.
3. Write the
4. Read the first 5 analogue values from the desired channel and disregard their results.
5. Increment the channel position i.e. change i/o port levels.

MUX Operations – High Speed Logging

To switch a MUX channel and then read new data at high speed you should:

1. Ensure that data acquisition operations are active.
2. Assign digital I/O port levels to select the new MUX channel.
3. Read 4 sample of analogue data from the input channel used to accept signals from the mux. Disregard these values.

   Note. Reading 4 samples allows mux time to switch and digital filters time to settle.

4. Start continuous data logging at the maximum rate achievable by the instrument used to acquire data.

Note. It is only on switching channels that you have to disregard the first four consecutive samples.

4 Library Of Function Calls

The following chapter describes the function calls available to the software developer to control and read instruments using the netpod.dll.

4.2 Function Calls

Configuration functions

function NP_GetStatus(int stat);
function NP_GetPodInfo(int pid, TPodInfoStruct* Info);
function NP_GetChannelInfo(int pid, int cid, TChanneInfoStruct* Info);
function NP_GetPodList(int* data);
function NP_GetGainList(int pid, int cid, int* list);
function NP_GetSampleRateList(int pid, int cid, int* list);

Data functions

function NP_SimpleRead(int pid,int flag,void* results);
[redacted]

Set configuration functions

function NP_SetDigital(int pid,int value);
function NP_SetStatus(int stat);
function NP_SetStatusExt(int stat, int pid, int cid);
function NP_PutPodInfo(int pid, TPodInfoStruct Info);
function NP_PutChannelInfo(int pid, TChannelInfoStruct Info);

4.3 Configuration Function Details

long PODAPI NP_GetStatus(int stat);

Stat: *Constants as listed below.*

Used to determine the status of the instruments i.e. are they transmitting data (acquisition mode) or set for configuration. Returns a true of false result. False = 0 True > 0.

Example: If an instrument is transmitting data then call the function **ShowMessage**. The command can be used to determine if data is being broadcast to the host computer no matter which communications interface is in use. The source of data link does not have to be specified.

This function is also used to scan Ethernet ports or serial ports for data.

Example. If instruments are transmitting data to the host PC then call function “ShowMessage”.

```
If (NP_GetStatus(NP_ISRUNNING) { ShowMessage(); }
     /* Call function ShowMessage only if the data acquisition operations are active */
```

also when wanting to know if the scanning network operation has completed then call function:

NP_GetStatus(NP_ISFINUPDATE). This function returns true (1) when finished updating, or false (0) should the process of changing/updating be still underway.

The following are valid constants for the ***stat*** parameter.

NP_ISCFGDLGSHOWN	is the main configuration dialog displayed?
NP_ISRUNNING	is the system in run mode – gathering data?
NP_ISCALLBACK	is this the main thread?
NP_ISINITSCAN	has an initial network scan been performed?
NP_INSTANCECOUNT	how many instances of the dll are loaded?
NP_GETOFFLINEPID	get the pid of the first offline instrument.
NP_OFFLINEPODS	return number of offline pods (this value should be zero unless an error has occurred).
NP_SCANNET	Scan 10Base2 & 10BaseT Ethernet ports.
NP_PORT	Scan RS232 & RS485 port.
NP_ISFINUPDATE	Are the instruments still be scanned.

[redacted]

Please note that you should allow a few milliseconds from calling NP_SetStatus(NP_SCANNET) to calling any other routines as the driver is multi-threaded and its the other threads which set these values.

and to scan the serial ports

NP_SetStatus(NP_SCANPORT + comm_number), where comm_number is the comm port number (COM1, COM2 etc. up to 256)

int POPAPI NP_GetPodInfo(int pid, TPodInfoStruct* Info);

Parameters: ***pid*** = Pod ID Number. **info** = Structure containing global instrument settings.

This function returns global configuration details from an instrument pointed to by ***pid***. The global information is stored in the structure pointed to by the parameter ***info***.

The function returns a 0 if it operates successfully or an integer error code >0 if it has failed. **True = 0 False > 0**

Example. Print the IP address of NetPod with PID 7654.

TpodInfoStruct info;
Pid = 7654;
NP_GetPodInfo(pid,&info);
Printf("Ip address is %s\n",info.StrIPAddr);

Global configuration details are those parameters which describe an instrument operations such as sample rate, block size, name etc.

Int PODAPI NP_GetChannelInfo(int pid, int cid, TChannelInfoStruct* Info);

Parameters: ***pid*** = Pod ID Number. ***cid*** = Channel number**.** **info** = Structure containing channel details.

This function returns the information held on the channel pointed to by ***cid*** within an instrument pointed to by ***pid***. Information such as the ADC type, resolution, part and serial numbers etc. are returned as data items in the form of a TChannelnfoStruct structure to the parameter ***info***.

The function returns a 0 if it operates successfully or an integer error code >0 if it has failed. **True = 0 False > 0**

Example. Print the part number for channel 5 of NetPod 7654.

```
int pid,cid;
TChannelnfoStruct info;
pid = 7654;
cid = 5;
NP_getChannelInfo(pid,cid,&info);
printf("Part Number of channel %d is %s",cid,info.PartNumber);
```

int POPAPI NP_GetPodList(int* data);

Function returns a list of POD ID (PID) numbers. Used to identify the instruments connected to a network. The user must allocate an array of suitable length to receive the data. The data array is delimited by a zero ID.

Example. For 2 instruments connected to a host PC across an Ethernet LAN with PID numbers 1234 and 3123 the following code can be used:

```
int pidarray[100];
NP_GetPodList(pidarray);        /* Call all Instruments and return PID numbers.*/
```

Results

pidarray[0]=1234, pidarray[1]=3123, pidarray[2]=0 (delimited result i.e. last PID in list is 0)

The value 0 in pidarray[2] is used to identify the last data item in the list.

Int PODAPI NP_GetGainList(int pid, int cid, int* list);

pid: Pod ID number. ***cid:*** Channel number***.*** ***List:*** Pre-amplifier gain setting available from the analogue cards.

Returns the list of pre-amplifier gain settings available to the analogue input channels within in a specified instrument. The instrument is chosen using the PID number. The data items are delimited with the value 0.

The parameter ***cid*** is reserved for future use.

The function returns a 0 if it operates successfully or an integer error code >0 if it has failed. **True = 0 False > 0**

Example. Read the pre-amp settings for the analogue channels installed within an instrument identified using PID=1234.

```
int pid = 1234;
int cid = 0;
int list[100];

NP_GetGainList(pid,cid,list);
/* list = {8,1,0}     Note. 0 indicates last data item.
```

the pre-amp gain settings are 8,1 etc. */

int PODAPI NP_GetSampleRateList(int pid, int cid, int* list);

pid: Pod ID number. ***cid:*** Channel number***.*** ***List:*** Pre-amplifier gain setting available from the analogue cards.

Returns the sample rate options gain settings available to a specified instrument using the specified PID number. The data items are delimited with the value 0.

The parameter ***cid*** is reserved for future use.

The function returns a 0 if it operates successfully or an integer error code >0 if it has failed. **True = 0 False > 0**

Example. Print the sample rates available to NetPod 7654. Note. It does not matter which comms. interface is used by the instrument. Example C code.

```
int i,pid;
float rates[100];
pid = 7654;
NP_GetSampleRateList(pid,rates);
printf("list of sample rates\r\n");
for(i=0;i<100;i++) {
   print("%f\r\n",rates[i]);
}
```

4.4 Data functions Details

The following functions are used to read configuration details and analogue data.

Int PODAPI NP_SimpleRead(int pid,int flag,void* results);

Parameters: ***pid*** = Pod ID Number***.*** ***flag*** = data type. ***results*** = array of data.

This is a data reading function. The function call returns an array of the most current analogue data. The type can be specified by the ***flag*** parameter that takes the value NP_PROC, NP_CAL or NP_RAW. Note. NP_PROC and NP_CAL are flags that represent a single precision floating point numbers. The NP_RAW flag represents a 32 Bit integer data type.

The function returns a 0 if the operation is successful or an integer error code >0 if it has failed. **True = 0 False > 0**

Example. Read a single processed data item (engineering units) from each channel within a single instrument. The instrument has a PID number 7265.

```
int pid,flag;                 /*  Assign parameters and data types for PID and flag* /
float results[16];            /*  Assign an array of single floating point numbers to store data */
pid = 7265;           / * Assign instrument PID number. This time the PID is known in advance /
flag = NP_PROC;   /* assign flag  to define data type of values*/

NP_SimpleRead(pid,flag,results);

/* results = {12.53, 3.22, 9.22 .... }
  channel-0=12.53.            The results are in engineering units i.e. Deg C, volts etc.
  channel-1=3.22
```

Example. Read the digital input port levels from each digital channel within an instrument that has a PID number 7265.

```
int pid,flag;             /* Assign parameters and data types for PID and flag* /
long results[16];         /* Assign an array of long integer numbers to store data */
pid = 7265;               /* Assign instrument PID number. This time the PID is known in advance /
flag = NP_DIGITAL;        /* assign flag to define data type of values*/

NP_SimpleRead(pid,flag,results);
      /* results = { 1.0,0.0,1.0,1.0. } channel-0=1.0 The results are in engineering units i.e. Deg C, volts etc. channel-
1=0.0
```

int PODAPI [redacted](int pid,TBufParamStruct* Param);

pid: Pod Identifier number**. TBufParamStruct*** structure used as buffer.

Function returns a pointer of data. This pointer is of **type TbufParamStruc** containing the time and sample number of the start of the data acquisition cycle, the latest sample received, and latest continuous sample. [redacted].

As communication over Ethernet is not reliable, some information may get lost between the instrument sending the data and the host PC used to receive it which may lead to temporary gaps in the data stream. Netpod.dll requests the missing data from the NetPod buffer and re-inserts the missing components at the correct location.

The "latest continuous" sample contained within the **TbufParamStruc** structure contains the latest data item for which the data is continuous between the start sample acquired at the beginning of the data acquisition operations and the latest acquired sample.

The function returns a 0 if the operation is successful or an integer error code >0 if it has failed. **True = 0 False > 0.**

Example. Obtain global configuration parameters for NetPod 7654. Global parameters are sample rate, IP address, Ethernet address etc.

```
int pid;
TBufParamStruct info;
pid = 7654;
NP_GetBufParam(pid,info);
```

The NP_GetBufParam function should be called for every NetPod for which data is to acquired. This action is required as a number of instruments may be running at different rates. Note the parameters StartP (start position of data) can be different for each instrument even if they are running at the same rate. There can be a small drift (difference) between the StartP parameters between instruments running at the same rate which can accumulate over time to cause an error.

int PODAPI[redacted](int pid, int cid, int flag, TSample Start, int Len, float* results);

Copies a channel buffer into the results array starting at the ***StartP*** sample position and continuing for ***length*** samples. This data type is used by the function is defined by the ***flag*** parameter.

pid: POD Identifier number.

cid : **Channel number.** Range 0 – 15 corresponds to the instrument layout.

TSample: Sample position in the data stream from were the earliest data item is read e.g. for 256 samples. The value is taken as the location of the latest sample – 255.

Data Types: Flag is **NP_CAL, NP_PROC** :- Single precision floating point.
Flag is **NP_RAW, NP_DIGITAL** :- 32 bit integer.

The function returns a 0 if ithe operatation is successful or an integer error code >0 if it has failed. **True = 0 False > 0.**

Example. Read 256 samples of the latest data from channel 4 of a Pod using PID 7265. The example demonstrates the use of the GetBufParam and ChannelBufRead functions. Data obtained is in single precision floating point format.

```
int pid,cid,flag,length;     /* Assign parameters types */
float results[256];          /*  Assign data array */
pid = 7265;                  /*  Assign instrument PID */
cid = 4;                     /*  Set Channel 4 as the channel that is going to be read */
flag = NP_PROC;  /*  Use flag to set data type as single precision floating point number */


NP_GetBufParam(int pid,TBufParamStruct* Param);      /* get the latest sample number */
NP_ChannelBufRead(pid,cid,flag,Param.LatestSample-256, 255, results);
                                            /* retrieve 256 samples of this data for channel 4 */
```

Note. When reading digital information you must use a 32 Bit integer array to store the results.

4.5 Example – Acquiring data from a number of instruments

The following examples demonstrates how to gather data from a number of instruments

[redacted]

```
i,pod : Integer;
 nLast,nReadFrames,nStartP : int64;
 Info : TBufParamStruct;
 SPC: TSpectrum;
 results : packed array[0..1023] of single;
 StartSamples : Int64;
begin
  if g_nChan = 0 then
    exit;

  pod := g_aChanList[0].PodID;                       //  Pod ID of Chan 0
  nLast := g_RtBuf.nLastSample;                           // Recently saved Sample number
  NP_GetBufParam(pod,Info);
  nReadFrames := info.LatestSample - nLast;                 // Calculate frames to read

  if nReadFrames > int64(g_RtBuf.uMaxTempFrames) then
    nReadFrames := g_RtBuf.uMaxTempFrames;     // Limit to the buffer size when large allocation
is required.
   StartSamples := info.LatestSample - nReadFrames;
                                       // The start sample becomes the first NetPod standard

  for i := 0 to g_nChan-1 do
  begin
    NP_GetBufParam(g_aChanList[i].PodID,Info);       // Loop for as many channels there
                                                          are in a specified Pod
    nStartP := info.LatestSample - nReadFrames;     // Calculate the starting point within buffer
    [redacted](                                     // Read data
           g_aChanList[i].PodID,
           g_aChanList[i].Channel,
           NP_PROC,
           nStartP,
           nReadFrames,
           @results);
                                                              // Save in memory buffer as well as
HDD

    Rtb_AddTempBuf(g_RtBuf,StartSamples,i,@results, nReadFrames);

// The function above is read Row style and rearranged Matrix style. Then saved in memory

end;
```

Set configuration function details

The following function calls are used to write configuration parameters to the instruments.

Int PODAPI NP_SetDigital(int pid,int value);

pid: POD Identifier number. ***value:*** Binary bit-wise code***.***

This function sets the state of the digital output devices such as relay, TTL output etc. The output port is set by assigning a bit-wise binary number to the ***value*** parameter. For example, to set digital output port 0 to a level 1 (high), ***value*** is set at 0x0001. To set digital port 1 to a high level ***value*** is set to 0x0002. Note. The port values can only be set if the I/O hardware is suitably installed. You cannot set a digital input to a high(1) level if it is not physically able to do so.

The function returns a 0 if the operation completes successfully or an integer error code >0 should it have failed. **True = 0** **False > 0**

Example.

```
int pid,value;
pid = 7654;                     /*  Set instrument PID */
value = 0x0010;                 /*  Hex reporesentation number */
NP_SetDigital(pid,value);       /*  Set port levels */
                                /* sets digital 4 to 1, the rest to 0 */
```

int PODAPI NP_SetStatus(int stat);

This routine is provided for backward compatibility with older software. Supplied for backward compatibility with older software.

NP_SetStatus is no-blocking (returns immediately) and therefore the return code is always successful. To get the status you could use any of the following:

poll using **NP_GetStatus(NP_ISFINUPDATE).** This function will return 0 if updating, 1 when complete.

poll the driver using **NP_GetStatus(NP_ISINITSCAN).** This function will determine if the initial scan has been completed, but not any subsequent scans.

poll using **NP_GetPodList**. This function will return an array of pod ID numbers (null terminated), and therefore you could count the number of pods that have been detected. The pod list will be deleted at start of scan and added to the list as they are detected.

When using the function **NP_SetStatus(NP_SCANNET)** to scan the network for instruments, you must allow a few milliseconds from calling this function before making any subsequent function calls. The driver is multi-treaded and it is another thread that sets function values.

Use function SetStatusExt in future.

int PODAPI NP_SetStatusExt(int stat, int pid, int cid);

This function sets overall system parameters such as those required for starting and stopping acquisition operations, showing the configuration dialogs, and general system interface routines.

The following are valid constants for the ***stat*** parameter:

NP_STARTRUN	Start data acquisition operations.
NP_STOPRUN	Stop data acquisition operations.
NP_TOGGLERUN	Toggle start/stop data acquisition operations.
NP_SHOWCFGDLG	Show the configuration dialog (main screen) – Podmng.exe program.
NP_HIDECFGDLG	Hide the configuration dialog – Podmng.exe program
NP_SHOWCFGDLG2	Show configuration dialog as topmost window – Podmng.exe program
NP_RUNPODMNG	Run pod manager (taskbar program).
NP_SETUPCHAN	Show the channel configuration dialog window – Podmng.exe program
NP_SETUPPOD	Show the Netpod configuration dialog – Podmng.exe program.
NP_SCANNET	Scan the network – Required for all comms. interfaces
NP_SETCALLBACK	Set the main DLL callback (not generally used in application programs)

The function returns a 0 if it operates successfully or an integer error code >0 if it has failed.
True = 0 False > 0

Example. Show the channel configuration details for Channel 4 within instrument an instrument identified with PID of 7654. The dialog window that appears is called from the podmng.exe software.

NP_SetStatusExt(NP_SETUPCHAN,7654,4); /* show the channel configuration
dialog for channel 4 of NetPod 7654 */

int PODAPI NP_PutPodInfo(int pid, TPodInfoStruct Info);

Parameters: ***pid*** = Pod ID Number. **info** = Structure containing global instrument settings.

This function writes global configuration details to an instrument with identification ***pid***. The global information has to be placed into the structure ***info*** prior to it being written to the instrument.

The string parameters within the TPodInfoStruct structure have no effect in this function call as only the binary form of the data is used.

The function returns a 0 if the operation is successful or an integer error code >0 if it has failed. **True = 0 False > 0.**

int PODAPI NP_PutChannelInfo(int pid, TChannelInfoStruct Info);

Parameters: ***pid*** = Pod ID Number. **info** = Structure containing channel settings.

This function is used to set NetPod channel settings. Information to be sent to a channel has to be first stored at the correct location within the TChannelInfoStruct variable ***info*.** The string parameters with **info** have no meaning as only the binary form of the data items is used.

The function returns a 0 if the operation is successful or an integer error code >0 if it has failed. **True = 0 False > 0.**