AGWPE TCP/IP API Tutorial

by Ing. Pedro E. Colla (LU7DID) and George Rossopoulos (SV2AGW)

Last Revised December 12, 2000

Abstract

The AGW Packet Engine (***AGWPE***) by George Rossopoulos (***SV2AGW***) is a powerful AX.25 Layer 2 Manager running under Windows 95/98/NT as a "state-of-the-art" 32 bits application.

At this time ***AGWPE*** supports two completely different APIs; one based in DDE (Dynamic Data Exchange) which is the first implemented and another based on TCP/IP which has been introduced recently.

This document would concentrate on the TCP/IP API, the intended audience for it are application programmers looking for information on how to write (or adapt) their programs to take advantage of the ***AGWPE*** services.

The information provided in this document is valid as per ***AGWPE*** Version 2000.20 or higher; previous versions might not support some of the functions so checking the release information is necessary; it's likely that future versions would be backward compatible unless documented otherwise.

-

Table of Contents

Overview

As an AX.25 Layer 2 (L2 for short) Manager it could control a huge number of AX.25 devices such as many TNC models (most of the commonly used), BayCom modems (most incarnations), quite a few really specialized high speed modems and the SoundBlaster card as a Packet device, ***AGWPE*** also provides an special “internal” port called loopback that could be used to interchange information among different applications running under the same ***AGWPE*** or (very useful) for test purposes; moreover, an almost unlimited number of them could be used at the **same** time each one being a “**port**” (well, sort of, George claims a maximum number of 100 ports, which is “unlimited” under every stretch of the concept).

As a manager, ***AGWPE*** is not functional per-se, meaning, the end user **need** to have it loaded but doesn’t make any direct **use** of it other than to configure it or to get a glimpse of the current status of the different AX.25 links and ports.

What uses the AGW Packet Engine are **applications** enabled to talk with it which in turn are used by end users to sustain activity over Packet Radio.

AGWPE comes with a basic “suite” of applications comprising a Packet Terminal program (AGWTerm), a monitor program (AGWMonitor), a mail client (AGWBBS/AGWFWD), a cluster program (AGWCLU) and a digipeater (AGWDigipeater), all of them written by George (***SV2AGW***); this suite is a complete albeit somewhat limited set for any end-user to sustain Packet operations.

A growing number of third party applications are starting to support the ***AGWPE*** also either directly (i.e.WinPack) or thru additional libraries (Tsthwin, WinFBB,etc); a fair number of authors had announced the future support of this platform with new versions of their programs.

The applications would see the ***AGWPE*** as a provider of services, those services are accessed thru a set of conventions named collectively the **Application Program Interface (API)**.

The application request services thru blocks of information called API Frames (or Frames, for short, but don’t confuse them with AX.25 L2 Frames), those blocks are just a chunk of data with a predefined length

and contents.

The frames are always composed by a section named header (36 bytes long) and depending on the action required another section named data (any length).

Frames could be generated for the application and sent to ***AGWPE*** to request an specific service (such as sending data or to configure a particular aspect of the ***AGWPE*** functional behaviour or to require information about the current status).

AGWPE could, in turn, send also frames to the application; either as an answer to a given service (i.e. query of some value) or as an unsolicited block of information (i.e. a block of data just received at some port).

Both the frames sent to ***AGWPE*** by the application and the frames sent by ***AGWPE*** to the application has the same format.

Communicating with AGWPE using the TCP/IP API

Enabling

The TCP/IP API must be enabled to be functional, the default API for AGWPE still is the old DDE based.

There are basicall two ways to enable the TCP/IP API, thru the AGWPE GUI or modifying the AGWPE.INI configuration file.

Both methods would be useful, the AGWPE GUI for manual configurations while the AGWPE.INI modification could be seen more appropriate for automatic setups.

AGWPE GUI

Upon loading click on the AGWPE icon at the task bar, go to the “Setup Interfaces” entry and on the “WinSock Interface Security” tab be sure the menu item “Enable Winsock TCP/IP Application Interface” is checked, verify which is the TCP Port where AGWPE listen for applications (should be 8000 unless you changed it).

Once checked the change would be Accepted to be effective.

AGWPE.INI

The following entry has to be added on the configuration file ***AGWPE***.INI (usually at the same directory than the executable).

```
[TCPIPINTERCONNECT]
ENABLE=1
```

Without this entry ***AGWPE*** will not operate with the TCP/IP API, it is recommended that an application program willing to use the TCP/IP API should check this configuration value to ensure it is set properly and either set it directly or provide instructions to the end-user on the need to set it as a part of the installation.

AGWPE doesn’t provide a way to activate the TCP/IP API thru any of the GUI dialogs, so the entry on the *AGWPE.INI* must be configured and it is the only way to activate the TCP/IP API.

AGWPE reads this information only at startup, so for any change to be made effective the program has to be stopped and started.

Communications

When starting with this configuration ***AGWPE*** starts to serve the TCP/IP port 8000 for incoming requests from applications, see the previous section (AGWPE GUI) on how to change it..

Every application would start as many TCP/IP connections (sockets) with the ***AGWPE*** as required (usually one will be enough), multiple applications could have open connections with ***AGWPE*** at the same time; the limit on the number of sockets or connections ***AGWPE*** could sustain is defined by the TCP/IP stack of the machine where ***AGWPE*** is running (every socket "tax" the system resources, mostly in terms of memory and CPU cycles, till eventually no additional sockets could be opened).

On machines with a modern configuration this limit is not easily achievable under practical uses; ***AGWPE*** itself is extremely efficient in terms of the memory used and CPU cycles taken by itself([1]).

Each application is a typical TCP/IP client, as usually referred to in the bibliography, while ***AGWPE*** itself is a TCP/IP server.

In order to open a socket the application should start a TCP/IP connection to the IP address of the machine where ***AGWPE*** is running and the TCP Port 8000.

One of the very powerful aspects of the TCP/IP API is the fact that no restriction bounds the application and ***AGWPE*** to be run on the same machine, as long as a TCP/IP connection could be established the ***AGWPE*** and the Application program could be run on the same machine, on close machines operating in some LAN or half a world appart.

The traffic (amount of data transferred between ***AGWPE*** and the Application) is quite substantial indeed, in order to achieve reasonable performance the bandwidth between ***AGWPE*** and the Application (each application) should be in the order of 3 to 4 times the combined bandwidth of all the AX.25 ports being serviced, this could be served in excess either running ***AGWPE*** and the application on the same machine or being linked by some Ethernet or TokenRing LAN (typical speeds between 10 and 100 Mbps), dial-up connections might be marginal depending on the total load planned to be serviced([2]).

An interesting (theoretical) possiblity is to run ***AGWPE*** and an application program being appart and linked thru Packet Radio (TCP/IP over AX.25), albeit most current networks won't have enough speed to provide even a minimum functionality in real world terms.

In order for the TCP/IP communication to be established the IP Address of the machine where ***AGWPE*** is running is assumed to be known (this is a pre-requisite), the TCP port is as stated usually 8000.

The IP address of the machine could be easily obtained with the Windows utility named ***winipcfg***, in case the machine has many adapters (each one eventually having one different IP address) you could use the one associated with the adapter that could "see" the machine running ***AGWPE***.

In the case the application and ***AGWPE*** runs on the same machine the definition of the IP address becomes trivial since the IP loopback address (***127.0.0.1***) should be always used; since most of the time ***AGWPE*** and the application will be run on the same machine the loopback IP address should be the one used by the application by default[3]

API Application Interface Security

Starting on version 2000.78 AGWPE brings security features that must be taken into account, the behaviour of the security model is controlled by the settings at the "WinSock Interface Security" tab on the "Interface Setup" menu entry.

- AGWPE allows applications to be:
    - Local only (entry "Accept only from MyComputer".

    - Intranet only (entry “Accept Only From MyLAN (standard)”.
    - Internet (entry “Accept from Anywhere”).
- Applications could override the default security setting as stated above by means of sending an special “login” frame.

Be aware that if the application is being ran from a machine that doesn’t comply with the security setting it has to provide a frame of type “P” to be able to interact with
AGWPE, more on this later.

This security model allows flexibility to make visible a node to a big (uncontrolled) environment such as the Internet and still enable the system operator to control who
is using his resources.

TCP/IP Exchange

The basics on how to program using TCP/IP are far beyond the scope of this document to explain, however, TCP/IP is a technology so pervasive and widely used that no modern language intended for the Windows environment lacks support for it.

Usually this support is in the form of a set of calls (the TCP/IP API or the programming conventions to use TCP/IP, do not get confused with the ***AGWPE*** TCP/IP API which is the way to communicate with ***AGWPE*** using TCP/IP).

The implementation varies from programming language to programming language, and even within them there are often many alternative implementations based on different vendors.

TCP/IP programming could be a mind boggling exercise at its limits, fortunately only a subset of all the functions are needed to establish and maintain a successful connection with ***AGWPE*** under most circumstances.

Communications with ***AGWPE*** will use TCP sockets only, so on most implementations of TCP/IP you would require to use just 3 API calls:

- A call to *OPEN* a socket.
- A call to *SEND* information thru an opened socket (usually a binary block of data).
- A call to *CLOSE* the socket upon termination.

Your program would also need to handle a minimum of 3 events related to an open socket:

- An event confirming the socket has been opened.
- An event informing when some error occurs.
- An event informing data had arrived from ***AGWPE*** and it’s available for processing (usually a block of data).

Depending on the language and the library the above basic elements might vary, through this document all techniques would be explained conceptually based on this set.

The sequence of a dialog between an application and ***AGWPE*** always steps thru the following major activities:

- A TCP/IP socket is established with ***AGWPE***, errors in this process must be handled.
- Some initial interchange of information with ***AGWPE*** in order to get or set configurations.
- The AX.25 activity itself (send and receive data+status).

- Some final configuration clean-up between the application and ***AGWPE***.
- The TCP/IP socket is closed.

AGWPE API Reference

Frame Structure

Information between ***AGWPE*** and the application flows in both directions using an overall format composed by a header (fixed) and a variable data area depending on the particular frame being sent (many frames are just formed by a header).

Field	Length	Meaning
AGWPE Port	1 Bytes	[0..n] the least significant value comes in the first byte while the most significant in the second. I.E. Port 2 would be expressed as 0x01 ([4])
Reserved	3 Bytes	Usually 0x00 0x00 0x00
DataKind	1 Byte	Is the frame code, reflects the purpose of the frame. The meaning of the DataKind DO VARY depending on whether the frame flows from the application to ***AGWPE*** or viceversa.
Reserved	1 Byte	Usually 0x00
PID	1 Byte	Frame PID, it's usage is valid only under certain frames only. Should be 0x00 when not used.
Reserved	1 Byte	Usually 0x00
CallFrom	10 Bytes	CallSign FROM of the packet, in ASCII, using the format {CALLSIGN}-{SSID} (i.e. ***LU7DID***-8) it is "null terminated" (it ends with 0x00). ([5]) The field ALWAYS is 10 bytes long. It's filled on packets where it has some meaning.
CallTo	10 Bytes	CallSign TO of the packet, same as above.
DataLen	4 Bytes	Data Length as a 32 bits unsigned integer. If zero means no data follows the header.
User (Reserved)	4 Bytes	32 bits unsigned integer, not used. Reserved for future use.

All reserved fields must not be used by application programs in any form, they should be initialized to binary zeros (0x00) on frames sent by the application to ***AGWPE***, undefined values could be present on frames sent by ***AGWPE*** to the application on those frames.

AGWPE is fairly tolerant on unused fields (either reserved or not used on a particular frame format) to held almost anything, so the need for proper initialize them reflected on this documentation aims towards proper programming practices rather than actual needs from ***AGWPE***.

Frames sent from the Application to AGWPE where the CallFrom/CallTo values are relevant must

contain our callsign in the CallFrom and the other end callsign in the CallTo fields. The other way around, frames from AGWPE to the Application would contain the other end callsign+SSID in the CallFrom and our callsign+SSID in the CallTo fields.

When the frame has data associated with it (DataLen <> 0) the bytes up to the number expressed by DataLen follows inmediately after the last byte of the header.

In order to allow for a fully transparent transport of data no delimiters of any kind are used on the data area, so binary information of any kind could be effectively transported.

A typical example of a frame (header+data) sent by ***AGWPE*** looks like this:

```
|01 00 00 00 4D 00 CF 00 4C 55 37 44 49 44 2D 34 |....M...LU7DID-4
|00 00 4E 4F 44 45 53 00 00 00 00 00 07 00 00 00 |..NODES.........
|00 00 00 00 FF 41 42 52 4F 57 4E -- -- -- -- -- |.....ABROWN
```

An example of a frame with just a header sent by ***AGWPE*** looks like this:

```
|00 00 00 00 58 00 00 00 4C 55 37 44 49 44 2D 34 |....X...LU7DID-4
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

It's worth to notice that in the second example not all fields (actually the ones not relevant to the function requested as we would see) have not been completed.

Frames sent by the Application to AGWPE

Application might (or must, sometimes) send data to ***AGWPE*** in order to retrieve configuration information or to sustain communication over any of the ports.

Colloquially, the frames are identified by it's **DataKind** (so a frame with a DataKind='X' is referred in this documentation as an 'X' frame).

The same DataKind could be used on a frame sent by the application to ***AGWPE*** or from ***AGWPE*** to the application (however, the meaning of a DataKind is unique in any given direction), usually the frames with the same DataKind on both directions are Query-Answer pairs (so, i.e., a 'G' frame sent by the application is replied by ***AGWPE*** with a 'G' frame filled with the information required).

Care has to be taken by the application program to handle sent and received frames separately.

Follows all the frame formats supported for the application to send to ***AGWPE***.

Application Login ('P' frame)

An application needs to login when the "WinSock Interface Security" setting rules doesn't allow the machine where the application is being ran to access the AGWPE directly; it should not bother applications running on the same machine where AGWPE is executing. Still applications should allow the user to define this security setting and be flexible to be run on machines other than the one AGWPE is running (and thus, potentially not enabled directly by the security settings to access AGWPE).

This frame is mandatory when the application is being run from a machine that doesn't comply with the

security rules, without it AGWPE will not accept nor send frames
to the application.

The login is made with a frame with an empty header and the login data into the Information part of the frame with the following format.

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘P’ (ASCII 0x50)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	Length of User+Password including 0x00’s
User & Password	N Bytes	UserId ended with 0x00 filled till 255 bytes Password ended with 0x00 filled with 255 bytes

AGWPE do not inform the application about the success.

Please note an application must send one of the UserId/Password combinations as stated in the “WinSock Interface Security” tab “User Setup” section in order to be validated and allowed by AGWPE to interact with it thru the WinSocks API.

Follows a sample content of the information area of the frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|4C 55 37 44 49 44 00 00 00 00 00 00 00 00 00 00 |LU7DID..........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4C |...............L
|49 5A 41 52 44 00 00 00 00 00 00 00 00 00 00 00 |IZARD...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
```

```
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 -- -- |..............
```

Register CallSign (‘X’ frame)

An application needs to register at least one callsign with ***AGWPE*** as a pre-requisite to be able to send data thru any AX.25 port or to sustain any connection and before any attempt on doing so.

To receive (monitor) information heard at the different ports the “m” frame should be used instead.

There is no limits on the number of callsigns that could be registered by a single application, each registration would require a separate frame.

When an application registers a callsign ***AGWPE*** “listen” on the radio ports for any packet frame directed to that callsign and when detected it would be sent to the application using the suitable frame format (depending on the type).

The registration is made with a frame with just a header (no data) with the following format.

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘X’ (ASCII 0x58)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	CallSign-SSID to register
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

A given callsign and SSID combination is allowed to be registered just once by an application (actually among all the applications connected to the same ***AGWPE*** at any given moment).
AGWPE informs the application about the success (callsign+SSID registered) or failure (callsign+SSID already in use) by means of an “X” frame sent to the application in response of this one.

Please note an application could register almost “anything” as a callsign+SSID (not necessarily a true callsign), so if for some reason is relevant to the application to receive frames directed to (i.e.) the “NODES” destination that could be accomplished registering the “NODES” callsign, in a way that any frames (likely UI frames) directed to the “NODES” destination (NODES-0 actually) would be directed to the application who registered it. As in with the case of true callsigns a given “destination”+SSID is allowed to be registered just once.

A registration could be performed at any time by the application.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 58 00 00 00 4C 55 37 44 49 44 2D 34 |....X...LU7DID-4
```

```
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Unregister CallSign ('x' frame)

This is the opposite function than to register a callsign, it means the callsign and SSID combination is not longer used by the application and it's free for further use, from the moment of the application become unregistered and till it's registred again all activity heard by ***AGWPE*** on the AX.25 ports directed to that callsign is ignored.

Also, all information sent by the application to ***AGWPE*** involving the unregistered callsign is ignored.

The overall format is very similar to the registration frame, just the DataKind is changed, as follows:

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'x' (ASCII 0x78)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	CallSign-SSID to unregister
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

As a difference with the registration frame the application should not expect any answer from ***AGWPE*** as a confirmation of the successful unregistration.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 78 00 00 00 4C 55 37 44 49 44 2D 34 |....x...LU7DID-4
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Ask Port Information ('G' frame)

Using this frame the application could query ***AGWPE*** to provide information about the currently defined ports.

This information is usually handy at the start of the application program in order to know the number of ports available and eventually use that information for functional or presentation purposes, the port information could not be changed dynamically on ***AGWPE*** (it requires ***AGWPE*** to be stopped and re-started) so this information should also be queried every time the TCP/IP connection is re-established.

The frame format comprises a header only with the following information.

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'G' (ASCII 0x47)
Reserved	1 Byte	0x00
PID	1 Byte	0x00

Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE answer this request with a “G” frame.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 47 00 00 00 00 00 00 00 00 00 00 00 |....G...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Enable Reception of Monitoring Frames (‘m’ frame)

In order for monitoring frames to be sent to the application this condition has to be signaled to ***AGWPE*** using this frame.

From the moment this frame is sent activity at all ports would be made available to the application (Frames S,I and U).

This function could be used even if the application didn’t registered any callsign.

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘m’ (ASCII 0x6D)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE didn’t confirm specifically this frame, however, the flow of monitored information should start inmediately after it has been sent by the application.

This frame acts like a switch, the first time issued it enables the reception of monitoring frames while the second disables it and so on; in general on odd times it would enable and on even times it would disable.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 6D 00 00 00 00 00 00 00 00 00 00 00 |....m...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

AGWPE Version Info ('R' frame)

It's sometimes important (at least it's is a good programming recommended practice) to care about the level of the ***AGWPE*** which the application is connecting to.

Several reasons support that practice, but the most important is to be sure the ***AGWPE*** will support all the frames and functions the application program would require to work properly; as a fast evolving platform ***AGWPE*** is being continuously upgraded with new functions and fixes for old ones.

The application programmer should not be surprised to find almost all version historically released of ***AGWPE*** thru the time, not all of them supporting the full set of frames documented here (which are valid as per version 2000.20 or higher).

The ***AGWPE*** version is queried with a frame with the following format:

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'R' (ASCII 0x52)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

In any case, this frame should be sent at least once per execution by the application program (even if the ***AGWPE*** connection could be stopped and restarted it's not unreasonable to assume the version didn't changed, doesn't hurt to query and confirm the version on each connection with ***AGWPE*** though).

This frame is answerd by ***AGWPE*** with an 'R' frame.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 52 00 00 00 00 00 00 00 00 00 00 00 |....R...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Ask Port Capabilities ('g' frame)

An useful complement of the "G" frame (Ask Port Information) is to query ***AGWPE*** about the particular configuration for every specific port.

Albeit ***AGWPE*** doesn't allow an application to change its configuration thru the API it's usually necessary or useful to get that information anyway for (mostly) presentation purposes.

This frame has the following format

Field	Length	Meaning

AGWPE Port	1 Bytes	Port to query 0=Port1,1=Port2,…
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'g' (ASCII 0x6D)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE answers this request with a "g" frame.

Some values are static and could not be changed without re-starting ***AGWPE***, but others reflects dynamically the current status of a given port in terms of traffic.

This function should be called at least once every time a connection with ***AGWPE*** is established, there is no limit on how many times this information could be queried, however, a practical limit from the performance (and usefulness) standpoint should limit this query to be performed once every minute or so.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 67 00 00 00 00 00 00 00 00 00 00 00 |....g...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Callsign Heard on a Port ('H' frame)

A very useful service required (or nice to have) on most applications is a list of the stations "heard" on a given port; this could be achieved by the application just collecting monitoring information.

However, this is not required since ***AGWPE*** helds such a list and makes it available to the application upon request (at any time).

In order to request the updated list of stations heard on a given port the following frame has to be sent.

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to query 0=Port1,1=Port2,…
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'H' (ASCII 0x48)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

This function makes ***AGWPE*** to answer the Heard information thru an "H" frame.

This frame could be sent as many times as required during the lifespan of a connection, every time the information provided will be updated to reflect the traffic actually heard.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 48 00 00 00 00 00 00 00 00 00 00 00 |....H...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Ask Outstanding frames waiting on a Port (‘y’ Frame)

This frame could be used by the application at any time to query ***AGWPE*** about the number of frames (from all sources, not only this application) that are queued and waiting to be transmitted by ***AGWPE*** thru a given port.

This would be useful to regulate the rate used to send information to ***AGWPE*** and to keep it realistic with the actual bandwidth of the destination port.

The information could be queried using the following frame:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to query 0=Port1,1=Port2,…
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘y’ (ASCII 0x79)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

This frame is answered by ***AGWPE*** with an ‘y’ frame.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 79 00 00 00 00 00 00 00 00 00 00 00 |....y...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Ask Outstanding frames waiting for a connection (‘Y’ frame)

This frame could be used with similar purposes than the ‘y’ frame but to query ***AGWPE*** about the outstanding frames waiting sourced on a given (and specific) connection as opposed to the overall activity of a port without any clue on how it had been sourced.

The information could be queried using the following frame:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to query 0=Port1,1=Port2,…
Reserved	3 Bytes	0x00 0x00 0x00

DataKind	1 Byte	'Y' (ASCII 0x59)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00)
CallTo	10 Bytes	Other CallSign-SSID i.e. ***SV2AGW***-14 ended with null (0x00)
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE would answer this frame with a 'Y' frame himself if the connection referred by the CallFrom/CallTo fields do exists currently.

Careful must be exercised to fill correctly both the CallFrom and CallTo fields to match the ones of an existing connection, otherwise ***AGWPE*** won't return any information at all from this query.

The order of the CallFrom and CallTo is not trivial, it should reflect the order used to start the connection,
so

- If we started the connection CallFrom=US and CallTo=THEM
- If the other end started the connection CallFrom=THEM and CallTo=US

Please refer to the 'C' frame sent by ***AGWPE*** upon connection to understand how to identify who initiated a connection.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 59 00 00 00 4C 55 37 44 49 44 2D 34 |....Y...LU7DID-4
|00 00 4C 55 37 44 49 44 00 00 00 00 00 00 00 00 |..LU7DID........
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Send UNPROTO Information ('M' frame)

This frame could be used by the application when an AX.25 unproto (UI) frame must be sent by the application.

For an application to send unproto information no registration is needed, however unproto information heard on the ports directed to it won't be made available by ***AGWPE*** and information exchange won't be possible (unless the application extract frames directed to it thru the inspection of monitoring frames, which is not utterly practical but still possible).

Typical uses for an unproto frame are beacon or any other broadcast message, it's also widely used by NETROM L3 broadcast, TCP/IP over AX.25 and the FBB mail client protocol among others.

In order to send an unproto frame the header to be used is

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the unproto frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00

DataKind	1 Byte	'M' (ASCII 0x4D)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID 0x00 or 0xF0 for AX.25 0xCF NETROM and others
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used
CallTo	10 Bytes	Destination of the unproto frame. Not necessarily a callsign+SSID (could be i.e. CQ, ID, another callsign+SSID), etc…
DataLen	4 Bytes	Number of Bytes to be sent
User (Reserved)	4 Bytes	0

Following the header the (exact) amount of bytes indicated in DataLen should follow.

AGWPE would indirectly inform the success of the unproto send thru both an 'I' frame, a 'U' frame and a 'T' frame (if monitoring is enabled thru the 'm' frame).

Even if ***AGWPE*** handles AX.25 frames larger than 255 bytes not so many other programs could over the air, so it's a reasonably programming practice to ensure than the length of the data to be transferred is equal to or less than 255 bytes.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 4D 00 F0 00 4C 55 37 44 49 44 2D 34 |....M...LU7DID-4
|00 00 4E 45 54 4D 45 00 00 00 00 00 39 00 00 00 |..NETME.....9...
|00 00 00 00 0D 0A 42 65 61 63 6F 6E 20 64 65 20 |......Beacon de
|4E 6F 64 6F 20 4C 55 37 44 49 44 2D 34 20 41 64 |Nodo LU7DID-4 Ad
|72 6F 67 75 65 20 42 41 20 41 72 67 65 6E 74 69 |rogue BA Argenti
|6E 61 20 5B 47 46 30 35 54 45 5D 0D 0A -- -- -- |na [GF05TE]..
```

Connect, Start an AX.25 Connection ('C' frame)

This frame is sent to ***AGWPE*** when an AX.25 connection with other station is required.

The station originating the connection (CallFrom) **must had been previously registered** with ***AGWPE*** ('X' frame) for the connection to be successfully established.

The connection started with this frame would always be a normal AX.25 connection (information frames with PID=0xF0).

The application is responsible to identify the port to be used for this connection and to properly inform it on the frame.

The format of the frame follows:

Field	Length	Meaning

AGWPE Port	1 Bytes	Port to send the connection request frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'C' (ASCII 0x43)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID (0xF0 or 0x00)
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used. must had been previously registered
CallTo	10 Bytes	Destination callsign+SSID of the connection.
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE would start inmediately to connect the destination callsign-SSID (it could be monitored, if monitoring has been enabled) thru the 'S' frames.

Upon connection or failure the application would receive a 'C' frame or a "Retryout message".

An application could sustain one connection per distinctive callsigns+SSID for both origin and destination pairs (only one connection by a given callsign+SSID on origin and destination is allowed by the AX.25 protocol).

No practical limit do exists on the number of connections an application could sustain with different destinations, even from the same originating callsign+SSID. This concept, of course, is extended when many callsigns+SSID are registered by the same application.

It is an application duty, as we'll see on the relevant frames sent by ***AGWPE***, to discriminate among data coming from diferent connections.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 43 00 00 00 4C 55 37 44 49 44 2D 34 |....C...LU7DID-4
|00 00 4C 55 37 44 49 44 00 00 00 00 00 00 00 00 |..LU7DID........
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Send Connected Data ('D' frame)

Once a connection had been successfully established data could be exchanged, the application could then send data to the other end by means of data frames.

The format of the frame would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the data frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'D' (ASCII 0x44)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID (0xF0 or 0x00)
Reserved	1 Byte	0x00

CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used. must had been previously registered
CallTo	10 Bytes	Destination callsign+SSID of the connection.
DataLen	4 Bytes	Number of Data Bytes to be transferred.
User (Reserved)	4 Bytes	0

Following the header the (exact) amount of bytes indicated in DataLen should follow.

AGWPE would indirectly inform the success of the unproto send thru both an 'I' frame, a 'U' frame and a 'T' frame (if monitoring is enabled thru the 'm' frame).

Data exchanged would be under a standard AX.25 Information PID (0xF0) unless the connection had been specifically started signalling ***AGWPE*** about a non-standard PID (connection started with the 'c' frame instead of the 'C' frame), on such situations the application must place the relevant PID on the respective field of the header. In all other situations the PID field is ignored by ***AGWPE*** and 0xF0 is used instead.

If a 'D' frame is sent by the application without an established connection the frame is ignored by ***AGWPE***.

It is the application responsibility to keep using the proper AGWPort and CallFrom/CallTo values on all the frames of a given connection than the used to establish it.

Even if ***AGWPE*** handles AX.25 frames larger than 255 bytes not so many other programs could over the air, so it's a reasonably programming practice to ensure than the length of the data to be transferred is equal to or less than 255 bytes.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 44 00 F0 00 4C 55 37 44 49 44 00 00 |....D...LU7DID..
|F0 15 4C 55 37 44 49 44 2D 34 00 00 02 00 00 00 |..LU7DID-4......
|A8 6D 45 00 3F 0D -- -- -- -- -- -- -- -- -- -- |.mE.?.
```

Disconnect, Terminate an AX.25 Connection ('d' frame)

When an AX.25 connection (started with a 'C' frame) needs to be terminated a disconnection frame must be sent by the application.

To send data between a connection and a disconnection is, of course, optional; however, the main purpose of a connection would be most of the time to exchange data with another station.

The format of the frame would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the data frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'd' (ASCII 0x64)
Reserved	1 Byte	0x00

PID	1 Byte	AX.25 PID (0xF0 or 0x00)
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used. must had been previously registered
CallTo	10 Bytes	Destination callsign+SSID of the connection.
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

No data is associated with this frame.

AGWPE will inform the completion of this request thru a 'd' frame[6]

If a 'd' frame is sent by the application without an established connection the frame is ignored by ***AGWPE***.

It is the application responsibility to keep using the proper AGWPort and CallFrom/CallTo values on all the frames of a given connection than the used to establish it.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 64 00 00 00 4C 55 37 44 49 44 2D 34 |....d...LU7DID-4
|00 00 4C 55 37 44 49 44 00 00 00 00 00 00 00 00 |..LU7DID........
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Connect VIA, Start an AX.25 circuit thru digipeaters ('v' frame)

This frame is used with similar purposes than the 'C' frame, but as it creates a "direct" connection this frame must be used when intermediate AX.25 digipeaters must be used to establish a connection.

It's the application responsibility to determine, based on it's functionality and user interface, whether a given connection should be started direct ('C' frame) or thru digipeaters ('v' frame).

Once the connection is established data is transferred between both ends with the same frame ('D' frame) and disconnection is started also with the same frame ('d' frame) on both.

The format of this frame would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the data frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'v' (ASCII 0x76)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID (0xF0 or 0x00)
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used. must had been previously registered
CallTo	10 Bytes	Destination callsign+SSID of the connection.
DataLen	4 Bytes	Length of the VIA information

User (Reserved)	4 Bytes	0

The VIA (number and sequence of digipeaters to be used) is informed in the data part of the frame inmediately following the header, the length of this area would vary depending on the number of digipeaters to be used, the exact length must be informed in the DataLen field.

The data area must contain the VIA information in the following format

Offset	Length	Meaning
+00	1 Bytes	Total number of digipeaters to be used (max 7)
+01	10 Bytes	CallSign+SSID of the first digipeater ended with null (0x00)
+11	10 Byte	CallSign+SSID of the 2nd digipeater ended with null (0x00)
……	……	…..
+10XN+1	10 Byte	CallSign+SSID of the Nnd digipeater ended with null (0x00)

Of course, only the number of needed digipeaters has to be informed (but at least ONE must be informed, otherwise a direct connection should be used instead).

The successful completion of the connection is informed by ***AGWPE*** thru the 'C' frame.

Send UNPROTO VIA ('V' frame)

When the application needs to send unproto information (as in the 'M' frame) but using a chain of repeaters to do so this frame format should be used instead.

The frame format is as follows:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the unproto frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'V' (ASCII 0x76)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID (0xF0 or 0x00)
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used. must had been previously registered
CallTo	10 Bytes	Destination callsign+SSID or ID of the unproto frame (i.e. CQ, ID, MAIL, etc)
DataLen	4 Bytes	Length of the VIA information and the data to be sent
User (Reserved)	4 Bytes	0

Right after the header the chain of digipeaters to be used is sent as the first part of the data area using the format already discussed for the Connect VIA ('v' frame)

Offset	Length	Meaning
+00	1 Bytes	Total number of digipeaters to be used (max 7)
+01	10 Bytes	CallSign+SSID of the first digipeater ended with null (0x00)

+11	10 Byte	CallSign+SSID of the 2nd digipeater ended with null (0x00)
……	……	…..
+10XN+1	10 Byte	CallSign+SSID of the Nnd digipeater ended with null (0x00)

As before, only the number of digipeaters to be used needs to be included (with at least one being informed).

After the VIA information the actual data to be sent is included, please note the DataLen field on the header should reflect the exact size of both the VIA information and the data information to be sent.

Even if ***AGWPE*** handles AX.25 frames larger than 255 bytes not so many other programs could over the air, so it's a reasonably programming practice to ensure than the length of the data to be transferred is equal to or less than 255 bytes.

However, nothing prevents the SUM of the VIA information and the data information to be sent to be larger than 255 bytes (the VIA information is internally decoded and used by ***AGWPE*** to build the AX.25 header but only the data information is included on the AX.25 information part).

Non-Standard Connections, Connection with PID ('c' frame)

On special situations the application might need to interchange information with a destination thru frames using a non-standard PID (standard AX.25 PID for Information Frames is 0xF0), examples of such a need are NETROM connections and frames related to a connected TCP/IP over AX.25 circuit.

On such ocassions ***AGWPE*** must be signaled of this singularity starting the connection with this frame instead of a "normal" 'C' frame as documented before; the application is also responsible to fill the PID field of the header of all data frames sent during the connection with the appropriate value (even if the PID is informed during the connection it has to be "repeated" on every data frame).

The format of the frame would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the connection request frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'c' (ASCII 0x63)
Reserved	1 Byte	0x00
PID	1 Byte	Non Standard PID to use
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Our CallSign-SSID i.e. ***LU7DID***-11 ended with null (0x00) used. must had been previously registered
CallTo	10 Bytes	Destination callsign+SSID of the connection.
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE would start inmediately to connect the destination callsign-SSID (it could be monitored, if monitoring has been enabled) thru the 'S' frames.

Upon connection or failure the application would receive a 'C' frame or a "RETRYOUT" message.

An application could sustain one connection per distinctive callsigns+SSID for both origin and destination pairs (only one connection by a given callsign+SSID on origin and destination is allowed by the AX.25 protocol).

No practical limit do exists on the number of connections an application could sustain with different destinations, even from the same originating callsign+SSID. This concept, of course, is extended when many callsigns+SSID are registered by the same application.

It is an application duty, as we'll see on the relevant frames sent by ***AGWPE***, to discriminate among data coming from diferent connections.

Please note that the destination application must know exactly how to handle data frames with non-standard PID in order for a data exchange to take place, the connection would succeed even with destinations not truly aware of the non-standard PID (the AX.25 protocol doesn't include PID information on a connection frame).

Send Data in “raw” AX.25 format (‘K’ frame)

On special situations when the application needs to control the exact content of a given frame (as when applications needs to deal with a hardware TNC in KISS mode) the complete frame could be built and sent using this frame.

This facility should be used on very special applications only.

The format of the frame follows:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port to send the data frame thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘K’ (ASCII 0x4B)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00 Actual origin is stated in the raw frame
CallTo	10 Bytes	10 0x00 Actual destination is stated in the raw frame
DataLen	4 Bytes	Number of Data Bytes to be transferred.
User (Reserved)	4 Bytes	0

The complete frame (AX.25 header followed by data if applicable) in raw format must follow as data, the exact length of it must be reflected on the DataLen field.

Full knowledge of the intricancies of the AX.25 must be mastered by the brave programmer trying to use this frame, for what is worth the heartfull recommendation is to try to identify other frames or combination of frames most suitable for a given purpose, so use this frame format as an absolute last resort.

For those brave souls in need to still use it the following recommendations should be used (refer to the AX.25 Protocol documentation for the naming conventions).

Field	Length	Description
Flag	1 byte	Will not be the standard 0b01111110 flag but the “TNC” to use 00=Port 1 16=Port 2 …
Address	112/360 bits	AX.25 coded Origin, Destination and (optionally) digipeaters.
Control	1 byte	AX.25 Control Field
PID	1 byte	AX.25 PID
Info	N bytes	AX.25 Information Area

Please note than the AX.25 FCS and the ending Flag are NOT included. No KISS escape codes nor bit stuffing is required to be performed (***AGWPE*** would add them as needed).

As per ***AGWPE*** Version 2000.20 this frame should be used to send **only unproto information**. This is a general recommendation, still it could be used to send both connected
and unconnected information. When connected information is sent using this frame the application will not receive monitor frames (T frames) with the frames sent. Connected frames will be both received as "K" frames and the appropriate monitoring frame.

Even if ***AGWPE*** handles AX.25 frames larger than 255 bytes not so many other programs could over the air, so it's a reasonably programming practice to ensure than the length of the data to be transferred is equal to or less than 255 bytes.

Activate reception of Frames in "raw" format ('k' Frame)

AGWPE send to the application all data using several frame formats (D or U for actual connected or unconnected data, I or S for monitored information, T for information the application sent, etc).

In particular, the I and S frames (as we would see) provides some "decoding" of the information as part of the data area of the frame; things such as the AX.25 header components, NETROM circuit control and TCP/IP connection control data are parsed and included in plain ASCII before the actual data.

The application program could, for light usages, process and decode the contents of either the 'I' (information) or 'S' (supervisory) frames.

For more serious usages it's likely the application would need the complete AX.25 frame and process it by it's own means.

This is accomplished with this frame, the application signals ***AGWPE*** that from this moment on all relevant information should be sent also in raw format; ***AGWPE*** will still continue to provide information with the regular frames (D/U/I/S) and it would also send the raw version of them using 'K' frames.

The format of the frame would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'k' (ASCII 0x6B)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	0
User (Reserved)	4 Bytes	0

AGWPE doesn't recognize this frame in any particular way, however, 'K' frames should start to flow into the application reflecting any activity at the ports.

This frame acts like a switch, the first time issued it enables the reception of raw frames while the second disables it and so on; in general on odd times it would enable and on even times it would disable.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 6B 00 00 00 00 00 00 00 00 00 00 00 |....k...........
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
|00 00 00 00 -- -- -- -- -- -- -- -- -- -- -- -- |....
```

Frames Sent by AGWPE to the Application

AGWPE might send information to the application basically for two main reasons:

- In response of a query from the application.
- To inform the application about some event (data arrived, monitoring frames, etc).

The frame format is exactly the same than the previously seen used by the application side, in fact, many frames shares the same datakind, so the meaning might differ depending on the direction of the information flow.

A cross reference among frames sent by the application and by AGWPE could be seen in a later section of this document (See ***Frame Cross-Reference*** on page 42)

Version Number ('R' frame)

This frame is sent by ***AGWPE*** to the application in response of an 'R' frame sent to ***AGWPE*** carrying the information about the current ***AGWPE*** version.

The format of the frame would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'R' (ASCII 0x52)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	8
User (Reserved)	4 Bytes	0

8 bytes of data would follow (as indicated by the DataLen field) containing the ***AGWPE*** version with the following contents:

Offset (Byte or Characters) into the Data Area	Meaning
+00	LSB of Major Version
+01	MSB of Major Version
+02	not used
+03	not used
+04	LSB of Minor Version
+05	MSB of Minor Version
+06	not used
+07	not used

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 52 00 00 00 00 00 00 00 00 00 00 00 |....R...........
|00 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00 |................
|00 00 00 00 D0 07 00 00 14 00 00 00 -- -- -- -- |............
```

Callsign Registration ('X' Frame)

This frame would be sent by ***AGWPE*** in response for a callsign registration ('X' frame) sent by the application.

The format of the frame would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'X' (ASCII 0x58)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Registered CallSign-SSID ended with null (0x00)
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	1
User (Reserved)	4 Bytes	0

1 byte of data would follow (as indicated by the DataLen field) containing the result of the registration:

Offset (Byte or Characters) into the Data Area	Meaning
+00	0x00 Registration Failed 0x01 Registration Successful

The application must refrain any further use of the callsign if the registration failed because it means that callsign is already in use (already registered) with ***AGWPE***.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 58 00 00 00 4C 55 37 44 49 44 2D 34 |....X...LU7DID-4
|00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 |................
|00 00 00 00 01 -- -- -- -- -- -- -- -- -- -- -- |.....
```

Port Information ('G' Frame)

This frame would be sent by ***AGWPE*** in response of a query for Port information ('G' frame) sent by the application.

The format of the frame would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	0x00
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'G' (ASCII 0x47)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	Length of port info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the port data using the following format:

- Total number of ports in ASCII (i.e. "1" ASCII 0x31) followed by ";".
- A data stream for each port using the general format "Portn xxxxxxx" followed by ";" with "n" being the port number (i.e. "Port1", "Port2"…) and "xxxxxxx" being the description of the port as seen in the Properties dialog of ***AGWPE***.

An example of a typical data area showing this information for two ports is

```
2;Port1 with KPC3 on COM1: 145.03 Mhz;Port2 with Loopback Port;
```

The first port information belongs to the Port1, the second to Port2 and so on. The application could rely on the ";" character to "parse" the successive components; ***AGWPE*** guarantees that the text information won't contain ';' characters other than the ones to separate succesive information pieces.

The total length of the stream would depend on the number of ports defined with ***AGWPE***

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 47 00 00 00 00 00 00 00 00 00 00 00 |....G...........
|00 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 00 |............\...
|00 00 00 00 32 3B 50 6F 72 74 31 20 77 69 74 68 |....2;Port1 with
|20 4B 50 43 33 20 4F 6E 20 43 4F 4D 31 3A 20 31 | KPC3 On COM1: 1
|34 35 2E 30 33 30 4D 68 7A 20 31 32 30 30 62 61 |45.030Mhz 1200ba
|75 64 3B 50 6F 72 74 32 20 77 69 74 68 20 4C 6F |ud;Port2 with Lo
|6F 70 42 61 63 6B 20 50 6F 72 74 3B 00 00 00 00 |opBack Port;....
|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
```

Capabilities of a Port ('g' Frame)

This frame is generated by ***AGWPE*** in response of a 'g' frame sent by the application and contain static configuration information as well as dynamically updated values for the particular port being queried.

The format of the frame would be:

Field	Length	Meaning

AGWPE Port	1 Bytes	Port being queried 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'g' (ASCII 0x67)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	12
User (Reserved)	4 Bytes	0

12 bytes of data would follow (as indicated by the DataLen field) containing the following information about the particular port referenced by the header's ***AGWPE***Port field :

Offset (Byte or Characters) into the Data Area	Meaning
+00	On air baud rate (0=1200/1=2400/2=4800 /3=9600…)
+01	Traffic level (if 0xFF the port is not in autoupdate mode)
+02	TX Delay
+03	TX Tail
+04	Persist
+05	SlotTime
+06	MaxFrame
+07	How Many connections are active on this port
+08 LSB Low Word +09 MSB Low Word +10 LSB High Word +11 MSB High Word	HowManyBytes (received in the last 2 minutes) as a 32 bits (4 bytes) integer. Updated every two minutes.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 67 00 00 00 4C 55 37 44 49 44 2D 34 |....g...LU7DID-4
|00 00 00 00 00 00 00 00 00 00 00 00 0C 00 00 00 |................
|00 00 00 00 00 01 19 04 C8 04 07 00 01 00 00 00 |................
```

Frames Outstanding on a Port ('y' Frame)

This frame is generated by ***AGWPE*** in response of a 'y' frame sent by the application and contains how many frames are waiting to be transmitted by ***AGWPE*** thru the indicated port (from all sources, not only from the application that makes the queries).

It could (should) be used by the application to introduce a "reality check" into the amount of data being sent to ***AGWPE*** for transmission in order to accommodate the real bandwidth of the port.

The format of the frame would be:

Field	Length	Meaning

AGWPE Port	1 Bytes	Port being queried 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'y' (ASCII 0x79)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	4
User (Reserved)	4 Bytes	0

4 bytes of data would follow (as indicated by the DataLen field) containing a 32 bits integer with the total number of frames waiting to be transmitted (outstanding frames) :

Offset (Byte or Characters) into the Data Area	Meaning
+00 LSB Low Word +01 MSB Low Word +02 LSB High Word +03 MSB High Word	Number of Frames waiting to be transmitted on the queried port.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 79 00 00 00 9C 09 58 00 A0 BE 9D 01 |....y.....X.....
|98 BE 9D 01 00 00 00 00 24 00 00 00 04 00 00 00 |........$.......
|98 02 BE 00 01 00 00 00 -- -- -- -- -- -- -- -- |........
```

Frames Outstanding on a Connection ('Y' Frame)

This frame is conceptually similar to the previous one but referring to a particular AX.25 L2 connection (CallFrom/CallTo pair) on a given port; it's returned by ***AGWPE*** when queried thru a 'Y' frame over an existing connection.

As in the 'y' frame this information could (should) be used by the application to control the pace of information delivery on a given connection in order to adjust it realistically to the port bandwidth.

The format of the frame would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port being queried 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'Y' (ASCII 0x59)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Callsign-SSID ended with null (0x00)
CallTo	10 Bytes	CallSign-SSID ended with null (0x00)

DataLen	4 Bytes	4
User (Reserved)	4 Bytes	0

4 bytes of data would follow (as indicated by the DataLen field) containing a 32 bits integer with the total number of frames waiting to be transmitted (outstanding frames) :

Offset (Byte or Characters) into the Data Area	Meaning
+00 LSB Low Word +01 MSB Low Word +02 LSB High Word +03 MSB High Word	Number of Frames waiting to be transmitted on the given AX.25 connection

Heard Stations on a Port ('H' Frame)

This frame is produced by ***AGWPE*** in response of an 'H' frame sent by the application and contains information about the stations heard by ***AGWPE*** on a given port.

Upon a single 'H' frame sent by the application ***AGWPE*** would produce 20 sucessive 'H' frames, one for each station heard.

If on a given port more than 20 stations were heard only the 20 most recently heard would be sent, if less than 20 stations were heard ***AGWPE*** would send as many "empty" 'H' frames as required to make the total number sent as 20.

The frame format would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Queried Port 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'H' (ASCII 0x48)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00
CallTo	10 Bytes	10 0x00
DataLen	4 Bytes	Length of heard info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the heard data using the following format:

- Callsign and SSID in ASCII ended with a blank.
- Timestamp of first hearing ended with a blank.
- Timestamp of last hearing ended with a blank.
- A null (0x00) at the end of the stream.

An example of a typical data area showing this information would be

```
LU7DID-4 Mon,21Feb2000 11:14:30 Mon,21Feb2000 12:18:22
```

After the null (0x00) signaling the end of the “plain ASCII” heard information follows two SYSTEMTIME structures containing the timestamp of the first hearing and the last hearing (see the Windows SDK help for information about this standard structure).

The application could “parse” the timestamps and get the individual components such as day, month, year, hour, minute and seconds of both the first hearing and last hearing information or to process the information using the SYSTEMTIME structures as it best suit the programmer’s preferences.

Take into consideration this information is stored by ***AGWPE*** from the moment it had been started the last time and it’s not preserved by ***AGWPE*** across succesive starts (meaning, the heard information would be empty just after ***AGWPE*** starts).

The empty entries (used to complete up to 20 entries) would have the callsign and timestamp information
not filled, so it would look like

```
                    00:00:00          00:00:00
```

Unexpected data might be expected by the application on an empty frame, it’s on the application responsibility to define when an entry contain valid information or it’s just empty and should be discarded.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|-- -- -- -- -- -- -- -- 01 00 00 00 48 00 00 00 |        ....H...
|4C 55 37 44 49 44 00 00 74 BD 9D 01 00 00 00 00 |LU7DID..t.......
|F0 BE 9D 01 5A 00 00 00 68 A0 F7 BF 20 20 20 4C |....Z...h...   L
|55 37 44 49 44 20 54 75 65 2C 32 32 46 65 62 32 |U7DID Tue,22Feb2
|30 30 30 20 31 30 3A 35 32 3A 31 32 20 20 54 75 |000 10:52:12  Tu
|65 2C 32 32 46 65 62 32 30 30 30 20 31 30 3A 35 |e,22Feb2000 10:5
|36 3A 30 38 00 27 11 D0 07 02 00 02 00 16 00 0A |6:08.'..........
|00 34 00 0C 00 32 00 D0 07 02 00 02 00 16 00 0A |.4...2..........
|00 38 00 08 00 3E -- -- -- -- -- -- -- -- -- -- |.8...>
```

AX.25 Connection Received (‘C’ Frame)

This frame is sent by ***AGWPE*** to the application when an AX.25 connection has been made, either started from the application from a registered callsign+SSID or initiated by a remote node with a registered callsign+SSID.

The frame format would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port where the connection had been made 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘C’ (ASCII 0x43)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00

CallFrom	10 Bytes	Callsign+SSID who the connection has been made to (usually the remote end) ended by null (0x00)
CallTo	10 Bytes	CallSign+SSID receiving the connection (usually one of our registered callsigns) ended by null (0x00)
DataLen	4 Bytes	Length of connect info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the connection message.

Depending on who started the connection the connection message could be:

- Connection started by our application, the message would be
  `*** CONNECTED With {Callsign-SSID}`
- Connection started by the other station, the message would be
  `*** CONNECTED To Station {Callsign-SSID}`

The application might “parse” the message to detect whether a given connection is the result of our connection request (thru a ‘C’ frame) or initiated independently the the other end; this verification has to always be made since we could not rule out the other station independently started a connection even simultaneously with our connection request (at the very least, it should be a good programming practice to perform that verification whenever a ‘C’ frame is received).

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 43 00 00 00 4C 55 37 44 49 44 2D 33 |....C...LU7DID-3
|00 2F 4C 55 37 44 49 44 2D 34 00 00 23 00 00 00 |./LU7DID-4..#...
|D3 73 F7 BF 2A 2A 2A 20 43 4F 4E 4E 45 43 54 45 |.s..*** CONNECTE
|44 20 54 6F 20 53 74 61 74 69 6F 6E 20 4C 55 37 |D To Station LU7
|44 49 44 2D 33 0D 00 -- -- -- -- -- -- -- -- -- |DID-3..
```

Connected AX.25 Data (‘D’ Frame)

This is a frame sent by ***AGWPE*** to the application when an information frame part of an established AX.25 connection directed to a registered station is detected.

The frame format would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port where the connection had been made 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	‘D’ (ASCII 0x44)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID
Reserved	1 Byte	0x00

CallFrom	10 Bytes	Callsign+SSID who sends the information(usually the remote end) ended by null (0x00)
CallTo	10 Bytes	CallSign+SSID receiving the information (usually one of our registered callsigns) ended by null (0x00)
DataLen	4 Bytes	Length of data info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the connected data in a fully transparent way (binary information, no delimiters, bit stuffing or escape codes), the data is as sent by the other end and could be inmediately used by the application without further processing.

Note that the PID reflected on the frame would be 0xF0 if the connection has been established by us using the 'C' command, could be any non-standard PID if the connection had been established by us using the 'c' command and could be anything if the connection has been established by the other end.

AGWPE guarantees the information is sent just once to the application in the right sequence (all the retries of information and resending of it due to link conditions is hidden from the application perspective), the frame doesn't provide any information about the frame sequence as received on the AX.25 link, in case the application needs that information pairing of the 'D' frame with other monitoring information should be made by the application by it's own means (albeit, this need should be extremely infrequent on normal uses).

Note there is no limit on the amount of data sent by ***AGWPE*** to the application with this frame since it's not necessarily related to a concrete AX.25 frame; so the application should not expect any given length to be used (i.e. several AX.25 frames could be bound together on a single 'D' frame sent by ***AGWPE***).

Since ***AGWPE*** supports the latest AX.25 specification no guarantee the frame is limited to 256 bytes do actually exist and the application should be able to process data of any arbitrary length.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 44 00 F0 00 4C 55 37 44 49 44 2D 33 |....D...LU7DID-3
|00 47 4C 55 37 44 49 44 2D 34 00 00 02 00 00 00 |.GLU7DID-4......
|A8 6D 45 00 62 0D -- -- -- -- -- -- -- -- -- -- |.mE.b.
```

Monitored Connected Information ('I' Frame)

This frame is sent by ***AGWPE*** to the application whenever any exchange of connected information is detected among any pair of stations on any port, for ***AGWPE*** to send this information the monitoring must be previously activated by the application thru the sending of a 'm' frame to ***AGWPE***.

The frame format would be

Field	Length	Meaning

AGWPE Port	1 Bytes	Port where the frame has been heard 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'I' (ASCII 0x49)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Callsign+SSID who sends the information(usually the remote end) ended by null (0x00)
CallTo	10 Bytes	CallSign+SSID receiving the information (usually one of our registered callsigns) ended by null (0x00)
DataLen	4 Bytes	Length of monitored info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the decoded headers of the AX.25 Frames and the connected data as sent by the transmitting end and could be inmediately used by the application without further processing.

AGWPE also includes a decode NETROM header or a decoded TCP/IP header when the respective frame types are detected.

The application might parse the information to extract both information relevant to the connected link and the actual data being interchanged.

The AX.25 headers included in the data area of the frame usually follows the format (example)

```
1:Fm LU7DID-4 To SV2AGW-2 <I R3 S1 pid=F0 Len=29 >[12:23:49]
```

Followed by the actual data being exchanged (binary information).

The application might choose to process the information and extract the relevant components, if so, the following things must be considered.

- The AX.25 header is sent by ***AGWPE*** in plain ASCII, no binary information, while the data itself is sent and should be handled as binary.
- The first number is the port where the information has been heard, please note it follows the convention used by ***AGWPE*** on it's Property Dialog and NOT the convention used on the frame headers (so "1' means "Port1", "2" means "Port2" and so on).
- The decoded header is presented by ***AGWPE*** in a consistent way, first the "From" (origin) station followed by the "To" (destination) station in callsign-SSID format.
- Then follows the AX.25 frame header data, a constant "I" meaning an information frame, followed by the Received and Sent AX.25 counters (N(R) and N(S) on the AX.25 protocol definition) as seen by the sending application.
- Follows the PID of the frame.
- Follows the length of the binary information on the frame.
- Then a timestamp of the frame reception at ***AGWPE***.

After the frame AX.25 header a CR (0x0D) follows and then the actual data in binary form.

Please note the application should handle TWO different lengths when handling this frame, the one stated on the header (DataLen) refers to the total amount of data transferred after the ***AGWPE*** header (which includes BOTH the decoded AX.25 header and the binary data).

A second length is the one stated on the decoded AX.25 header (Len=…) which referes to the actual amount of data transferred AFTER the header.

The application, in order to process this frame, should

- First get the whole data block as stated on the ***AGWPE*** header (DataLen).
- Then it should parse the data block till the first CR (0x0D) character and decode it on their components (Port/From/To/DataKind/N(R) /N(S)/Pid/Len).
- Then get as many bytes after the CR as stated in the “Len=” part of the AX.25 header decoded by ***AGWPE*** as plain text, those bytes (which are a binary block without an escape code, KISS masking or bit stuffing) are the actual data exchanged between both stations.
- Beware that ***AGWPE*** might include some few extra bytes of information after the binary block actually exchanged by the two connected stations being monitored which should be ignored by the application processing an ‘I’ frame. The application should read them in order to complete the processing of the ***AGWPE*** frame but later should ignore them.

The application should be aware on the fact that the data transmitted (or received) by it would be communicated by ***AGWPE*** using the appropriate frame AND also thru an ‘I’ frame, so if the ‘I’ frames sent by ***AGWPE*** are used with any functional purpose the redundancy has to be considered and solved by the application.

The conceptual thinking behind an ‘I’ frame is to provide the application with a way to provide it with “presentation” ready monitored information rather than to rely on them for any functional purpose (altrough, this could be done provided appropriate caution is taken as show above).

Take note that the correct PID of the monitored frame is correctly reflected on the “decoded” header provided by ***AGWPE*** rather than on the relevant field on the ***AGWPE*** header.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 49 00 00 00 4C 55 37 44 49 44 2D 34 |....I...LU7DID-4
|00 E7 4C 55 37 44 49 44 00 FA 6A 00 6F 00 00 00 |..LU7DID..j.o...
|2C 0E 45 00 20 32 3A 46 6D 20 4C 55 37 44 49 44 |,.E. 2:Fm LU7DID
|2D 34 20 54 6F 20 4C 55 37 44 49 44 20 3C 49 20 |-4 To LU7DID <I
|50 20 52 31 20 53 34 20 70 69 64 3D 46 30 20 4C |P R1 S4 pid=F0 L
|65 6E 3D 34 39 20 3E 5B 31 30 3A 35 35 3A 35 35 |en=49 >[10:55:55
|5D 0D 0D 5B 4C 55 37 44 49 44 40 4C 55 37 44 49 |]..[LU7DID@LU7DI
|44 2D 34 5D 20 42 2C 43 2C 44 2C 45 2C 58 2C 49 |D-4] B,C,D,E,X,I
|2C 4D 2C 3F 2C 4E 2C 50 2C 55 2C 4A 2C 52 3A 20 |,M,?,N,P,U,J,R:
|0D 0D 00 -- -- -- -- -- -- -- -- -- -- -- -- -- |...
```

Monitored Supervisory Information (‘S’ Frames)

Conceptually similar to the ‘I’ frames discussed before ***AGWPE*** sends the application information regarding supervisory frames interchanged among any two stations as a part of the AX.25 connected session as stated on the AX.25 protocol in order to administer a given link.

Those frames are usually

- SABM (Connection Request).
- UA (Connection Acknowledge/Accepted).
- DISC (Disconnection Request).
- DM (Disconnect Accepted).
- RR (Receiver Ready)
- RNR (Receiver not Ready)
- REJ (Frame Rejected)

Those frames doesn't transport information other than their own meaning in the context of a given connection.

The frame format would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port where the frame has been heard 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'S' (ASCII 0x53)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Callsign+SSID who sends the information(usually the remote end) ended by null (0x00)
CallTo	10 Bytes	CallSign+SSID receiving the information (usually one of our registered callsigns) ended by null (0x00)
DataLen	4 Bytes	Length of supervisory info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the decoded headers of the AX.25 Supervisory Frames sent by the transmitting end and could be inmediately used by the application without further processing.

The usual format of the information is as follows

```
1:Fm LU7DID-2 To SV2AGW-11 <RR P/F R2 >[12:11:19]
```

In the above example a "RR" frame is shown; all information provided by ***AGWPE*** is formatted as plain ASCII text (no binary elements) and is "presentation ready" for the application; the application could choose to "parse" that information and use it functionally if needed.

Most of the programming recommendations provided for the 'I' frames still does apply with the exception of no data to be associated with the frame after the header.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 53 00 00 00 4C 55 37 44 49 44 2D 33 |....S...LU7DID-3
|00 E7 4C 55 37 44 49 44 2D 34 00 00 2F 00 00 00 |..LU7DID-4../...
|2C 0E 45 00 20 32 3A 46 6D 20 4C 55 37 44 49 44 |,.E. 2:Fm LU7DID
|2D 33 20 54 6F 20 4C 55 37 44 49 44 2D 34 20 3C |-3 To LU7DID-4 <
|53 41 42 4D 20 50 3E 5B 31 30 3A 35 37 3A 35 32 |SABM P>[10:57:52
|5D 0D 00 -- -- -- -- -- -- -- -- -- -- -- -- -- |]..
```

Monitored Unproto Information ('U' Frames)

Conceptually similar to the 'I' frames discussed before ***AGWPE*** sends the application information regarding unnumbered (unproto) frames frames interchanged among any two stations as a part of the AX.25 connected session as stated on the AX.25 protocol in order to administer a given link.

Those frames are usually related to beacons or broadcasted data of some sort or in more advanced uses convey NETROM or TCP/IP links related information.

The frame format would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port where the frame has been heard 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'U' (ASCII 0x55)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Callsign+SSID who sends the information(usually the remote end) ended by null (0x00)
CallTo	10 Bytes	CallSign+SSID receiving the information (usually one of our registered callsigns) ended by null (0x00)
DataLen	4 Bytes	Length of unproto info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the decoded headers of the AX.25 Unproto Frames sent by the transmitting end and could be inmediately used by the application without further processing.

The usual format of the information is as follows

```
1:Fm LU7DID-2 To SV2AGW-11 <UI pid=F0 Len=57 >[12:11:19]
```

The application might choose to process the information and extract the relevant components, if so, the following things must be considered.

- The AX.25 header is sent by ***AGWPE*** in plain ASCII, no binary information, while the data itself is sent and should be handled as binary.
- The first number is the port where the information has been heard, please note it follows the convention used by ***AGWPE*** on it's Property Dialog and NOT the convention used on the frame headers (so "1' means "Port1", "2" means "Port2" and so on).
- The decoded header is presented by ***AGWPE*** in a consistent way, first the "From" (origin) station followed by the "To" (destination) station in callsign-SSID format.
- Then follows the AX.25 frame header data, a constant "UI" meaning an unproto frame, as seen by the sending application.
- Follows the PID of the frame.
- Follows the length of the binary information on the frame.
- Then a timestamp of the frame reception at ***AGWPE***.

After the frame AX.25 header a CR (0x0D) follows and then the actual data in binary form.

Please note the application should handle TWO different lengths (as if the 'I' frame) when handling this frame, the one stated on the header (DataLen) refers to the total amount of data transferred after the ***AGWPE*** header (which includes BOTH the decoded AX.25 header and the binary data).

A second length is the one stated on the decoded AX.25 header (Len=…) which referes to the actual amount of data transferred AFTER the header.

The application, in order to process this frame, should

- First get the whole data block as stated on the ***AGWPE*** header (DataLen).
- Then it should parse the data block till the first CR (0x0D) character and decode it on their components (Port/From/To/DataKind/Pid/Len).
- Then get as many bytes after the CR as stated in the "Len=" part of the AX.25 header decoded by ***AGWPE*** as plain text, those bytes (which are a binary block without an escape code, KISS masking or bit stuffing) are the actual data exchanged between both stations.
- Beware that ***AGWPE*** might include some few extra bytes of information after the binary block actually exchanged by the two connected stations being monitored which should be ignored by the application processing an 'I' frame. The application should read them in order to complete the processing of the ***AGWPE*** frame but later should ignore them.

The application should be aware on the fact that the data transmitted (or received) by it would be communicated by ***AGWPE*** using the appropriate frame AND also thru an 'U' frame, so if the 'U' frames sent by ***AGWPE*** are used with any functional purpose the redundancy has to be considered and solved by the application.

The conceptual thinking behind an 'U' frame is to provide the application with a way to provide it with "presentation" ready monitored information rather than to rely on them for any functional purpose (altrough, this could be done provided appropriate caution is taken as show above).

Take note that the correct PID of the monitored frame is correctly reflected on the "decoded" header provided by ***AGWPE*** rather than on the relevant field on the ***AGWPE*** header.

If the unproto frame transport additional information on known formats used by other protocols such as NETROM or TCPIP ***AGWPE*** would attempt to "decode' them also and provide a "plain text" version of them.

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 55 00 00 00 4C 55 37 44 49 44 2D 34 |....U...LU7DID-4
|00 E7 4E 45 54 4D 45 00 C8 FA 6A 00 6F 00 00 00 |..NETME...j.o...
|2C 0E 45 00 20 32 3A 46 6D 20 4C 55 37 44 49 44 |,.E. 2:Fm LU7DID
|2D 34 20 54 6F 20 4E 45 54 4D 45 20 3C 55 49 20 |-4 To NETME <UI
|70 69 64 3D 46 30 20 4C 65 6E 3D 35 37 20 3E 5B |pid=F0 Len=57 >[
|31 30 3A 35 37 3A 34 32 5D 0D 0D 42 65 61 63 6F |10:57:42]..Beaco
|6E 20 64 65 20 4E 6F 64 6F 20 4C 55 37 44 49 44 |n de Nodo LU7DID
|2D 34 20 41 64 72 6F 67 75 65 20 42 41 20 41 72 |-4 Adrogue BA Ar
|67 65 6E 74 69 6E 61 20 5B 47 46 30 35 54 45 5D |gentina [GF05TE]
|0D 0D 00 -- -- -- -- -- -- -- -- -- -- -- -- -- |...
```

Monitoring Own Information ('T' Frames)

All information sent unproto by the application thru the 'M' frame is returned by ***AGWPE*** once transmitted as a 'T' frame, this frame could be used for confirmation purposes.

The format would be:

Field	Length	Meaning
AGWPE Port	1 Bytes	Port where the connection had been made 0x00 Port1 0x01 Port2 ….
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'T' (ASCII 0x54)
Reserved	1 Byte	0x00
PID	1 Byte	AX.25 PID
Reserved	1 Byte	0x00
CallFrom	10 Bytes	Callsign+SSID who sends the information(usually the remote end) ended by null (0x00)
CallTo	10 Bytes	CallSign+SSID receiving the information (usually one of our registered callsigns) ended by null (0x00)
DataLen	4 Bytes	Length of data info
User (Reserved)	4 Bytes	0

A stream of bytes would follow (as indicated by the DataLen field) containing the sent data in a fully transparent way (binary information, no delimiters, bit stuffing or escape codes), the data is as sent by the application and could be (eventually) used inmediately by the application without further processing. Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|00 00 00 00 54 00 00 00 4C 55 37 44 49 44 00 00 |....T...LU7DID..
|38 E7 4C 57 35 44 47 4D 00 FA 6A 00 44 00 00 00 |8.LW5DGM..j.D...
|2C 0E 45 00 20 31 3A 46 6D 20 4C 55 37 44 49 44 |,.E. 1:Fm LU7DID
|20 54 6F 20 4C 57 35 44 47 4D 20 3C 55 49 20 70 | To LW5DGM <UI p
|69 64 3D 46 30 20 4C 65 6E 3D 31 32 20 3E 5B 31 |id=F0 Len=12 >[1
|30 3A 35 39 3A 31 30 5D 0D 3F 20 30 30 30 38 36 |0:59:10].? 00086
|31 34 31 61 61 0D 0D 00 -- -- -- -- -- -- -- -- |141aa...
```

Monitored Information in Raw Format ('K' Frames)

When enabled to do so thru the 'r' Frame ***AGWPE*** would send to the application a "raw" version of every monitored frame (on top of the respective I/S/U frames if enabled to).

This frame is basically a representation of the AX.25 frame actually received on the port.

The ***AGWPE*** frame format would be

Field	Length	Meaning
AGWPE Port	1 Bytes	Port where the data frame had been received thru {0=Port1,1=Port2,…}
Reserved	3 Bytes	0x00 0x00 0x00
DataKind	1 Byte	'K' (ASCII 0x4B)
Reserved	1 Byte	0x00
PID	1 Byte	0x00
Reserved	1 Byte	0x00
CallFrom	10 Bytes	10 0x00 Actual origin is stated in the raw frame
CallTo	10 Bytes	10 0x00 Actual destination is stated in the raw frame
DataLen	4 Bytes	Number of Data Bytes to be transferred.
User (Reserved)	4 Bytes	0

The complete frame (AX.25 header followed by data if applicable) in raw format follows as data, the exact length of it must be reflected on the DataLen field.

Full knowledge of the intricancies of the AX.25 must be mastered by the brave programmer trying to use this frame, for what is worth the heartfull recommendation is to try to identify other frames or combination of frames most suitable for a given purpose, so use this frame format as an absolute last resort.

Probably, for most light uses it would suffice to decode the equivalent information on the I/S/U frames.

For those brave souls in need to still use it the following recommendations should be used (refer to the AX.25 Protocol documentation for the naming conventions).

Field	Length	Description
Flag	1 byte	Will not be the standard 0b01111110 flag but the "TNC" to use 00=Port 1 16=Port 2 …
Address	112/360 bits	AX.25 coded Origin, Destination and (optionally) digipeaters.
Control	1 byte	AX.25 Control Field
PID	1 byte	AX.25 PID
Info	N bytes	AX.25 Information Area

Please note than the AX.25 FCS and the ending Flag are NOT included. No KISS escape codes nor bit

stuffing is required to be performed (***AGWPE*** would add them as needed).

Follows a sample frame using a dump format of this frame (16 hexadecimal formatted bytes at the left and the ASCII, interpretation when feasible at the right), this sample could be used for study and comparation purposes.

```
|01 00 00 00 4B 00 00 00 4C 55 37 44 49 44 00 00 |....K...LU7DID..
|38 E7 4C 55 37 44 49 44 2D 34 00 00 42 00 00 00 |8.LU7DID-4..B...
|2C 0E 45 00 00 98 AA 6E 88 92 88 80 98 AA 6E 88 |,.E....n......n.
|92 88 69 38 F0 0D 0A 5B 4C 55 37 44 49 44 40 4C |..i8...[LU7DID@L
|55 37 44 49 44 2D 34 5D 20 42 2C 43 2C 44 2C 45 |U7DID-4] B,C,D,E
|2C 58 2C 49 2C 4D 2C 3F 2C 4E 2C 50 2C 55 2C 4A |,X,I,M,?,N,P,U,J
|2C 52 3A 20 0D 0A -- -- -- -- -- -- -- -- -- -- |,R: ..
```

Frame Cross-Reference

Follows a cross-reference of all frame types supported by ***AGWPE*** and it's meaning whether they are sent by either the application to ***AGWPE*** or ***AGWPE*** to the Application (N/A means frame not supported on this particular end or no answer frame produced).

DataKind	Description	Application	AGWPE
'P'	***Application Login (User/Password)***	Login into AGWPE	N/A
'R'	***AGWPE*** Version	Query ***AGWPE*** Version	Answer ***AGWPE*** Version
'G'	Port Information	Query ***AGWPE*** Ports	Answer ***AGWPE*** Port Information
'g'	Port Capabilities	Query ***AGWPE*** Port Capability	Answer queried Port capabilities
'X'	Register CallSign	Register CallSign	Success/Failure of Registration
'x'	Unregister CallSign	Unregister CallSign	N/A
'y'	Outstanding frames on a Port	Query outstanding frames waiting to be transmitted on a port (all sources)	Answer outstanding frames waiting to be transmitted on a port (all sources)
'Y'	Outstanding frames on a connection	Query outstanding frames waiting to be transmitted on an AX.25 connection (From/To pair)	Answer outstanding frames waiting to be transmitted on a existing AX.25 connection (From/To pair).
'H'	Heard Stations on a Port	Query heard stations on a port	Answer heard stations on the requested port (20 frames would be generated)
'm'	Start Monitoring Data	Start/Stop the flow of monitoring data (first time switch on, second time switch off, and so on).	N/A
'M'	Send Unproto Info	Send Unproto Info	N/A
'V'	Send Unproto Info VIA	Send Unproto Info VIA	N/A
'C'	Start an AX.25 Connection	Start an AX.25 connection	Success or Failure of AX.25 Connection Request
'v'	Start an AX.25 Connection using digipeaters	Start an AX.25 connection using digipeaters	N/A answered as in 'C'
'c'	Start a non-standard AX.25 connection (data frames would have PID not 0xF0)	Start a non standard AX.25 Connection	N/A answered as in 'C'

‘D’	Connected Data	Send Connected Data	Receives Connected Data
‘d’	Disconnect an AX.25 connection	Start disconnection of AX.25 connection	Success or Failure of AX.25 disconnection
‘U’	Unnumbered Information (UI) frame received for a registered application.	N/A	Unnumbered Information (UI) frame received for a registered application.
‘I’	Information Frame between any two connected station (not necessarily the ones registered by the application)	N/A	Information Frame monitored
‘S’	Supervisory Frame (SABM/UA/DISC/DM/RR/REJ)	N/A	Supervisory Frame monitored
‘T’	Monitored frame sent by this application	N/A	Monitored Frame sent by this application.
‘K’	Raw Frame	Raw Frame To be sent	Raw Frame Received
‘k’	Start monitoring using raw frames	Start/Stop sending information in raw format. (First time switch on, Second switch off, and so on).	N/A

Programming Hints, Tips and Techniques

AGWPE could be truly despicted as a “middleware” application, a very powerful one, or in plain language a component that other applications uses as an enabler to perform “things”, in this case the “thing” is to exchange information thru AX.25 Packet Radio.

In one hand, ***AGWPE*** is extremely easy to use and powerful compared with the alternative which is to deal directly to the intrincancies of a myriad of different hardware components not to mention the arcane AX.25 protocol itself.

In the other hand, ***AGWPE*** could have a rather step learning curve for the novel programmer; after all it’s a rather complex piece of software doing a rather complex activity, and as any such software it has it’s idiosincracyes and “dark spots” that the application programmer should plan for.

No middleware application (no matter how brilliantly conceived and implemented) is more successful than the applications written or adapted to it, applications is what the final user “uses” and likes (or dislikes).

Applications could be written by seasoned programmers, fluent on the world of TCP/IP sockets, used to deal thru different APIs and aware of the fact that no perfect platform (perfectly coherent and bug free) has ever been written; this audience would flip thru this manual and probably felt at home right away, with perhaps more attention to the colloquial sidenotes than to the overall technical details.

But applications could also be written by novel programmers with great ideas for whom all those factors could be truly disabling.

The following sections aims for the second audience, some general references and known tricks/workarounds would be documented to aid a novel programmer on their first steps; it could be also useful for the seasoned ones when some part of the reference didn’t fulfill it’s goal to try to clarify a given funcion (so a programming example could spoke aloud by itself).

Programming Language

The issue of the programming language is easy to fix, Applications programs could be written on almost anything under the sun as long as

- It could handle standard TCP/IP sockets.
- ….. Could not think on more pre-requisites….

AGWPE itself has been written in MS C++, many examples do exist in the form of short pieces of code or sample programs provided by the author using that language; this is of course a matter of convenience and not a pre-requisite by any stretch of the concept that Applications intended to be run with ***AGWPE*** has to be written in C++.

Some rather major pieces of code to work with ***AGWPE*** has been written using Delphi 4/5 (Object Pascal) and surely Visual Basic programmers could extensively benefit from good “helpers” to implement TCP/IP oriented applications available for free or at reasonable costs.

Those are the languages more common in the Windows world, but any other language satisfying the

requirements could be used as well.

Nothing prevents that a given application could be implemented on MORE than one language, in fact, there are some experiences writing a part of an application in (say) Delphi4 and other part of it (as a DLL) in (say) C++; given the proper “bound” conditions are met nothing prevents such projects to take place.

While a multilanguage implementation could only make sense on the context of a program really big, most likely programs will not be that complex to make that a reasonable decision; however, helpers and other supporting libraries could be developed in the form of shareable components (such as ActiveX controls or Windows DLLs) that could be shared among almost anything platform making the usage of ***AGWPE*** easier for others to use.

When outlining the prerequisites the need of the Application to run on Windows was made as a relative statement; ***AGWPE* must run** on a Windows environment but the Application doesn’t needs to run on the same machine as long as it could have TCP/IP connectivity with the machine where ***AGWPE*** is running.

No actual part of the ***AGWPE*** API depends on Windows itself (with the exception of the SYSTEMTIME structure present on the ‘H’ Frames, and that is quite solvable as a problem) and thus an application for ***AGWPE*** could truly be written on any platform as long as it supports some form of standard TCP/IP, quite extraneous things come to my mind after this statement.

Nothing prevents an ***AGWPE*** application to be run under a 8086 DOS, on a Cray machine or a big IBM mainframe under OS/390 ; hopefully those possibilities could be seen as extreme and certainly not recommended.

However, a much more down to the ground possibility is for applications either written, adapted or ported to be run under the very popular Linux environment.

The only drawback of the “multiplatform” approach is the need of a minimum of two machines, one to run ***AGWPE*** (which must run under Windows) and other to run the application; this might or might not be very exotic as a proposition given the falling prices for hardware.

All the remaining of this document would assume the Application program is implemented under Windows though.

Talking with AGWPE

The first requirement to use ***AGWPE*** from an application is to be able to talk with it, and for that the application must be able to open, sustain and close a TCP/IP socket.

Countless libraries do exist either free or at reasonable cost to allow almost any language to accomplish that, most modern versions of the most widely used languages comes from the factory with support for TCP/IP right inside.

Unfortunately, even if the description at a conceptual level of programming with TCP/IP are common across any language used, the details might differ from application to application; so it’s sort of difficult to provide a single and definitive recommendation on how to program that part.

Each programmer should take the general guidelines provided here or in other places pointed, check with the information provided by the particular libraries and implementation of the compiler at hand and make the necessary adjustments.

Using C++

George (***SV2AGW***) provides a simple sample monitoring program written in C++ for the ***AGWPE*** on his Web site at http://www.forthnet.gr/***SV2AGW***

While limited in functionality a monitor frame shows a great deal of the most commonly used aspects of the ***AGWPE*** API, but more important, it has ALL the TCP/IP components needed on any program of any complexity.

A monitor program should open a socket, send some information to ***AGWPE*** (i.e. the 'm' frame to start monitoring), receive information thru the socket and eventually close it.

For a detailed information the download of this sample is recommended, it also contains many of the tips outlined in this document.

Using Delphi4/5

Delphi comes with Visual Component Library (VCL) components to support a TCP/IP socket client.

The application needs to sets the minimum properties (IP Address/TCP Port), make the component active and be sure handlers are written for the events OnConnect, OnRead,OnError and OnClose.

The connection with ***AGWPE*** is quite similar (despite the different information exchanged and the port number used) to a Telnet application, so check one of the many examples of a Telnet application on how to handle a connection like this.

Overall Communication Cycle

The communication between an application and ***AGWPE*** could take many different "logics" or "implementations" but the recommended one should implement the following topics.

- Establish a TCP/IP socket to the known IP Address where ***AGWPE*** is at TCP port 8000.
  - Handle both the successful connection of the socket and a unsuccessful one.
  - Don't code your program in a way that if it successfully connect with ***AGWPE*** everything is fine but if the connection fails everything goes down the tubes on an uncontrolled manner.
  - The user should be able to change the IP address of ***AGWPE*** at any time in order to establish a successful connection. The program should also plan for the user to change the IP address while AGWPE is connected and this lead to a disconnection/re-connection cycle automatically performed.
  - On extremely rare ocassions ***AGWPE*** might be so busy that it might delay a little on establsh a connection, the application should retry a controlled number of times before to give up and assume a connection could not be made.
- Once the socket has been established the following information should be exchanged with ***AGWPE***.
  - Send the login with an authorized User/Password, the application should be flexible to allow the user to define whether or not the delivery of a login frame is needed and if needed which is the User/Password to be used.

    - Request and Get the ***AGWPE*** Version information, verify if the release is proper for the functions used by the program; refuse to operate the application if the version could not be obtained or it's some very old one.
    - Request and Get information about the ***AGWPE*** current port configuration ('G').
    - Optionally, retrieve the capabilities of each port informed ('g').
    - Optionally, enable the sending of monitoring frames ('m').
    - Optionally, enable the sending of raw frames ('r').
    - Optionally, register one or more callsigns+SSID as needed ('X'), check the success of the operation before any actual use of it. Do not over-register, register callsigns as needed during the program execution.
    - Optionally, Get an initial state of the ***AGWPE*** ports ('y') and the heard stations ('H').
- Once the initial information has been exchanged the application is ready to fully operate with ***AGWPE***.
    - Start connections ('C','c' or 'v' as needed) and handle the answer of ***AGWPE***.
    - Send connected data ('D') and receive unsolicited data from other station ('D'), use the 'Y' frame to avoid to send information on a volume not reasonable to the physical capability of the port (in terms of bandwidth) to handle.
    - Send unconnected data ('M','V') such as beacons or other broadcasted data.
    - Stop connections ('d').
    - Optionally, handle monitored information ('I','S','U' or 'K').
    - Optionally, refresh periodically the status information on ***AGWPE***, the recommended method would be:
        - Every reasonable amount of time (i.e. using a 1 minute timer) send a 'G' request to ***AGWPE***.
        - Even if the 'G' information won't change between successive calls it would return the Port information.
        - For every Port informed send to ***AGWPE*** a 'g', a 'H' and a 'y' frames requesting information about the Port status, the stations heard and the queued frames on it. Process the answer from ***AGWPE*** and store it in your own program.
    - At any momment during the dialog the socket connection could terminate with an error, your application should be prepared for that. This might happen for some error on the transport (impossible if running on the same machine, unlikely if running over an Ethernet connection and more likely if running over a dialup connection) or because for some reason ***AGWPE*** were stopped (or decided to make a…. uhmmmm…. unsolicited stop).
        - In such cases the application should try to reconnect (re-establish the socket connection) a reasonable number of times or until stopped.
        - Beware that ***AGWPE*** doesn't transport the knowledge of the particular connection across connections, so when reconnected all the initial configuration steps must be repeated.
- When the application program needs to terminate (either because the user selected that, the functional purpose had been acomplished or the program is handling an abnormal termination) certain steps should be followed.
    - All connected links should be terminated sending as many 'd' frames as needed to terminate them, it's not absolutely necessary but recommended that the program waits for the finalization of the disconnection (a dying program might not afford that luxury).
    - Unregister all registered callsigns sending a 'x' frame for each one.
    - Close the TCP/IP socket connection with ***AGWPE***.

Frames Fiesta

The absolute key point to manage the communication with ***AGWPE*** is to be able to send to it and receive from it properly formatted frames to accomplish a given activity.

Sending Frames

This is a relatively easy activity to do if properly organized.

The ***AGWPE*** frames are composed by a fixed formatted header with some data added to it depending on the particular frame being sent.

The development of a particular routine that enable the application to send frames from whatever part of the program logic when it's needed is absolutely recommended (in case Object Oriented Programming is used the equivalent action is to define ***AGWPE*** as an object and to provide a method to send frames to it).

The function/method should be something like the following prototype (in Pascal)

```
Function Send(cPort : Char;
              cDataKind : Char;
              cPID : Char;
              sFrom : String ;
              sTo : String;
              iLen : Integer;
              sData : String) : Boolean;
```

This function/method should

- Validate the parameters are valid (i.e. known DataKind, existing port, known PID, valid From/To for the particular DataKind, etc).
- Format the frame (i.e. transform variable length strings into fixed length & null terminated strings, etc).
- Verify a proper TCP/IP connection with ***AGWPE*** had been already established.
- Send the frame to ***AGWPE***.
- Return a value reflecting the success failure of the action.

A complete and flexible send function/method would probably be the best single investment a programmer could do at the beginning of the activity with ***AGWPE***.

Once this function is available to send frames to ***AGWPE*** would be an surprinsingly trivial thing.

i.e. to register a callsign

Send(NUL,'X',NUL,'***LU7DID***-2','',0,'');

Would do it!!!

George (***SV2AGW***) proposes the following C++ procedure for a simple routine

```
void  SendPacket(char *ToCall,char *str,int count,int DataKind,int port)
{
if (count==0) count=lstrlen(str);
char szTemp[3000];
MoveMemory(szTemp.&port,sizeof(int));//which port to tx
MoveMemory(szTemp+sizeof(int),&DataKind,sizeof(int));//datakind here LOWORD should be
'D'
//datakind here should be 'D'
       MoveMemory(szTemp+(sizeof(int)*2),MyCall,strlen(MyCall)+1);//mycall
       MoveMemory(szTemp+(sizeof(int)*2)+10,ToCall,strlen(ToCall)+1);//other station
call
       MoveMemory(szTemp+(sizeof(int)*2)+10+10,&count,4);//length of the data we send
```

```
to other station
       MoveMemory(szTemp+(sizeof(int)*4)+10+10,str,count+1);//now add the actual data
after leving 4 additional bytes for USER which are reserved for the moment;
TXDATA(szTemp,count+26);//send them over our socket connection
}
```

The following fragment shows how to use this routine to send an unproto frame to all ports

```
for (int x=0;x<HowManyPorts;x++)
{
SendPacket("BEACON",test,strlen(test),MKELONNG('M',0),MAKELONG(x,0));
}
```

A comprehensive information about this and other routines could be found on George's authored demo monitoring program.

The following couple of methods written in Delphi4 are used on most (if not all) programs written by Pedro (***LU7DID***).

The first method (Write) is just a wrapper that isolates the calling routine about the details of the TCP/IP connection, basically it verify the link is ready and then pass all the information to a second (send) who actually delivers the frame.

This routines are used on multithreaded environments so the resources are protected from re-entrancy issues (this should not be a concern for the writer or a simple, single threaded program), some proprietary functions for selective tracing and debug are also present that should be ignored.

```
(*--->>> Write         <<<-----------------------------------------------*)
{*Write information to the packet engine                                 *}
{*-------------------------------------------------------------------------*}
Function TLink.Write(cPort : Char; cPID : Char; cDataKind : Char; szFrom : String; szTo
: String; iDataLen : DWORD; szData : String) : Boolean;

begin { TLink.Write}

                                              {*-----------------------*}
                                              {*Only write to valid    *}
                                              {*sockets                *}
                                              {*                       *}
{*-------------------------------------------------------------------------*}

     If AGWState <> 2 then begin
        PutAGW(1,'AGWPE Not connected, ignoring Write request');
        Result := FALSE;
        Exit;
     end;

    Send(AGWSocket,cPort,cPID,cDataKind,szFrom,szTo,iDataLen,szData);
    Result := TRUE;

end; { TLink.Write}

(*--->>> Send          <<<-----------------------------------------------*)
{* Function to send a frame to AGWPE (Low Level Routine)                 *}
{*-------------------------------------------------------------------------*}
Function TLink.Send     (Socket    : TCustomWinSocket ;
                         cPort     : Char;
                         cPID      : Char;
                         cDataKind : Char;
```

```
                        szFrom    : String;
                        szTo      : String;
                        iLen      : DWORD;
                        szBuffer  : String)    :  Boolean;
  Var
    Index               :       Integer;
    MSB                 :       Byte;
    LSB                 :       Byte;
    szFrame             :       String;
    szFromAux           :       String;
    szToAux             :       String;
    iBigLen             :       DWORD;
    szBigBuffer         :       String;
    dwStatus            :       DWORD;

begin { Send }

                                           {*-----------------------*}
                                           {*Re-entrancy protection *}
{*-------------------------------------------------------------------*}

  dwStatus := WaitForSingleObject(hAGWSend,INFINITE);
  If dwSTATUS <> WAIT_OBJECT_0 then begin
     PutAGW(1,'dwStatus <> WAIT_OBJECT_0 returned by WaitForSingleObject');
  end; {endif}


  If ((cDataKind = 'g') or (cDataKind = 'H') or (cDataKind = 'G')) then begin
     PutAGW(1,'SEND to AGWPE:Port {'+inttostr(ord(cPort))+'} DataKind ('+cDataKind+')
Pid=('+inttostr(ord(cPid))+') From <'+szFrom+'> To <'+szTo+'> Len
('+inttostr(iLen)+')');
  end else begin
     PutAGW(1,'SEND to AGWPE:Port {'+inttostr(ord(cPort))+'} DataKind ('+cDataKind+')
Pid=('+inttostr(ord(cPid))+') From <'+szFrom+'> To <'+szTo+'> Len
('+inttostr(iLen)+')');
  end; {endif}



  DumpHex(4,szBuffer);

                                           {*-----------------------*}
                                           {*does we have a live    *}
                                           {*connection already?    *}
                                           {*If NOT -> Error        *}
{*-------------------------------------------------------------------*}
  If Socket = Nil then begin
     PutAGW(1,'SEND: Socket = Nil, frame discarded');
     ReleaseSemaphore(hAGWSend,+1,Nil);
     Result  := FALSE;
     Exit;
  End;

                                           {*-----------------------*}
                                           {*Init buffer and temp   *}
                                           {*areas                  *}
                                           {*                       *}
{*-------------------------------------------------------------------*}
  szFrame     := '';
  szFromAux   := szFrom;
  szToAux     := szTo;

                                           {*-----------------------*}
                                           {*Ensure the whole header*}
                                           {*is filled with nulls   *}
                                           {*as well as the callsign*}
{*-------------------------------------------------------------------*}
```

```
  szFrame     := PadStr(szFrame,AGW_HEADER,NUL);
  szFromAux   := PadStr(szFromAux,10,NUL);
  szToAux     := PadStr(szToAux,10,NUL);

                                          {*-----------------------*}
                                          {*Format the frame       *}
                                          {*                       *}
                                          {*                       *}
{*-------------------------------------------------------------------*}

  szFrame[01] := Chr(ord(cPort)-1);       {* Port                  *}
  If szFrame[01] = Chr($FF) then begin
     szFrame[01] := Chr($00);
  end; {endif}

  szFrame[02] := NUL;
  szFrame[03] := NUL;
  szFrame[04] := NUL;

  szFrame[05] := cDataKind;               {* LOWord(DataKind)      *}
  szFrame[06] := NUL;
  szFrame[07] := cPID;                    {* HiWord(bPID)          *}
  szFrame[08] := NUL;

  For Index := 1  to 10  do begin         {* From Call             *}
    szFrame[08+Index] := szFromAux[Index];
  end; {endfor}

  For Index := 1  to 10  do begin         {* To Call               *}
    szFrame[18+Index] := szToAux[Index];
  end; {endfor}

  If iLen <= (MAXFRAME-1) then begin
     MSB := Trunc(iLen/MAXFRAME);
     LSB := Trunc(iLen - (MSB*MAXFRAME));
  end else begin
     MSB := $00;
     LSB := $FF;
  end; {endif}

  szFrame[29] := chr(LSB);                {* Size                  *}
  szFrame[30] := chr(MSB);
  szFrame[31] := NUL;
  szFrame[32] := NUL;

  szFrame[33] := NUL;                     {* User - Reserved       *}
  szFrame[34] := NUL;
  szFrame[35] := NUL;
  szFrame[36] := NUL;

  PutAGW(1,'Frame to send to AGWPE is');
  DumpHex(1,szFrame+szBuffer);


                                          {*-----------------------*}
                                          {*Efficiency trick       *}
                                          {*If there is data send  *}
                                          {*with the frame,othewise*}
                                          {*send just the frame    *}
                                          {*This will reduce the   *}
                                          {*chances for TCP to frag*}
{*-------------------------------------------------------------------*}
  If (iLen = 0) then begin
     PutAGW(3,'Send Header');
     DumpHex(3,szFrame);
     SendAGW(szFrame);
```

```
  end else begin
     If (iLen <= (MAXFRAME-1)) then begin
        PutAGW(3,'Send Header+Data');
        DumpHex(3,szFrame+szBuffer);
        SendAGW(szFrame+szBuffer);
     end else begin

                                                {*-----------------------*}
                                                {*Handles data areas     *}
                                                {*longer than 256 bytes  *}
                                                {*in sucessive frames    *}
                                                {*of up to 256 bytes     *}
{*--------------------------------------------------------------------------*}
        SendAGW(szFrame+Copy(szBuffer,1,(MAXFRAME-1)));

        szBigBuffer := Copy(szBuffer,MAXFRAME,Length(szBuffer)-(MAXFRAME-1));
        iBigLen     := Length(szBigBuffer);

        ReleaseSemaphore(hAGWSend,+1,Nil);

        Self.Send(Socket,cPort,cPid,cDataKind,szFrom,szTo,iBigLen,szBigBuffer);
        If cDataKind <> 'Y' then begin
           Self.Send(Socket,cPort,NUL,'Y',szFrom,szTo,0,'');
        end; {endif}

        Result := TRUE;
        Exit;
     end; {endif}
  end; {endif}

  ReleaseSemaphore(hAGWSend,+1,Nil);


                                                {*-----------------------*}
                                                {*Piggyback a request    *}
                                                {*for AGW Status on every*}
                                                {*Data Frame Sent        *}
                                                {*                       *}
{*--------------------------------------------------------------------------*}

  If cDataKind = 'D' then begin
     Self.Send(Socket,cPort,NUL,'Y',szFrom,szTo,0,'');
  end; {endif}
  Result := TRUE;

end; { Send }

(*--->>> SendAGW       <<<-------------------------------------------*)
{* Function to send at low level (TCPIP) the actual frame             *}
{*--------------------------------------------------------------------*}
Procedure TLink.SendAGW( szBuffer : String);

begin { TLink }
    PutAGW(1,'Send to AGWPE (TCP/IP)');
    DumpHex(1,szBuffer);
    If AGWSocket <> Nil then begin
       AGWSocket.SendText(szBuffer);
    end else begin
       PutAGW(1,'AGW Frame IGNORED because AGWPE is not connected');
    end; {endif}
end;
```

Complex?…. A little, however once invested here look what means to send an unproto beacon to all available ports….

```
  …..
    sMessage := 'Hello World!'+Chr($0D);
```

```
    For iPort = 0 to iMaxPorts do begin
            Self.Write(Chr(iPort),Chr($F0),'M',
                       sMyCall,'BEACON',Length(sMessage),sMessage);
    end;
  ....
```

There could be another couple of zillion ways to do this work (even far more efficiently) and everybody is encouraged to find it's own way.

Receive Frames

Receive and processing ***AGWPE*** frames isn't quantum physics nor rocket science, it's deceptively simple once a couple of issues are properly addressed.

In order to understand the proper way to process an ***AGWPE*** frame a golden rule must be understood.

TCP/IP doesn't guarantees the data would arrive at the destination blocked in the same way than was blocked on the transmision end. In other words, ***AGWPE*** could send in one end a perfectly formatted frame complete with header and data at once on a single TCP send call.

At the other end, however, and due to TCP and (specially) IP fragmentation factors, the data could be made available to the application as it's received and not necessarily as the same block that has been sent.

So the application must deal with the following situations.

- The block of data received isn't a complete frame.
    - A fragment of the header.
    - A complete header but incomplete data.
- The block of data received is a complete ***AGWPE*** frame.
- The block of data received contains more than one completed ***AGWPE*** frame.
- The block of data received contains several completed ***AGWPE*** frames and the fragment of one.

So, you could decide to follow this advice and save yourself many hours of frustration and debugging or find it by yourself the hard way…. ***"never assume anything about how the data arrives to the application"***

The recommended way to process ***AGWPE*** frames is to tackle that activity as three differenciated stages:

- Receive whatever arrives thru the TCP/IP connection and store it somewhere as a bulk of data without any attempt to extract any meaning of it (this "somewhere" should be some buffer, big enough to held several BIG ***AGWPE*** frames and persistent across different invocations to the receiving method).
- Every time data arrives to the application and is stored on the buffer examine this buffer with the following high level logic:
    - See if the buffer contains at least 36 bytes already, if not, just go do something else.
    - If at least 36 bytes exists a complete frame header had arrived, the frame might or might not have data.
    - Extract the fields of the header in a way that could be inspected individually.
    - If the frame has data associated see if there are DataLen bytes after the header at the

Buffer. If not the frame is not yet completed, leave.
    - If the frame has DataLen=0 or DataLen<>0 and there are DataLen+36 bytes on the buffer then extract the data from the buffer.
    - Format a complete header as discrete variables (Port, DataKind, Pid, From, To and Len) and the Data Area as a block.
    - Pass it to a frame handling routine.
    - If after the removal of the frame just processed there are 36 bytes or more still on the buffer there is a good chance that another frame is ready for processing, so call recursively this routine to process it.
- A decoding routine should cascade thru a switch or case structure where every relevant DataKind is handled, frame types not relevant to the application are then ignored.

This is the method recommended by George (***SV2AGW***)

```
Don't assume that you will receive a complete frame,TCPIP may send to your program part
of a frame or more than a frame so the procedure for reading data is like reading from
a file.Read only what you need. Like

A complete frame is HEADER+DATA  or just HEADER with no data
1.check to see if in the stream socket there are at least HEADER bytes. If not then
return
2.Examine the header and  the DataLen field
3.If the DataLen field is greater than 0 check to see if there are in the stream socket
DataLen bytes.
4.If there are DataLen bytes then read exactly DataLen bytes no more, otherwise wait
until DataLen bytes are available.
5.go to step 1 again till all the frames read.

Follow these steps carefully. If your application is running in the same machine with
agw packet engine then the usal is that you will receive more than a frame ,if the
monitor traffic is large.
```

Follow some of the routines used by Pedro (***LU7DID***) written in Delphi4 with the same purpose.

This routine is the OnRead event (some TCP/IP implementation calls it OnDataAvailable) handler, it's activated anytime data is ready to be processed from TCP/IP, this is the first stage recommended previously.

```
(*--->>> AGWSocketRead        <<<----------------------------------------*)
{* Receives the OK from the connection request                          *}
{*---------------------------------------------------------------------*}
procedure TLink.AGWSocketRead(Sender: TObject;
                              Socket: TCustomWinSocket);
  Var
    szData      :       String;

begin
    PutAGW(3,'Data Available from AGWPE <event>');

    szData := Socket.ReceiveText;
    DumpHex(3,szData);

    If Length(szData) <> 0 then begin
       Store(szData);
    end; {endif}

end;
```

The following method decides whether or not completed frames are ready for processing, the Decode routine is the one actually handling the different frames, see how the routine is called recursively.

```
(*--->>> Store         <<<-----------------------------------------------*)
{*Store the information from ***AGWPE*** and handles fragmentation issues     *}
{*---------------------------------------------------------------------------*}
Function TLink.Store (szReceived : String) : Boolean;
  Var
    szHeader    :       String;
    MSB         :       Integer;
    LSB         :       Integer;
    Index       :       Integer;

begin

  PutAGW(5,'Received from TCPIP');
  DumpHex(5,szReceived);

  AGW.Buffer := AGW.Buffer + szReceived;

  if AGW.Pending = FALSE then begin

     {*---------------------------------*}
     {*This is where it comes for a fresh*}
     {*packet from ***AGWPE***                 *}
     {*---------------------------------*}

     if Length(AGW.Buffer) >= AGW_HEADER then begin

        szHeader    := Copy(AGW.Buffer,1,AGW_HEADER);
        PutAGW(5,'Translated into Header Buffer');
        DumpHex(5,szHeader);


        If Length(AGW.Buffer) > 0 then begin
           AGW.Buffer := Copy(AGW.Buffer,AGW_HEADER+1,Length(AGW.Buffer)-AGW_HEADER);
        end; {endif}

        AGW.Data     := '';
        If szHeader[1] = Chr($31) then begin
           AGW.Buffer := Copy(AGW.Buffer,2,Length(AGW.Buffer)-1);
           Self.Store('');
           Result := TRUE;
           Exit;
        end; {endif}

        AGW.cPort    := szHeader[1];
        AGW.cPort    := Chr(ord(AGW.cPort)+1);

        AGW.DataKind := szHeader[5];
        AGW.cPID     := szHeader[7];

        AGW.CallFrom := '';
        AGW.CallTo   := '';

        For Index := 9  to 18  do begin
          If szHeader[Index] <> NUL then begin
             AGW.CallFrom := AGW.CallFrom + szHeader[Index];
          end else begin
             Break;
          end; {endif}
        end; {endfor}

        For Index := 19 to 28  do begin
          If szHeader[Index] <> NUL then begin
             AGW.CallTo  := AGW.CallTo   + szHeader[Index];
          end else begin
             Break;
          end; {endif}
        end; {endfor}
```

```
        LSB := ord(szHeader[29]);
        MSB := ord(szHeader[30]);

        AGW.DataLen := MSB*MAXFRAME + LSB;

       PutAGW(3,'Just Decoded as Port('+inttostr(ord(AGW.cPort))+')
Kind['+AGW.DataKind+'] {'+HexByte(Ord(AGW.cPID))+'} <'+AGW.CallFrom+
             '> <'+AGW.CallTo+'> Len('+inttostr(AGW.DataLen)+') + <<--Store');
                                    {*----------------------------------*}
                                    {*If DataLen is zero there is no    *}
                                    {*data, HOWEVER other frames could  *}
                                    {*be pending as well                *}
                                    {*----------------------------------*}
        If (AGW.DataLen = 0) then begin

                                    {*----------------------------------*}
                                    {*A frame were received and has no  *}
                                    {*data, we might call it a complete *}
                                    {*frame so store it on the port     *}
                                    {*object.                           *}
                                    {*----------------------------------*}

           AGW.Pending := FALSE;
                                    {*----------------------------------*}
                                    {*Store the frame on that port obj  *}
                                    {*----------------------------------*}

           PutAGW(3,'Decoded Port{'+inttostr(ord(AGW.cPort))+'}
DataKind['+AGW.DataKind+'] <'+AGW.CallFrom+'> <'+AGW.CallTo+'> {NO DATA}');
           Decode(AGW.cPort,AGW.cPID,AGW.DataKind,AGW.CallFrom,AGW.CallTo,0,'');

           Result       := TRUE;

                                {*----------------------------------*}
                                {*Wonder if something else came with*}
                                {*that frame and still in the buffer*}
                                {*----------------------------------*}

           If Length(AGW.Buffer) = 0 then begin

                                {*----------------------------------*}
                                {*Buffer is empty, see ya next time *}
                                {*                                  *}
                                {*----------------------------------*}
           end else begin
                                {*----------------------------------*}
                                {*Ooops, something else there       *}
                                {*recurse on myself to process      *}
                                {*----------------------------------*}
              Result := Self.Store('');
           end; {endif}


           Exit;

        end; {endif}

                                    {*----------------------------------*}
                                    {*If DataLen is NOT zero then       *}
                                    {*continue processing the buffer to *}
                                    {*see if we could extract the data  *}
                                    {*----------------------------------*}
        Result := Self.FragFrame;
        Exit;

     end; {endif}

  end else begin
```

```
     Result := Self.FragFrame;
     Exit;

  end; {endif}
  Result := TRUE;

end;
(*--->>> FragFrame     <<<-----------------------------------------------*)
{*Method to handle the likely fragmentation of frames over a TCPIP link *}
{*---------------------------------------------------------------------*}
Function TLink.FragFrame : Boolean;

begin { TLink }

                                        {*----------------------------------*}
                                        {*If DataLen is NOT zero then       *}
                                        {*continue processing the buffer to *}
                                        {*see if we could extract the data  *}
                                        {*----------------------------------*}
  If Length(AGW.Buffer) >= AGW.DataLen then begin

     AGW.Data  := Copy(AGW.Buffer,1,AGW.DataLen);

     If Length(AGW.Data) < Length(AGW.Buffer) then begin
        AGW.Buffer:= Copy(AGW.Buffer,Length(AGW.Data)+1,Length(AGW.Buffer)-
Length(AGW.Data));
     end else begin
        AGW.Buffer := '';
     end; {endif}

     AGW.Pending := FALSE;
                                        {*----------------------------------*}
                                        {*Store the frame on that port obj  *}
                                        {*instance                          *}
                                        {*Same solution than previous       *}
                                        {*----------------------------------*}

     PutAGW(3,'Decoded (Frag) Port{'+inttostr(ord(AGW.cPort))+'}
DataKind['+AGW.DataKind+'] <'+AGW.CallFrom+'> <'+AGW.CallTo+'>
Len=('+inttostr(AGW.DataLen)+') Data: '+AGW.Data);

Decode(AGW.cPort,AGW.cPID,AGW.DataKind,AGW.CallFrom,AGW.CallTo,AGW.DataLen,AGW.Data);

     Result       := TRUE;

                                        {*----------------------------------*}
                                        {*Wonder if something else came with*}
                                        {*that frame and still in the buffer*}
                                        {*----------------------------------*}

     If Length(AGW.Buffer) = 0 then begin

                                        {*----------------------------------*}
                                        {*Buffer is empty, see ya next time *}
                                        {*                                  *}
                                        {*----------------------------------*}
     end else begin
                                        {*----------------------------------*}
                                        {*Ooops, something else there       *}
                                        {*recurse on myself to process      *}
                                        {*----------------------------------*}
        Result := Self.Store('');
     end; {endif}


     Exit;
```

```
  end else begin

     AGW.Pending := TRUE;
     Result      := FALSE;
     Exit;

  end; {endif}
  Result := TRUE;

end; { TLink }
```

Last, but not least, the actual actions for each frame received, this is a method very application specific so only the skeleton is provided as a sample (despite its length this is a conceptually simple skeleton).

```
(*--->>> Decode        <<<-----------------------------------------------*)
{*Procedure to decode, validate and route a received frame               *}
{*-----------------------------------------------------------------------*}
Procedure TLink.Decode  (cPort     : Char;
                         cPID      : Char;
                         cDataKind : Char;
                         szFrom    : String;
                         szTo      : String;
                         iDataLen  : DWORD;
                         szData    : String) ;
  Var
    szYourCall  :       String;
    szAuxStr    :       String;
    szInfo      :       String;
    szVIA       :       String;

    IsNew       :       Boolean;
    lpAux       :       PtrLink;

    MSB         :       Byte;
    LSB         :       Byte;
    SysDateTime :       TDatetime;

begin { TLink.Decode }

    szTo   := UpCaseStr(szTo);
    szFrom := UpCaseStr(szFrom);

                                              {*-----------------------*}
                                              {*Return of call registr.*}
{*-----------------------------------------------------------------------*}

    If cDataKind = 'X' then begin
       {* Validate our registrations *}
       Exit;
    end; {* endif *}

                                              {*-----------------------*}
                                              {*Outstanding Frames Call*}
{*-----------------------------------------------------------------------*}
    If cDataKind = 'Y' then begin
       {* Your code here… *}
       Exit;
    end; {endif}

                                              {*-----------------------*}
                                              {*Outstanding Frames Port*}
{*-----------------------------------------------------------------------*}
    If cDataKind = 'y' then begin
       {* Your code here … *}
       Exit;
    end; {endif}
```

```
                                            {*-----------------------*}
                                            {*MHEARD List            *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'H' then begin

       PutAGW(1,'AGW Message <H>');
       DumpHex(1,szData);

       szAuxStr := Parse(szFrom);

       PutAGW(1,'AGW Message <H> Parsed From is ('+szAuxStr+') Remaining('+szFrom+')');
       If szAuxStr <> '' then begin
          ParseHeard(cPort,szData);
       end; {endif}

       PutAGW(1,'AGW Message <H> End of Message');

       Exit;
    end; {endif}

                                            {*-----------------------*}
                                            {*PARAMS List            *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'g' then begin
       PutAGW(5,'AGW Message <g>');
       ParseParam(cPort,szData);
       PutAGW(3,'End of processing Message <g>');
       Exit;
    end; {endif}

                                            {*-----------------------*}
                                            {*AGWPE KISS Raw Frame   *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'K' then begin
       PutAGW(5,'AGW Message <K>');
       DumpHex(5,szData);
       {*------- Here Decode the Raw Frame ------*}
       RawDecode(szData,szFrom,szTo,szVIA,cDataKind,cPID,iDataLen,szInfo);
       szData := szInfo;
       Exit;
    end; {endif}
                                            {*-----------------------*}
                                            {*AGWPE UNPROTO          *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'U' then begin
       PutAGW(5,'AGW Message <U>');
       {* Your code here….*}
       Exit;
    end; {endif}

                                            {*-----------------------*}
                                            {*AGWPE Radio Ports      *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'G' then begin
       PutAGW(5,'AGW Message <G>');
       ParsePort(szData);
       Exit;
    end; {endif}

                                            {*-----------------------*}
                                            {*AGWPE Version          *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'R' then begin
       PutAGW(5,'AGW Message <R>');

       If VersionFlag = FALSE then begin
```

```
         VersionFlag := TRUE;

         If Length(szData) >= 8 then begin
            LSB := ord(szData[1]);
            MSB := ord(szData[2]);
            AGWVerHigh := MAXFRAME * MSB + LSB;

            LSB := ord(szData[5]);
            MSB := ord(szData[6]);
            AGWVerLow  := MAXFRAME * MSB + LSB;

            PutAGW(1,'AGWPE Version '+inttostr(AGWVerHigh)+'-'+inttostr(AGWVerLow));

         end; {endif}
      end; {endif}

      Exit;
   end; {endif}

                                          {*-----------------------*}
                                          {*CONNECT Event Handler  *}
{*-------------------------------------------------------------------------*}
   If cDataKind = 'C' then begin
      PutAGW(5,'AGW Message <C>');
      IsNew      := FALSE;
      szYourCall := '';

      If Pos('CONNECTED To',szData) <> 0 then begin
         szAuxStr  := Parse(szData);
         szAuxStr  := Parse(szData);
         szAuxStr  := Parse(szData);
         szAuxStr  := Parse(szData);
         szYourCall:= Parse(szData);
         IsNew     := TRUE;
         PutAGW(1,'Connection initiated by station '+szYourCall);
      end; {endif}

      If Pos('CONNECTED With',szData) <> 0 then begin
         szAuxStr  := Parse(szData);
         szAuxStr  := Parse(szData);
         szAuxStr  := Parse(szData);
         szAuxStr  := Parse(szData);
         szYourCall:= Parse(szData);
         IsNew     := FALSE;
         PutAGW(1,'Connection initiated by us with station '+szYourCall);
      end; {endif}

                                          {*-----------------------*}
                                          {*We are looking for a   *}
                                          {*connection we started  *}
{*-------------------------------------------------------------------------*}
      If IsNew = FALSE then begin

         {* Your code here to handle a connection started by us *}

      end; {endif}

                                          {*-----------------------*}
                                          {*We are looking to serve*}
                                          {*an unsolicited connect *}
{*-------------------------------------------------------------------------*}
      If IsNew = TRUE then begin

         {* Somebody connected us, so handle it….*}
      end;

   end; {endif} {*-- This is the Footer of the whole <C> event handler --*}
```

```
                                          {*-----------------------*}
                                          {*DISC    Event Handler  *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'd' then begin
                                          {*First look at IC link  *}
       PutAGW(1,'Disconnect Frame: '+szData);

       If ((Pos('DISCONNECTED',szData) <> 0)  and
           (Pos('RETRYOUT',szData)     <> 0)) then begin
           PutAGW(2,'Disconnection by Retryout detected');

       end else begin
          If ((Pos('DISCONNECTED',szData) <> 0)  and
              (Pos('From',szData)         <> 0)) then begin
              PutAGW(2,'Normal Disconnection detected');
          end else begin
             If ((Pos('DISCONNECTED',szData) <> 0)  and
                 (Pos('With',szData)         <> 0)) then begin
                PutAGW(2,'AbNormal Disconnection detected, swap From<->To');
                szAuxStr := szFrom;
                szFrom   := szTo;
                szTo     := szAuxStr;
             end else begin
                PutAGW(2,'Bogus Disconnection form detected, ignored');
                Exit;
             end; {endif}
          end; {endif}
       end; {endif}

       {* Process the remaining of the disconnection *}
       Exit;
    end; {endif}

                                          {*-----------------------*}
                                          {*DATA    Event Handler  *}
{*-------------------------------------------------------------------*}
    If cDataKind = 'D' then begin

       {* DO whatever it fits with the Data from a connected partner *}
       Exit;
    end; {endif}

    PutAGW(3,'AGW Message not processed <'+cDataKind+'>');

end; { TLink.Decode }
```

Format VIA Areas

The way a digipeater string is informed to ***AGWPE*** seems intrincated and terrible on first look, but it's rather simple actually, see on how to create the data area in C++ (code excerpt from George, ***SV2AGW***).

```
char str[100];

str[0]=HowManyDigis;
str+1=1digi;//null terminated
str+1+10=2digi;//null terminated
str+1+20=3digi//null terminated
```

A more comprehensive routine to prepare a VIA list in C++ (also from George, ***SV2AGW***) follows

```
int PrepareViaList(char *InVia,char *OutVia)
{
//InVia string contains the via list like SV2AGW,SV2BBO,SV2DFK
//OutVia is the same list suitable prepared for AGWPE
char *token;
char temp[15];
char HowVia=0;
token=strtok(InVia,", ");
if (token)
   {
   HowVia++;
   strcpy(temp,token);
   memmove(OutVia+1,temp,10);
   }
for (;;)
        {
        token=strtok(NULL,", ");
        if (token)
        {
        strcpy(temp,token);
           memmove(OutVia+1+(HowVia*10),temp,10);
           HowVia++;
        } else break;
        }//end for
OutVia[0]=HowVia;
return((HowVia*10)+1);
}
```

The following function does about the same, only in Delphi4 and written by Pedro (***LU7DID***); the routine accepts a string with the list of callsigns+SSID to be used as digipeaters and returns a data area directly in the format ***AGWPE*** likes it.

```
(*--->>> FormatVIA     <<<-----------------------------------------------*)
{*Routine to format the VIA path in the particular way AGWPE likes it   *}
{*-------------------------------------------------------------------------*}
Function FormatVIA (InStr : String) : String;
  Var
    AuxStr      :       String;
    CountVIA    :       Byte;
    ArrayVIA    :       Array[0..256] of Char;
    Index       :       Integer;
    Token       :       String;
    PtrVIA      :       Byte;
    Jndex       :       Integer;

begin { FormatVIA }

    For Index := 0  to 256  do begin
        ArrayVIA[Index] := NUL;
    end; {endfor}

    AuxStr   := InStr;
    PtrVIA   := 1;
    CountVIA := 0;

    While AuxStr <> '' do begin
      Token := Parse(AuxStr);
      If Token <> '' then begin
         For Jndex := 1  to Length(Token)  do begin
             ArrayVIA[Jndex-1+PtrVIA] := Token[Jndex];
         end; {endfor}
         ArrayVia[Length(Token)+1+PtrVIA] := NUL;
         PtrVIA := PtrVIA + 10;
         inc(CountVIA);
```

```
      end; {endif}
    end; {endwhile}

    If PtrVIA <> 1 then begin
       AuxStr      := '';
       ArrayVia[0] := chr(CountVIA);

       For Index := 0  to PtrVIA-1  do begin
           AuxStr := AuxStr + ArrayVia[Index];
       end; {endfor}
       Result := AuxStr;
       Exit;
    end else begin
       Result := '';
       Exit;
    end; {endif}

end; { FormatVIA }
```

Parsing Port Information

This code excerpt written in Delphi4 shows how to “chain” port information sent by ***AGWPE*** with request for further information to ***AGWPE***, this routine/method should be originally called from the main dispatcher switch when a frame with DataKind=’G’ is received.

```
(*--->>> ParsePort     <<<-----------------------------------------------*)
{*Procedure to decode, validate and store information about ports       *}
{*---------------------------------------------------------------------*}
Procedure TLink.ParsePort(szPort : String);
  Var
  PortNum       :       Integer;
  AuxStr        :       String;
  RadioPort     :       Integer;
  cPort         :       Char;
  bPort         :       Byte;
  Index         :       Integer;

begin { TLink }

  AGWPortInit;

  PutAGW(3,'Port String is '+szPort);
  PortNum := strtoint(PopBang(szPort,';'));
  PutAGW(3,'Parsed Number of Ports is '+inttostr(PortNum));
  RadioPort := 1;

  While PortNum > 0 do begin
                                        {*---------------------------------*}
                                        {*Store the port description        *}
                                        {*---------------------------------*}
    AuxStr := PopBang(szPort,';');
    Ports[RadioPort].PortStr := GetStrZ(AuxStr);
    Ports[RadioPort].Enabled := TRUE;

                                        {*---------------------------------*}
                                        {*Initialize the MHEARD structure   *}
                                        {*for the port and pull a refresh   *}
                                        {*---------------------------------*}
    bPort := Trunc(RadioPort);
    cPort := chr(bPort);

    For Index := 1  to MAXHEARD  do begin
        Ports[RadioPort].HeardStr[Index] := '';
    end; {endfor}
```

```
                                        {*---------------------------------*}
                                        {*Pull outstanding info, heard and  *}
                                        {*capabilities info                 *}
                                        {*---------------------------------*}

    Send(AGWSocket,cPort,NUL,'y','','',0,'');
    Send(AGWSocket,cPort,NUL,'H','','',0,'');
    Send(AGWSocket,cPort,NUL,'g','','',0,'');

    PutAGW(3,'  Stored RadioPort # '+inttostr(RadioPort)+' as
+Ports[RadioPort].PortStr);
    inc(RadioPort);
    dec(PortNum);
  end; {endwhile}
```

Port Capabilities

This is the C++ structure recommended by George (***SV2AGW***) to held and decode the data area with the port capabilities as provided in the ‘g’ Frame sent by ***AGWPE***.

```
unsigned char OnairBaud;
unsigned char TrafficLevel;// if    this is 255 then the port is not in autoupdate mode
unsigned char TxDelay;
unsigned char TxTail;
unsigned char Persist;
unsigned char Slottime;
unsigned char maxframe;
unsigned char AX25Channels;  // How many connections we have
unsigned int HowManyBytes;// how many bytes are received the last 2 minutes
```

An alternate way, this time in Delphi4 from a code excerpt from Pedro (***LU7DID***) obtain this information right out of the data area as received in the ‘G’ frame from ***AGWPE***.

```
(*--->>> ParseParam    <<<-----------------------------------------------*)
{*Procedure to decode, validate and store information about ports params*}
{*-------------------------------------------------------------------------*}
Procedure TLink.ParseParam(cPort : Char; szData : String);

  Var
    bPort       :       Byte;
    szAuxStr    :       String;

begin { TLink }

  bPort := Ord(cPort);
  If (bPort > 0) and (bPort <= MAXPORT) then begin
  end else begin
     PutAGW(1,'ParseParam exit with cPort('+inttostr(bPort)+') too high');
     Exit;
  end; {endif}

  PutAGW(3,'Storing Params for Port ('+inttostr(bPort)+')');
  Ports[bPort].bOnAirBaud     := ord(szData[1]);
  Ports[bPort].bTrafficLevel  := ord(szData[2]);
  Ports[bPort].bTxDelay       := ord(szData[3]);
  Ports[bPort].bTxTail        := ord(szData[4]);
  Ports[bPort].bPersist       := ord(szData[5]);
  Ports[bPort].bSlotTime      := ord(szData[6]);
  Ports[bPort].bMaxFrame      := ord(szData[7]);
  Ports[bPort].bAX25Channel   := ord(szData[8]);
  Ports[bPort].iHowManyBytes  :=    ord(szData[9])+MAXFRAME*ord(szData[10])+
($10000*ord(szData[11]));
```

```
end; { TLink }
```

Heard Information for a Port

The following Delphi4 code excerpt from Pedro (***LU7DID***) shows a possible way to decode the heard information provided on the data area of an 'H' frame (note that the data associated with the binary format is just ignored).

```
(*--->>> ParseHeard    <<<-------------------------------------------*)
{*Procedure to decode, validate and store information about ports       *}
{*-----------------------------------------------------------------------*}
Procedure TLink.ParseHeard(cPort : Char; szData : String);
  Var
    bPort       :       Byte;
    iPort       :       Integer;
    Index       :       Integer;
    szAuxStr    :       String;
    szHeardStr  :       String;

begin { TLink }

  bPort := Ord(cPort);
  If (bPort > 0) and (bPort <= MAXPORT) then begin
  end else begin
     PutAGW(1,'ParseHeard exit with cPort('+inttostr(bPort)+') too high');
     Exit;
  end; {endif}
  iPort := bPort;

  PutAGW(1,'ParseHeard:');
  DumpHex(1,szData);

  szAuxStr   := szData;
  szHeardStr := PopBang(szAuxStr,Chr($00));

  For Index := 1  to MAXHEARD  do begin
      If Ports[iPort].HeardStr[Index] = '' then begin
         Ports[iPort].HeardStr[Index] := szHeardStr;
         PutAGW(1,'Stored '+szHeardStr+' at Offset ('+IntToStr(Index)+')');
         Exit;
      end; {endif}
  end; {endfor}
  PutAGW(1,'Table Full. Not Stored '+szHeardStr);

end; { TLink }
```

Raw Frames

In case raw frames want to be handled the following Delphi4 routine written by Pedro (***LU7DID***) could give you a starting point.

You must enter the routine (method, actually) with the data frame as received over the air on the 'K' frame from ***AGWPE***, the routine would parse and decode the components of that frame.

```
(*--->>> RawDecode     <<<-----------------------------------------------*)
{*Procedure to decode a Raw KISS frame                                    *}
{*-----------------------------------------------------------------------*}
Procedure TLink.RawDecode(    szData    : String;
                          Var szFrom    : String;
                          Var szTo      : String;
                          Var szVIA     : String;
                          Var cDataKind : Char;
                          Var cPID      : Char;
                          Var iDataLen  : DWORD;
                          Var szInfo    : String);
  Var
    Index       :       Integer;
    AuxChar     :       Char;
    AuxByte     :       Byte;
    szCall      :       String;
    Jndex       :       Integer;
    iCount      :       Integer;
    iFlag       :       Integer;

begin { TLink }

    Index   := 2;
    szCall  := '';
    iCount  := 1;
    iFlag   := 1;
    szVIA   := '';
    For Jndex := Index  to Length(szData)  do begin
       AuxChar := szData[Index];
       AuxByte := Ord(AuxChar);
       szCall := szCall + Chr(((AuxByte and $FE) shr 1));
       inc(iCount);
       If iCount > 7 then begin
          iCount := 1;
          If iFlag = 1 then begin
             szTo   := szCall;
             szCall := '';
             inc(iFlag);
          end else begin
             If iFlag = 2 then begin
                szFrom := szFrom;
                szCall := '';
                inc(iFlag);
             end else begin
                szVIA  := szVIA + szCall;
                szCall := '';
             end; {endif}
          end; {endif}
       end; {endif}
       inc(Index);
       If (AuxByte and $01) = $01 then begin
          Break;
       end; {endif}
    end; {endfor}

    szCall    := szFrom;
    szFrom    := Parse(szCall);
    szCall    := szTo;
    szTo      := Parse(szCall);

    AuxChar   := szData[Index];
    AuxByte   := Ord(AuxChar);
    If (AuxByte and $FE) = $00 then begin
```

```
       cDataKind := 'I';
    end else begin
       If (AuxByte and $FC) = $01 then begin
          cDataKind := 'S';
       end else begin
          cDataKind := 'U';
       end; {endif}
    end; {endif}

    inc(Index);
    cPid      := szData[Index];
    inc(Index);

    szInfo := '';
    For Jndex := Index  to Length(szData)  do begin
        szInfo := szInfo + szData[Jndex];
    end; {endfor}

    PutAGW(2,'Decoded KISS Frame as From=('+szFrom+') To=('+szTo+') VIA('+szVIA+')
DataKind('+cDataKind+') Pid('+HexByte(Ord(cPid))+')');
    DumpHex(2,szInfo);

end; { TLink }
```

Tracking Frames

Some of the frames used for the application to communicate with AGWPE acts like a switch, the first time sent activates a function, the second de-activates it and so on (i.e. the ‘m’ and ‘k’ frames).

Since AGWPE doesn’t return a “confirming” frame to the requirement the application has to keep track of the current status if for functional reasons the underlying functions must be alternated as on or off.

A good technique of doing so is to use a “counter” to track how many frames of a given type has been sent, at any time the value of that counter could be inspected and being odd it would signal the function is activated while being even it’s inactive.

The general algorithm would be:

- Everytime the connection with AGWPE is established (at the beginning of the execution or because of a re-connection) the counter bFrame must be set to zero (0x00);
- Everytime a frame under traking is sent the counter must be increased by one and “AND”ed with 7 (0x07 or 0b00000111).
- At any time the value of the counter being even (i.e. multiple of 2) would signal the function is inactive (0,2,4,….) while being odd (i.e. not multiple of 2) would signal the function is active (1,3,5,…).

Managing Connections

Once the basic frame processing is mastered just one additional issue has to be understood for the would be ***AGWPE*** programmer in order to be ready to write applications of any arbitrary complexity, and that is how to manage AX.25 connections.

Simple applications that involves the processing of some sort of monitoring only doesn't need to be bothered by the managing of connections at all, those applications doesn't even need to register a call to work properly.

Simple terminal programs could assume that a single user could sustain a single connection using a single registered callsign, so no complex issues occurs in that scenario.

However, most real world applications would face the need to be able to manage multiple simultaneous connections possibly using several registered callsigns at once.

The basic issue to solve is assuming a single TCP/IP connection is held between the application and ***AGWPE*** different frames might arrive thru that connection which logically belongs to different connections and the application is responsible to identify to which particular connection the frame belongs and to route the relevant information to that connection.

Unfortunately ***AGWPE*** doesn't provide any handle or id that uniquely identify a given connection among others; this is not needed actually since a close look of the AX.25 protocol provides such unique identifier as we would see.

Some suggested tactics would be discussed on the following sections.

One Callsign, Many Connections

This is the simplest case, so it's a good starting point.

The application needs to sustain several AX.25 connections at the same time (i.e. a multiwindow terminal program) using a single registered callsign.

At first look, the initial attempt of many programmers should be to create different TCP/IP sockets one for each connection, so frames flowing from ***AGWPE*** to the application would be automatically bucketed into their destination by means of the socket where the data arrived.

Given the relative simplicity to establish TCP/IP sockets from the application standpoint and the fact that ***AGWPE*** could handle a limitless number of simultaneous connections (from any meaningful number required in the real world) this seems to be the right approach.

But it is not, the main drawback is in the fact that ***AGWPE*** only allows a given callsign+SSID to be registered once across all applications running on a given moment, ***AGWPE*** doesn't really knows nor care if the multiple sockets were opened from a single application or from many applications.

So the first socket to register a callsign+SSID would take it all, the others would either fail trying to register the same callsign+SSID or be forced to use different callsign+SSID.

To register a different callsign+SSID for each connection could be both extremely impractical and a very limited approach, after all AX.25 tolerates just 16 different SSID to be used by callsign (0 to 15),

and with a fair number of applications running on a typical environment this would quickly become a limiting factor.

So, the best solution would be for the application to open just a single socket with ***AGWPE***, register a single callsign with it and manage many connections with the same callsign+SSID.

A limitation of the AX.25 protocol L2 connections come to provide a sort of help, the protocol doesn't allow more than one connected session among a pair of callsign+SSID ends, so the combination From/To (in any order) of any two callsign+SSID has to be unique. Since ***AGWPE*** could handle multiple ports at the same time the uniqueness could be obtained adding the Port to the identification key.

The overall logic the application should follow is despicted as follows:

- The application should register a callsign+SSID as a part of the initialization cycle.
- A memory structure (a table, a linked list, a double linked list, whatever) should be created and maintaned by the application with Port,From,To and status. This table would be initially empty. Additional information closely tied to the application functionality should also be included on each entry.
- Everytime a 'C' frame arrives from ***AGWPE*** as an unsolicited connection an entry is created, in the case the connection had been requested by us the entry might be created at the moment the connection was requested and the status updated when the connection is actually acomplished. The application should check at this point if another connection is already active with the same callsign+SSID pairs and if so refuse to start another connection (that would fail in the AX.25 realm anyway).
- Everytime a 'D' frame arrives from ***AGWPE*** the application scans the memory structure looking for a match with the Port/From/To and Port/To/From of the arriving data frame, if an entry is found the application could use the additional (functionality dependent) information to properly route the data block (object pointers, window handler, whatever). If an entry is not found the frame is just discarded or the error is flagged in some meaningful way for the application context.
- Everytime a 'd' frame arrives from ***AGWPE*** the application scans for the frame looking to match the Port/From/To and Port/To/From of it, once an entry is found all the actions associated with the disconnection are made and the entry is either destroyed or flagged as inactive (so no additional data frames could be handled by it).

Since this logic would be exercised fairly often the programmer should pay premium attention to the efficiency of the creation, search and disposal of entries on the memory structure making them as efficient and fast as possible (or buy an umpteen Mhz Pentium IX iron to run it).

Many CallSigns, Many Connections

In this case the added complexity is the need for the application to register and use any arbitrary number of callsign+SSID (i.e. on a BBS program some callsigns to accept connections and some others to start forward sessions).

Again, multiple sockets could be opened, in this case one for each registered callsign, so the case trivialized itself into many instances of the "One CallSign, Many Connection" just seen in the previous section.

However, the very logic stated in that case would still be valid for this one, as long as we add a validation on every connection started that the our involved callsign belongs to a given (and limited)

authorized set or pool of callsigns.

This control won't hurt, actually is highly desirable, even on the case a single callsign is used.

Down the Tubes, Climb the Ladder

In fantastic Wonderland bug free programs do exist; however, on Earth this is not true. Programs does have bugs, plenty of them. Some inocuous enough to survive the entire lifespan of a program and still be undetected (only triggered for a combination of factors so unlikely that it actually never happens).

AGWPE itself is not an exception, and certainly no application program would be out of that rule.

So the ideal profile of a programmer is a person that it's moderately paranoid and expect things that might go wrong from time to time; good programs often differentiate from bad programs that provide exactly the same functionality just by the extend of how hardened it has been made by it's author.

Some general guidelines:

- ***AGWPE*** is in general terms an stable platform, it could run days or even weeks under heavy use without any major trap or problem. However, from time to time, it might fail. Prepare your application to handle unexpected disconnections from ***AGWPE*** and recover graciously from them.
- Whenever a disconnection do occur assume something wrong happened to ***AGWPE***, so don't assume anything about the state of execution, just re-init everything as if it would be a fresh start. ***AGWPE*** might fail not because of a resident bug of it, Windows 95/98/NT aren't themselves the most stable OS on Earth, sometimes a ill behaved application might push it down the tubes for no particular good reason (NEVER rule out that is YOUR application the guilty one!!!).
- Don't trust data you didn't generated, always extensively check data contents, data ranges, meaningful values of data that came into your application from the outside; plan for actions when something non-compliant does arrives, because eventually does arrives.
- The landmark of a novel programmer is to suspect something is wrong with the compiler when a program doesn't work; even if it could be true it's extremely unlikely on most cases. ***AGWPE*** is not a compiler grade bug-free but still is a platform in use by thousands of nodes across the globe. So if something doesn't seems to work correctly with your application double check, and then check again. ***AGWPE*** might react in obscure ways when feed with improperly formated frames or data.
- Keep your feets on the ground at all times, the ***AGWPE*** API might lead you think you have a T3 bandwidth at your disposal because everything is transferred so fast; however, if the phisical port still is 1200 bps over Packet Radio is little what ***AGWPE*** could do about, you are just overwhelming the internal buffers of ***AGWPE*** for no good reason and this won't make your data move faster. Feed data at volumes that could be handled within reason with the available port bandwidth, check reasonably often how big your queue is on ***AGWPE*** and act accordingly.
- Plan carefully the whole AX.25 cycle your application would use, provide for unexpected events to happen and still the application provide meaningful results on them.
- Plan for your application to fail miserably from time to time, even for reasons you don't have any reasonable clue about at the writing time; use all exception resources your language of choice provide to handle the "unthinkable" and still end gracefully and if possible leaving a trace of what happened.
- ***AGWPE*** could sustain substantial abuse in terms of the resources and information required by each application, but still request from ***AGWPE*** what you could actually handle. There is no point on requiring information (i.e. activate monitoring in both conventional and raw formats) you are not going to actually process. Assume your application would be part on a typical station of a multiapplication scenario (using application mixes you could not imagine probably), so

everytime you drag unneded resources you might be impacting other's application ability to run smoothly.

- Don't fall on the typical programmer attitude to consider the world divided into the persons who write applications and the idiots who uses them; applications are black boxes for end users and many things you take for granted because you know your program might be obscure and arcane to a third person, no matter how literate with computers this person might be, just because he lacks your "common knowledge". Be forgiving with your user interface and if possible think on the unthinkable, it might well happen.

Credits and other stuff

This document is the intellectual property of Pedro E. Colla (***LU7DID***) and George Rossopoulos (***SV2AGW***) and as such is a copyrighted piece subject to international laws covering intellectual property.

However, it's usage is free for radioamateur uses as long as the reader understand is using the material contained on it at his or her own risk, authors doesn't bear any responsibility on any damage direct or indirect been made thru the usage of this material.

Excerpts had been taken from other sources and the relevant copyright information provided when this has been done, the copyright over those parts remains on it's beholder.

This documentation could be understood as an annotated version of the original "AGW TCPIP Socket Interface" document originally written by George Rossopoulos (***SV2AGW***) and included as a documentation with thw ***AGWPE*** package.

AGWPE is copyright© of G.Rossopoulos (***SV2AGW***) who held it's property and the right to change any aspect of the functionality without prior notice.

---

[1] As a reference on a 233 Mhz Pentium it could take as little as 4% of the total processor time and less than 4 MBytes of memory (allocated) under a moderate to heavy port activity. Under idle port conditions the amount of CPU taken might be 0.5% or less.

[2] If (as an example) two 9k6 bps packet ports are serviced the minimum bandwidth between ***AGWPE*** and each application should be between 3 X 2 X 9K6 = 57K6 and 4 X 2 X 9k6 = 76K8, so a modern 56K dialup connection would be in the lower end to service such arrangement with reasonable performance, assuming the effective speed is about 56K bps and not something lower.

[3] TCP/IP addressing is not particularly difficult once understood, however, the author of any program that forces the user to deal with them should be prepared to expend a great deal of effort to make the configuration of it as user friendly and "foolproof" as possible.

[4] Some oddity of the API, while ***AGWPE*** number the ports in the order of creation starting with Port 1 the API reflects the ports in the same order but starting with 0 (zero). So, the port reflected as Port 1 in the ***AGWPE*** Properties dialog would be reflected as 0x00 on the API, Port 2 as 0x01, and so on.

[5] When the callsign plus ssid plus the ending null (0x00) requires less than 10 bytes the remaining bytes of the field are not guaranteed to be cleared, data after the null (0x00) must be handled as "garbage" and cleared out by the application program. C programmers would handle it nicely (ASCIIZ variables), Pascal programmers should handle it more carefully.

[6] An exception to this is when the application terminates a link with a 'd' frame and inmediately unregisteres the callsign with a 'x' frame, in this situation no further information regarding the fate of the disconnection is sent to the application. In fact, if the unregistration closely follows the 'd' frame in such a way that the frame doesn't have enough time to be sent the other end will be left "connected", the connection would be terminated then either by the remote end inactivity timeout or when a data exchange attempt is made.