FlexiPanel

FlexiPanel Protocol 3.0

User Interface Services Layer for Bluetooth

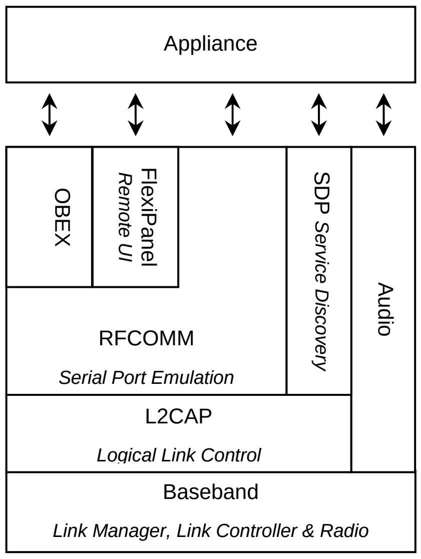
Summary

The FlexiPanel Bluetooth Protocol is a remote user interface service for computers, electrical appliances and other machinery.

A *FlexiPanel server* resides on the application and holds a user interface database that reflects the appliance's human-machine interface needs.

A *FlexiPanel client* can connect at any time, read the database and displays the user interface. A user may then control the application from the client device. Using Bluetooth, the client can be up to 330 feet away, without need for line-of-sight communication. FlexiPanel clients have been implemented on a range of PDAs and cellphones and are freely available.

Like many higher-level protocols such as OBEX file exchange, FlexiPanel sits on top of the RFCOMM serial port emulation layer of the Bluetooth protocol stack (see graphic below). It is not part of the "official" Bluetooth standard. However, the standard is relatively open in that anyone is free to create FlexiPanel clients, and FlexiPanel server licenses are modest.



The types of control that may be created, and the underlying data types they represent, are listed in the table below. Under the hood, the FlexiPanel Protocol is based on just a few basic types of message passed between client and server – these are the core of the protocol described in this document.

The main differences between the FlexiPanel Protocol and regular user interface services (such as Microsoft Windows) are:

Client Device Independence: The nature of the client's user interface may be unknown. The controls displayed will always be logically correct, but appearances may vary between different client devices, e.g. a cellphone and a PDA. If the client device can be anticipated in advance, certain additional preferences can be requested, such as a particular control layout or keyboard accelerators.

Fail Safe Performance: The connection might be broken at any time, for example if the client's batteries fail or the client goes out of range. The appliance must enter a fail-safe state if connection is lost at a critical moment.

Compact Server Code: FlexiPanel servers might be very small, low cost microcontrollers. Consequently system requirements on the server side must be extremely lean and communication very succinct. The remote client device takes over as many responsibilities as possible. For example, a server is not required to buffer any I/O, manipulate any floating point numbers or make any conversions between single-byte characters and Unicode.

Control type	Function / value
Button	Single-press event
Latch, check box, radio button	Binary value
Text	Character string
Number	Integer or fixed-point value
Matrix, chart	2-D array of numbers
Date, time	Seconds to years
List box	1-of-*n* selection
Section, popup menu	Arranges controls in a hierarchy
Password	Provides access control
Message box	Alerts user
Blob	Exchanges binary data
Files	Exchanges files
Image	Graphical image

FlexiPanel Protocol 3.0

Overview

The FlexiPanel protocol conveys user interface information between a server, which needs to interact with a user, and a client, which provides the user interface on behalf of the server. Server and client communicate by passing messages.

The FlexiPanel protocol transmits information about the 'logical' user interface, for example that a button or a list box is required. This information is quite independent of the client it connects to.

In addition, the server may choose to send client-specific information, for example the exact controls to use and how these are laid out. It is not required to do so.

The separation of the logical user interface from its layout information keeps servers 'future-proof', since if a new type of client is created, the user interface will be useable, if not ideal.

Most clients have been developed in-house by FlexiPanel Ltd. The publication of the protocol happen was retrospective and its main purpose is as a reference work for internal FlexiPanel use. It is possible that key conceptual points have been omitted because we have taken them for granted. Let us know if you require clarification.

In addition, transcription errors are possible. If you encounter an omission or error, please contact us to let us know and to get full clarification. In this way we can continually improve this document.

In all cases, stress-test your implementation of the protocol either using a FlexiPanel client product (for servers), or a FlexiPanel Designer simulation (for clients).

Message Based Protocol

Server and client communicate by passing one-way messages. These do not necessarily require a response or an acknowledgement.

Messages always begin with a 20-byte header detailing the message contents. This is followed by additional information if the message requires it.

Data Transmission

No error correction has been incorporated into the protocol because it is assumed that this will be managed by lower levels of the Bluetooth protocol.

All values are little-endian. For example, the Button control type is `0x0042`; this is transmitted as the byte `0x42` followed by the byte `0x00`.

The server may choose to use Unicode or ASCII characters to convey text. However, it must choose one or the other, it may not interchange them.

Message Header

All messages begin with a 20 byte header.

Data	Contains
char[8]	FlexiPanel identifier
uint16	Serial number
uint16	Message checksum
uint16	Protocol version number
uint16	Backward compatibility version number
uint16	Message type
uint16	Flags

FlexiPanel identifier

The byte values 0x48, 0xEF, 0xF0, 0x74, 0x72, 0xEF, 0xE6, 0x66. These allow the message parser to re-sync easily in the event of a synchronization error.

Serial number

Identifies the message source. A client may optionally identify itself as follows:

Constant	Message
`0x0000`	Unspecified
`0x0100`	FlexiPanel Client for Pocket PC
`0x0200`	FlexiPanel Client for Windows
`0x0300`	FlexiPanel Client for Smartphone
`0x0400`	FlexiPanel Client for Java Phones
`0x0500`	FlexiPanel Client for Windows API
`0x0600`	FlexiPanel Client for Palm

Servers are primarily identified by their title. They may optionally use the serial number to further identify the batch or serial number as desired.

Message verification

Message verification by checksum may be implemented. The checksum is optional and, since lower levels of the Bluetooth stack usually contains error correction, generally superfluous.

The calculation of the checksum is defined as follows: The checksum value should be such that when all even bytes in the message, header and body, are added together, the lowest eight bytes of the result will be zero. Likewise for all odd bytes.

There are two limitations to the checksum system. First, checksum verification requires the entire message to be read. However, data indicating the length of the message may themselves be corrupted which may cause storage problems. Second, if the error is due to a bug in the remote device, the checksum will be correct anyway. For both these reasons, a server or client may choose to reject a message simply because the memory required to read in the message is so large the resources are not available.

Protocol version number

Identifies the FlexiPanel protocol version number implemented on the sender. The value should be the same for all messages sent. This number increases in new product releases as the protocol definition changes:

Constant	Version
`0x0001`	FlexiPanel 2.x
`0x0002`	FlexiPanel 3.x

Backward compatibility version number

Identifies the lowest FlexiPanel protocol version which should be backwardly-compatible with this all messages from this device. The value should be the same for all messages sent from the device. Constants are the same as for the protocol version number.

If the message receiver's protocol number is lower than this value the message will be ignored (as of version 3.0) and, if the receiver is a client, it will inform the user to upgrade their software.

Message Type

Integer identifying the type of message contained in the message body.

Constant	Message
`0x0001`	Greetings from client
`0x0002`	Greetings from server
`0x0004`	Goodbye from client
`0x0005`	Goodbye from server
`0x0006`	New control panel from server
`0x0007`	Control update from client
`0x0008`	Control update from server
`0x0009`	Ping from server

Constant	Message
0x000A	Ping from client
0x000B	Ping reply from server
0x000C	Ping reply from client
0x000D	Acknowledge from server
0x000E	Acknowledge from client
0x000F	New server
0x0010	Control properties update from server
0x0011	Files from server
0x0012	Profile request from client
0x0013	Profile data from server
0x0081	Program control panel
0x0082	Program host

Flags

OR-able values identifying specific features about the message sender:

Constant	Meaning if set
0x0001	Unicode text strings
0x0002	Pings not supported
0x0004	Acknowledge not supported
0x0008	Acknowledge requested
0x0010	One-shot server
0x0020	Checksum field is valid
0x0040	Image controls can generate update messages when clicked on

The server specifies whether communication is Unicode (16-bit characters) or ASCII (8-bit characters). Clients must be prepared to handle both. The server must stick to one or the other, not switch between them, and the client must follow: *i.e.* all Unicode flags in messages headers and control blocks from the same server must be the same and the client must reply in the same format.

One-shot servers disconnect immediately after sending the New Control Panel From Server message. The data cannot be modified by the client and re-transmitted to the server. Unlike usual disconnection, the client should continue to display the controls after disconnection until dismissed by the user.

Worked example

A FlexiPanel Client for Pocket PC 3.0 sends a Unicode *Goodbye from Client* message as follows:

Bytes in order of transmission (hex)	Meaning
48,EF,F0,74, 72,EF,E6,66	FlexiPanel identifier
00,01	Pocket PC Client identifier
00,00	Checksum not implemented
02,00	Protocol version 3.0
01,00	Backwardly compatible to 2.0
04,00	Goodbye message ID
01,00	Flags – Unicode flag only

Greetings From Client

The Greetings From Client message tells a server that a client has connected over the Bluetooth link. It should be sent when a client first connects to a server or if a New Server message is received.

The Greetings From Client message initiates a FlexiPanel client/server link. It is sent as a header with message constant `0x0001` and no body.

In response to a Greetings From Client message, a server should send a Greetings From Server message in confirmation. It will almost certainly then send a New Control Panel From Server message.

A server shouldn't be concerned if it receives two Greetings From Client messages, since one may be due to usual connecting and one may be in response to an earlier New Server message that got stuck in a buffer somewhere.

The Greetings From Client message was implemented in protocol version 2.0 and remains current.

Worked example

A FlexiPanel Client for Pocket PC 3.0 sends a Unicode *Greetings from Client* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74,` `72,EF,E6,66`	FlexiPanel identifier
`00,01`	Pocket PC Client identifier
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`01,00`	Greetings message ID
`01,00`	Flags – Unicode flag only

Greetings From Server

The Greetings From Server message is a response to the Greetings From Client message and confirms that the device connected to is indeed a compatable FlexiPanel device. It is sent as a header with message constant `0x0002` and no body.

The Greetings From Server message confirms the initiation of a FlexiPanel client/server link. It is sent as a header with message constant `0x0002` and no body.

The Greetings From Server message should be sent when a server receives a Greetings From Client message. It will almost certainly then send a New Control Panel From Server message.

The Greetings From Server message was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Greetings from Server* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74,` `72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`02,00`	Greetings message ID
`01,00`	Flags – Unicode flag only

Goodbye From Client

The Goodbye From Client message indicates that a client wishes to terminate the FlexiPanel client/server link. It is sent as a header with message constant `0x0004` and no body.

No reply is required. Any reply received will be ignored.

Either client or server may terminate a link. It may also fail unannounced, e.g. if the client or server lose power or go out of range. Ping functionality, acknowledge functionality, or low-level interrogation of the Bluetooth driver, should be implemented in order to detect this condition and treat it in a fail-safe manner.

The Goodbye From Client message was implemented in protocol version 2.0 and remains current.

Here is a worked example so you are sure you understand it

Worked example

A FlexiPanel Client for Pocket PC 3.0 sends a Unicode *Goodbye from Client* message as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,01`	Pocket PC Client identifier
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`04,00`	Goodbye message ID
`01,00`	Flags – Unicode flag only

Goodbye From Server

The Goodbye From Server message indicates that a server wishes to terminate the FlexiPanel client/server link. It is sent as a header with message constant `0x0005` and no body.

No reply is required. Any reply received will be ignored.

Either client or server may terminate a link. It may also fail unannounced, e.g. if the client or server lose power or go out of range. Ping functionality, acknowledge functionality, or low-level interrogation of the Bluetooth driver, should be implemented in order to detect this condition and treat it in a fail-safe manner.

The Goodbye From Server message was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Goodbye from Server* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74,` `72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`05,00`	Goodbye message ID
`01,00`	Flags – Unicode flag only

New Control Panel From Server

The New Control Panel From Server sends a list of controls to be displayed by the client.

The New Control Panel From Server message is sent whenever a client first connects and whenever the controls list of controls changes. Minor changes to individual controls' values and properties do not require this message to be sent.

The New Control Panel From Server message was implemented in protocol version 2.0 and remains current.

The message starts with a header with message constant `0x0006`. The body that follows will vary depending on the controls to be displayed. However, the general format is guaranteed, allowing client devices to ignore controls and/or features it does not recognize or is not able to support. The general format is:

DataSize	Contains
uint32	number `nCtl` of *Control Blocks* to follow
then `nCtl` *Control Blocks*	

Control Block

The exact nature of a control block depends on the type of control it is. However, all control blocks conform to the following structure:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type `CtlTyp` as defined below
`uint32`	Control Flags `dwFlags` as below
`uint32`	Control Unique ID
`uint32`	Size `ValSz` of control value field, bytes
`ValSz`	Control Value
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that signals the contents of the *Descriptor Block*. *Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Control Z-Order

Image controls are depicted behind any controls which overlap them.

Control Type & Value

Each control has a value field whose contents may be modified by client or server. Their size and interpretation depend on the control type as follows:

Control	`CtlTyp` constant	Control Value
Button	`0x0042`	Registers a button press (only client can modify)
Text	`0x0054`	Zero terminated ASCII or Unicode text
Latch	`0x0047`	Binary value encoded as a byte 0x00 (false/off) or 0xFF (true/on)
Section	`0x0053`	Binary value encoded as a byte 0x00 (open) or 0xFF (closed)
Date/ Time	`0x0044`	An 8-byte date-time field
Number	`0x004E`	A 4-byte signed integer (descriptors provide decimal point adjustment)
Matrix	`0x004D`	A 2-D array of 1-, 2- or 4-byte integers; each row may have label data
Password	`0x0050`	Binary value encoded as a byte 0x00 (locked) or 0xFF (unlocked)
List	`0x004C`	A 4-byte signed integer indicating selection of single item from list
Message	`0x0058`	Zero terminated ASCII or Unicode message text
Blob	`0x004F`	Unrestricted binary transfer; primarily for providing URL links
Files	`0x0046`	File transfer; primarily for providing local HTTP file service
Image	`0x0049`	Color image

Control Flags – Generic

The control flags are features of the field which are bitwise OR-ed together to make the `uint32` value. Most flags are control-specific; however, the following flags apply all controls:

Name / `dwFlags` constant	Contains
`CTL_INVISIBLE` `0x00000001`	Control is not visible
`CTL_RIGHTTOLEFT` `0x00000002`	Right-to-left text (i.e. Arabic style) preferred if possible
`CTL_UNICODE` `0x00000004`	Text is Unicode (16-bit) rather than ASCII. Must be same as message header
`CTL_STARTGROUP` `0x00000008`	Control is start in a logical group which should be displayed together
`CTL_ENSUREVISIBLE` `0x00000010`	If control is not currently visible, make it visible now. (Only really applies in Control Properties Update messages)
`CTL_ENDGROUP` `0x00000020`	Control is the last in a logical group which should be displayed together
`CTL_COLCHANGED` `0x00000040`	(Only really applies in Control Properties Update messages).

Control flags are generally fixed throughout the life of a control. The following may change in a Control Properties Update From Server message:

- `CTL_INVISIBLE` may change to hide or show a control.
- `CTL_ENSUREVISIBLE` may be set to request that a control be brought into view.
- `CTL_MSG_ICON_` flags.
- `CTL_MSG_RESP_` flags.
- If the color is to be changed, the `CTL_COLCHANGED` flag should be set.

Control Unique ID

The control unique ID should be the only control that the server provides with that ID. In a client / server environment it possible that the client sends an update message relating a control which has since ceased to apply. The unique ID ensures that an update message can be applied to the control to which it was intended or, if no longer applicable, ignored. The ID should not be 0x00000000 nor greater than 0xFFFFFFF0.

Control Descriptor

Control-specific descriptors contain the remaining information required by a particular control. Each descriptor block consists of:

DataSize	Contains
uint32	Size `nDsc` of *descriptor block*
`nDsc`	Descriptor data

Descriptor blocks may come in any order but must follow the same order as they appear in the *Descriptor Identifier Header.* Most descriptors are optional; some, however, are required – generally those that indicate the size of other data elements.

Although the *Descriptor Identifier* values are printable ASCII characters, the *Descriptor Identifier Header* is not expected to contain a zero terminator – all *Descriptor Identifier* values should have a meaning.

The structure of the different control blocks will now be described. A worked example of an entire New Control Panel From Server message the follows right at the end of this section.

Button Control Block

The Button Control is usually represented on a client as a button. It has no retained state as such but is a user-initiated event, *i.e.* the button is pressed. The event is communicated from client to server as a Control Update From Client message, which is always associated as a press event. A Control Update From Server message for a button would be meaningless.

The Button Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0042`

`uint32`	Generic control flags plus *Button Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `0x00000001`
`0x01`	*Button Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Button Value

The button value is a single byte although the value is ignored.

Button Flags

The following control flags are specific to the button control and may also only apply to certain servers:

Name / `dwFlags` constant	Meaning
`CTL_BTT_RESET` `0x00010000`	Entire server product should reset when button pressed

Name Descriptor Block

The *Name Descriptor Block* defines the title text that appears on the button. It is not actually required but the button may be nameless without it. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
`uint32`	Size `nNam` of *Name* in bytes
`nNam`	Button name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends a Unicode “Reset Host” button control block as follows:

Bytes in order of transmission (hex)	Meaning
`2B,00,00,00`	Size of fields to follow (43 bytes)
`42,00`	Control type, value `0x0042`
`04,00,01,00`	Specifies control flags `CTL_BTT_RESET` and `CTL_UNICODE`
`54,53,52,00`	Control ID `0x00525354`
`01,00,00,00`	Control value size = 1
`00`	Control value
`02,00,00,00`	2 descriptor identifiers follow, 1 byte each
`5A,63`	Descriptor identifiers: Name , followed by Color
`0A,00,00,00`	Size of name descriptor data in bytes
`52,00,65,00,73, 00,65,00,74,00`	Text “Reset” in Unicode
`04,00,00,00`	Size of color descriptor data in bytes
`FF,00,00,00`	Bright red button requested

Text Control Block

The Text Control stores variable length text up to a maximum length. Single-byte (ASCII) and Unicode text can be stored, although FlexiPanel Clients may not know how to represent all Unicode characters.

The Text Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0054`
`uint32`	Generic control flags plus *Text Flags* described in this below
`uint32`	Control Unique ID
`uint32`	Control Value Size `szTxtSz`, equal to the maximum number of *bytes* (not characters) that can be stored, including zero terminator.
`szTxtSz`	*Text Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Text Value

The text value is the current contents of the text control including zero terminator. This may be ASCII or Unicode, according to the *Control Flags.* `szTxtSz` bytes are transmitted, which must be equal to or less than the number of bytes specified in the Length Descriptor Block.

Text Flags

The following control flags are specific to the text control:

Name / `dwFlags` constant	Meaning
`CTL_TXT_ MODIFIABLE 0x00010000`	Text may be modified by the client, *e.g.* by providing an edit text control
`CTL_TXT_ PASSWORD 0x00020000`	Modifiable text should have secure entry, *e.g.* a password style edit control

Name Descriptor Block

The *Name Descriptor Block* defines the title text which describes what the text data represents. It is not required but from version 2.3 it is expected as a way to describe the information the control represents. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Text control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Length Descriptor Block

The *Length Descriptor Block* is the maximum length in bytes that the text is permitted to be, including zero terminator and must be equal or greater than `szTxtSz`. It is required. The *Descriptor Identifier* used to indicate the Length Descriptor Block is `0x6D`.

DataSize	Contains
uint32	*Length* size, value `0x00000004`
`0x04`	`szTxtSz`

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends a Unicode editable text control block, maximum 4 characters (=(4+1)*2=10 bytes), current value "Hi", as follows:

Bytes in order of transmission (hex)	Meaning
`36,00,00,00`	Size of fields to follow (54 bytes)
`54,00`	Control type, value `0x0054`
`04,00,01,00`	Specifies control flags `CTL_TXT_MODIFIABLE` and `CTL_UNICODE`
`54,58,54,00`	Control ID `0x00545854`
`16,00,00,00`	Control value size = 10
`48,00,69,00,00, 00,00,00,00,00`	Control value "Hi" in zero terminated Unicode
`02,00,00,00`	2 descriptor identifiers follow, 1 byte each
`5A,6D`	Descriptor identifiers: Name, followed by Length
`08,00,00,00`	Size of name descriptor data in bytes
`54,00,65,00,78, 00,74,00`	Text "Text" in Unicode
`04,00,00,00`	Size of length descriptor data in bytes
`0A,00,00,00`	Length 10 bytes

Latch Control Block

The Latch Control Block stores a binary on/off value. It would usually be represented by a check box or radio button but other representations are possible. For example, the Pocket PC has a 'Momentary Button' representation whose state is on only while the button is pressed.

The Latch Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0047`
`uint32`	Generic control flags plus *Latch Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `0x00000001`
`0x01`	*Latch Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block*. *Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Latch Value

The latch value is a single byte. `0x00` represents off and `0xFF` represents on.

Latch Flags

The following control flags are specific to the latch control:

Name / `dwFlags` constant	Meaning
`CTL_LCH_ RADIORESET 0x00010000`	Radio button has reset behavior (see below)

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the latch control value represents. For example, a check box uses this as the text to place next to the check square. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Latch control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Radio Button Descriptor Block

The *Radio Button Descriptor block* requests radio button behavior. The block describes a value called the Radio Group ID. This is interpreted as follows:

- If the Radio Button Descriptor Block is absent or the Radio Group ID value is zero, normal latch control behavior is assumed.

- If the Radio Group ID value is non-zero, all other latch controls with the same Radio Group ID should be placed in the off state when this latch enters the on state.

- If the Radio Group ID value is non-zero and the `CTL_LCH_RADIORESET` flag is set, all latch controls with the same Radio Group ID (including itself), should be placed in the off state when this latch enters the on state. Thus this control never stays in the on state when pressed and its only function is to turn off all others.

The *Descriptor Identifier* used to indicate the Radio Button Descriptor Block is `0x72`.

DataSize	Contains
uint32	*Radio Group ID* size, value `0x00000004`
`0x04`	Radio Group ID.

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an off Unicode “Radio 1” Radio Button latch control block as follows:

Bytes in order of transmission (hex)	Meaning
`30,00,00,00`	Size of fields to follow (48 bytes)
`42,00`	Control type, value `0x0042`
`04,00,01,00`	Specifies control flags `CTL_LCH_RADIORESET` and `CTL_UNICODE`
`54,53,52,00`	Control ID `0x00525354`
`01,00,00,00`	Control value size = 1
`00`	Control value (off)
`03,00,00,00`	3 descriptor identifiers follow, 1 byte each
`5A,72,63`	Descriptor identifiers: Name , Radio Group, Color
`0E,00,00,00`	Size of name descriptor data in bytes
`52,00,61,00,64, 00,69,00,6F,00, 20,00,31,00`	Text “Radio 1” in Unicode
`04,00,00,00`	Size of color descriptor data in bytes
`FF,00,00,00`	Bright red latch requested

Section Control Block

The Section Control Block stores a binary open/closed value relating to “child controls” which are only communicated and displayed when the section control is open. It is used to reduce communication overload and display real estate.

A change of state of a section control would usually result in the server sending a New Control Panel From Server message; however, it entirely up to the server which controls are displayed in each state. From the client’s perspective, the control is logically the same as a latch control but will probably be displayed differently.

The Section Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0053`
`uint32`	Generic control flags plus *Section Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `0x00000001`
`0x01`	*Section Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that

	indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Section Value

The section value is a single byte. `0x00` represents closed and `0xFF` represents open.

Section Flags

The following control flags are specific to the section control:

Name / `dwFlags` constant	Meaning
`CTL_SCT_ AUTOCLOSE 0x00010000`	Server always returns the Section control to the closed state when a client disconnects

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the section contains. For example, a section containing controls which are used for rarely modified settings might be named "Settings…" The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Section control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an open, automatically closing, ASCII "Settings" section control block as follows:

Bytes in order of transmission (hex)	Meaning
`20,00,00,00`	Size of following fields (32 bytes)
`53,00`	Control type, value `0x0053`
`00,00,01,00`	Specifies control flag `CTL_SCT_AUTOCLOSE`
`54,45,53,00`	Control ID `0x00534554`
`01,00,00,00`	Control value size = 1
`FF`	Control value (open)
`01,00,00,00`	1 descriptor identifier follows, 1 byte long
`5A`	Descriptor identifier: Name
`08,00,00,00`	Size of name descriptor data in bytes
`53,65,74,74,69, 6E,67,73`	Text "Settings" in ASCII

Date-Time Control Block

The Date-Time Control Block stores an 8-byte Date-Time value. It would usually be represented by formatted text or by Date and Time controls.

The Date-Time control is a little unusual in that both the server and client may wish to update the control at once. The situation arises because a real-time clock date-time control will be updated by the server every second. At the same time, the client may wish to set the clock time. Client software needs to ignore server updates while the client is in the process of modifying the value: real-time clock date-times controls should be tested when designing a FlexiPanel client.

The Date-Time Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control*

	block except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0044`
`uint32`	Generic control flags plus *Date-Time* Flags described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `0x00000008`
`0x08`	*Date-Time Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Date-Time Value

The 8-byte Date-Time value consists of the following fields:

Datatype	Contains	Range
byte	Second	0 – 59
byte	Minute	0 – 59
byte	Hour	0 – 23
byte	Date	1 – 31
byte	Day of week	0 – 6 Sunday to Saturday respectively 7 = Unknown
byte	Month	1 – 12
uint16	Year	0 – 65535

Depending on the interpretation of the control, not all Date/Time fields need be valid. For example, a date-time field may represent a birthday, in which case only date and month would be valid; an alarm time, in which hour and minute are valid, *etc*.

Date-Time Flags

The following control flags are specific to the date-time control:

Name / `dwFlags` constant	Meaning
`CTL_DTM_ MODIFIABLE 0x00010000`	At least one Date-time field is modifiable (must be specified in addition to other modifiable flags)
`CTL_DTM_ MODIFYSECS 0x00020000`	The seconds field is modifiable
`CTL_DTM_ MODIFYMINS 0x00040000`	The minutes field is modifiable
`CTL_DTM_ MODIFYHOURS 0x00080000`	The hours field is modifiable
`CTL_DTM_ MODIFYWEEKDAY 0x00100000`	The weekday field is modifiable. (Should not be set if any of date, month or year are modifiable.)
`CTL_DTM_ MODIFYDAY 0x00200000`	The date field is modifiable
`CTL_DTM_ MODIFYMONTH 0x00200000`	The month field is modifiable
`CTL_DTM_ MODIFYYEARS 0x00200000`	The year field is modifiable
`CTL_DTM_ REALTIMECLOCK 0x01000000`	The date-time value is a real-time clock, *i.e.* the server will advance the value to keep time

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the date-time control value represents. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Date-Time control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Format Descriptor Block

The *Format Descriptor block* expresses a preference as to how the date-time value is displayed as a text string. A client is not required to do so; in particular, it may not be practical for modifiable controls. The format descriptor is interpreted as the string of text to be displayed, after the following substitutions have been made:

Text in format string	Value substituted
`%H%`	Hour (24 hour, 1 or 2 digits)
`%HH%`	Hour (24 hour, 2 digits always)
`%h%`	Hour (12 hour, 1 or 2 digits)
`%hh%`	Hour (24 hour, 2 digits always)
`%m%`	Minute (1 or 2 digits)
`%mm%`	Minute (2 digits always)
`%s%`	Second (1 or 2 digits)
`%ss%`	Second (2 digits always)
`%d%`	Date (1 or 2 digits)
`%dd%`	Date (2 digits always)
`%ddd%`	Day of week (3 letter abbreviation)
`%dddd%`	Day of week (whole word)
`%M%`	Month (1 or 2 digits)
`%MM%`	Month (2 digits always)
`%MMM%`	Month (3 letter abbreviation)
`%MMMM%`	Month (whole word)
`%yy%`	Year (2 digits)
`%yyyy%`	Year (all digits)
`%t%`	A or P (for AM or PM)
`%tt%`	AM or PM

For example, to display the time

Mon, 9 Jun, 8:45 PM

use the format descriptor

`%ddd%, %d%. %MM%, %h%:%mm%%t%`

The *Descriptor Identifier* used to indicate the Format Descriptor Block is `0x66`.

DataSize	Contains
uint32	Size `nFmt` of *Format String* in bytes
`nFmt`	Format String in ASCII or Unicode as specified in message header.

The format string may or may not contain a zero terminator. If it does not, its length should be assumed to be `nFmt`

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an ASCII modifiable real-time date-time control block with format HH:MM named "Time" as follows:

Bytes in order of transmission (hex)	Meaning
`3A,00,00,00`	Size of fields to follow (58 bytes)
`44,00`	Control type, value `0x0044`
`00,00,0D,01`	Specifies control flags `CTL_DTM_MODIFIABLE`, `CTL_DTM_MODIFYMINS`, `CTL_DTM_MODIFYHOURS` and `CTL_DTM_ REALTIMECLOCK`
`45,4D,49,54`	Control ID `0x54494D45`
`08,00,00,00`	Control value size = 8
`21,2D,0D,11, 03,0B,D4,07`	Control value 13:45:33, date Wed, 17 Nov 2004
`03,00,00,00`	3 descriptor identifiers follow, 1 byte each
`5A,66,63`	Descriptor identifiers: Name, Format, Color
`04,00,00,00`	Size of name descriptor data in bytes
`54,69,6D,65`	Text "Time" in ASCII
`09,00,00,00`	Size of format string in bytes
`25,48,48,25,3A, 25,4D,4D,25`	Text "%HH%:%MM%" in ASCII
`04,00,00,00`	Size of color descriptor data in bytes
`FF,00,00,00`	Bright red latch requested

Number Control Block

The Number Control Block stores a 4-byte signed integer, *i.e.* in the range –2,147,483,648 to +2,147,483,647. It is usually represented as numerical text, possibly with spin or slider controls for data entry.

A *Mantissa* (decimal-shifting) descriptor value allows the value to represent fixed-point non-integer values. However, if such a decimal shift operation is specified, the number value (and related values such as minimum and maximum values) are expressed in terms of the underlying 4-byte signed integer value.

The Number Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x004E`
`uint32`	Generic control flags plus *Number Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `0x00000004`
`0x04`	*Number Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Number Value

The Number value is a 4-byte integer, transmitted least significant byte first (as are all multi-byte integers in FlexiPanel).

Number Flags

The following control flags are specific to the number control:

Name / `dwFlags` constant	Meaning
`CTL_NUM_ MODIFIABLE 0x00010000`	Number is modifiable
`CTL_NUM_MIN 0x00020000`	Number has a minimum permitted value
`CTL_NUM_MAX 0x00040000`	Number has a maximum permitted value
`CTL_NUM_FLOAT 0x00000000`	Use *printf* `%.lf` style formatting (floating point)
`CTL_NUM_ FIXEDPOINT 0x00010000`	Use *printf* `%.nnlf` style formatting (fixed point)
`CTL_NUM_ EXPONENT 0x00020000`	Use *printf* `%.nnle` style formatting (base-10 exponent e.g. 1E8 for 10^8)
`CTL_NUM_ FIXEDPOINT 0x00030000`	Use *printf* `%.nnlg` style formatting (shortest of above three)

Exactly one of `CTL_NUM_FLOAT`, `CTL_NUM_ FIXEDPOINT`, `CTL_NUM_EXPONENT and CTL_ NUM_FLOAT` should be specified.

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the number control value represents. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Number control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Format Descriptor Block

The *Format Descriptor block* expresses a preference as to how the number value is displayed as a text string. A client is not required to do so; in particular, it may not be practical for modifiable controls. The format descriptor is interpreted as the string of text to be displayed, after the %% value has been replaced with the current number value. For example, to display

$7.99c

use the format descriptor

$%%c

The *Descriptor Identifier* used to indicate the Format Descriptor Block is `0x66`.

DataSize	Contains
uint32	Size `nFmt` of *Format String* in bytes
`nFmt`	Format String in ASCII or Unicode as specified in message header.

The format string may or may not contain a zero terminator. If it does not, its length should be assumed to be `nFmt`

Minimum Value Descriptor Block

The *Minimum Value Descriptor block* specifies a minimum allowable value that the number value may take. Neither server nor client should permit the value to go below the minimum value. The *Descriptor Identifier* used to indicate the Minimum Value Descriptor Block is `0x6E`.

DataSize	Contains
uint32	*Minimum Value* size, value `0x04`
`0x04`	4-byte signed integer minimum value

Maximum Value Descriptor Block

The *Maximum Value Descriptor block* specifies a maximum allowable value that the number value may take. Neither server nor client should permit the value to go above the maximum value. The *Descriptor Identifier* used to indicate the Maximum Value Descriptor Block is `0x78`.

DataSize	Contains
uint32	*Maximum Value* size, value `0x04`
`0x04`	4-byte signed integer maximum value

Decimals Descriptor Block

The *Decimals Descriptor block* specifies the number of decimal places which should be displayed. This does not shift the decimal place of the underlying number value, but how it is displayed in terms of digits after any decimal point. The *Descriptor Identifier* used to indicate the Mantissa Descriptor Block is `0x64`.

DataSize	Contains
uint32	*Decimals* size, value `0x01`
`0x01`	1-byte integer decimals value 0 to 127

Mantissa Descriptor Block

The *Mantissa Descriptor block* specifies a power ten multiplier to use when displaying the number value. For example, if the integer value is 42 and the mantissa is 3, the displayed value is 42000. If the mantissa is –2, the same number value would be displayed as 0.42. The *Descriptor Identifier* used to indicate the Mantissa Descriptor Block is `0x6D`.

DataSize	Contains
uint32	*Mantissa* size, value `0x01`
`0x01`	1-byte signed integer -128 to 127

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an ASCII modifiable number control block with minimum value 0, 2 fixed point decimal places, -2 (hundredths) mantissa and value *4.04* named “Num” as follows:

Bytes in order of transmission (hex)	Meaning
`3D,00,00,00`	Size of fields to follow (61 bytes)
`4E,00`	Control type, value `0x004E`
`0D,00,13,00`	Specifies control flags `CTL_NUM_MODIFIABLE`, `CTL_NUM_MIN`, and `CTL_NUM_FIXEDPOINT`
`54,00,00,00`	Control ID `0x00000054`
`08,00,00,00`	Control value size = 4
`94,01,00,00`	Control value 404
`05,00,00,00`	5 descriptor identifiers follow, 1 byte each
`5A,6E,64,6D,63`	Descriptor identifiers: Name, Minimum, Decimals, Mantissa, Color
`03,00,00,00`	Size of name descriptor data in bytes
`4E,75,6D`	Text "Num" in ASCII
`04,00,00,00`	Size of minimum in bytes
`00,00,00,00`	Minimum value
`01,00,00,00`	Size of decimals in bytes
`02`	Decimals value
`01,00,00,00`	Size of mantissa in bytes
`FE`	Mantissa value (-2)
`04,00,00,00`	Size of color descriptor data in bytes
`FF,00,00,00`	Bright red latch requested

Matrix Control Block

The Matrix Control Block stores matrix of 4-byte signed integers, *i.e.* a table of number values. There are four basic types, based on what the X-axis (i.e. the rows of the matrix) represent:

- A *list* matrix type has no X-axis information associated with each row.
- A *XY* matrix type has an X-axis number value information associated with each row.
- A *labels* matrix type has a text label associated with each row.
- A *date-time* matrix type has a date-time value associated with each row.

As a minimum, a client must be able to display the data as a table of numbers, even if it is able to display it in other ways, too. Matrix controls are demanding of client UI capabilities so a server should expect nothing better than this, particularly on devices such as cellphones.

However, a good client would offer many possibilities which would be specified in the device-specific profiles. From FlexiPanel Protocol 3.0, the following styles are rendered by default where possible:

- A *table of numbers* for the *list* matrix type.
- A *points chart* for the *XY* matrix type.
- A *column chart* for the *labels* matrix type.
- A *line chart* for the *date-time* matrix type.

The Matrix Control Block was implemented in protocol version 2.2 and remains current. From version 3.0, a server may update a single row in a message. In no version may the client modify the data.

The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x004D`
`uint32`	Generic control flags plus *Matrix Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value size, value `mSiz`
`mSiz`	*Matrix Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Matrix Value

The matrix value size `mSiz` will be implied by the number of rows and columns and the type of data required to represent the X axis. The matrix value comprises of the following fields:

- The matrix of integers 'unwrapped' into a linear array. All the values in the first column come first, then the next column, etc. Each column of values is known as a Column Sub-Array. Each integer will be signed but may be 1- 2- or 4-byte depending on the matrix flags.

- If the matrix is of the *List* matrix type, no X-axis data is sent.

- If the matrix is of the *XY* matrix type, the X-axis values are sent as an array of signed integers which may be 1- 2- or 4-byte depending on the matrix flags.

- If the matrix is of the *Labels* matrix type, the X-axis text values are sent as a *Zero Interspersed String List* (see ***Error! Reference source not found.***, page **Error! Bookmark not defined.**). A client must infer the size of the list from the overall size of the Matrix Value, `mSiz`.

- If the matrix is of the *Date-Time* matrix type, the X-axis Date-Time values are sent as an array of 8-byte Date-Time values (see *Date-Time Control Block*, page 16).

- An four byte signed integer `NumValid` indicating the number of rows of the matrix which contain valid data and whether it must be interpreted as an offset circular array:

    - If `NumValid` is zero or positive, the first `NumValid` rows contain valid data and the rest are unspecified.

    - If `NumValid` is negative, all rows contain valid data. The row elements of the matrix have been offset and row `R` of the matrix is located at Column Sub-Array element offset `E`, where

      ```
      E = ( R – NumValid ) % NumRows
      ```

      and `NumRows` is the total number of rows in the matrix. `E` and `R` are zero-based; `%` is the modulus operator; `NumValid` may be no more negative than `–NumRows`. If XY style or Date-Time style, this offset applies to the row data, too. (It does not apply to row labels in the Labels style.)

Matrix Flags

The following control flags are specific to the matrix control:

Name / `dwFlags` constant	Meaning
`CTL_MTX_DATA_LIST 0x00000000*`	Matrix is List type (no X data)
`CTL_MTX_DATA_XY 0x00010000*`	Matrix is XY type (X values are Number values)
`CTL_MTX_DATA_LABELS 0x00020000*`	Matrix is Labels type (X values are text labels)
`CTL_MTX_DATA_TY 0x00030000*`	Matrix is Date-Time type (X values are Date-Time)
`CTL_MTX_X_4BYTE 0x00000000**`	(XY type only) X values are 4-byte signed integers
`CTL_MTX_X_2BYTE 0x00040000**`	(XY type only) X values are 2-byte signed integers
`CTL_MTX_X_1BYTE 0x00080000**`	(XY type only) X values are 1-byte signed integers
`CTL_MTX_Y_4BYTE 0x00000000`†	Y values are 4-byte signed integers
`CTL_MTX_Y_2BYTE 0x00100000`†	Y values are 2-byte signed integers
`CTL_MTX_Y_1BYTE 0x00200000`†	Y values are 1-byte signed integers
`CTL_NUM_X_FLOAT 0x00000000`††	(XY type only) Use *printf* `%.lf` style formatting (floating point) for X integer data
`CTL_NUM_X_FIXEDPOINT 0x00400000`††	(XY type only) Use *printf* `%.nnlf` style formatting (fixed point) for X number values
`CTL_NUM_X_EXPONENT 0x00800000`††	(XY type only) Use *printf* `%.nnle` style formatting (base-10 exponent e.g. 1E8 for 10^8) for X number values
`CTL_NUM_X_FIXEDPOINT 0x00C00000`††	(XY type only) Use *printf* `%.nnlg` style formatting (shortest of above three) for X number values
`CTL_NUM_Y_FLOAT 0x00000000`‡	Use *printf* `%.lf` style formatting (floating point) for Y number values
`CTL_NUM_Y_FIXEDPOINT 0x01000000`‡	Use *printf* `%.nnlf` style formatting (fixed point) for Y number values
`CTL_NUM_Y_EXPONENT 0x02000000`‡	Use *printf* `%.nnle` style formatting (base-10 exponent e.g. 1E8 for 10^8) for Y number values

Name / dwFlags constant	Meaning
CTL_NUM_Y_ FIXEDPOINT 0x03000000[‡]	Use *printf* %.nnlg style formatting (shortest of above three) for Y number values
CTL_MTX_ROW_ UPDATE 0x04000000	Server updates send a single row of data at a time (from version 3.0)

Exactly one of each group of flags marked *, **, [†], [††] and [‡] should be specified.

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the number control value represents. The *Descriptor Identifier* used to indicate the Name Descriptor Block is 0x5A.

DataSize	Contains
uint32	Size nNam of *Name* in bytes
nNam	Control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be nNam.

Row Count Descriptor Block

The *Row Count Descriptor block* specifies the number of rows of data for which memory should be allocated. Not all these rows need contain valid data. This Descriptor Block is required. The *Descriptor Identifier* used to indicate the Row Count Descriptor Block is 0x6E.

DataSize	Contains
uint32	*Maximum Value* size, value 0x04
0x04	4-byte integer NumRows number of rows

Column Count Descriptor Block

The *Column Count Descriptor block* specifies the number of columns of data for which memory should be allocated. This Descriptor Block is required. The *Descriptor Identifier* used to indicate the Row Count Descriptor Block is 0x4E.

DataSize	Contains
uint32	*Maximum Value* size, value 0x04
0x04	4-byte integer NumRows number of rows

X Format Descriptor Block

(XY type only.) The *X Format Descriptor block* expresses a preference as to how the X-axis number values are displayed as a text strings. The format descriptor is interpreted as the string of text to be displayed, after the %% value has been replaced with the current number value. For example, to display

$7.99c

use the format descriptor

$%%c

The *Descriptor Identifier* used to indicate the Format Descriptor Block is 0x66.

DataSize	Contains
uint32	Size nFmt of *Format String* in bytes
nFmt	Format String in ASCII or Unicode as specified in message header.

The format string may or may not contain a zero terminator. If it does not, its length should be assumed to be nFmt.

Y Format Descriptor Block

The *Y Format Descriptor block* expresses a preference as to how the matrix number values are displayed as a text strings. The format descriptor is interpreted as the string of text to be displayed, after the %% value has been replaced with the current number value. For example, to display

$7.99c

use the format descriptor

$%%c

The *Descriptor Identifier* used to indicate the Format Descriptor Block is 0x46.

DataSize	Contains
uint32	Size nFmt of *Format String* in bytes

`nFmt`	Format String in ASCII or Unicode as specified in message header.

The format string may or may not contain a zero terminator. If it does not, its length should be assumed to be `nFmt`.

X Axis Title Descriptor Block

The *X Axis Title Descriptor Block* defines the X Axis title text. The *Descriptor Identifier* used to indicate the X Axis Descriptor Block is `0x61`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	X Axis name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Y Axis Title Descriptor Block

The *Y Axis Title Descriptor Block* defines the Y Axis title text. The *Descriptor Identifier* used to indicate the Y Axis Descriptor Block is `0x41`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Y Axis name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

X Decimals Descriptor Block

(XY type only.) The *X Decimals Descriptor block* specifies the number of decimal places which should be displayed for row values. This does not shift the decimal place of the underlying number value, but how it is displayed in terms of digits after any decimal point. The *Descriptor Identifier* used to indicate the Mantissa Descriptor Block is `0x64`.

DataSize	Contains
uint32	*Decimals* size, value `0x01`
`0x01`	1-byte integer decimals value 0 to 127

X Mantissa Descriptor Block

The *X Mantissa Descriptor block* specifies a power ten multiplier to use when displaying the row value. For example, if the integer value is 42 and the mantissa is 3, the displayed value is 42000. If the mantissa is –2, the same number value would be displayed as 0.42. The *Descriptor Identifier* used to indicate the Mantissa Descriptor Block is `0x6D`.

DataSize	Contains
uint32	*Mantissa* size, value `0x01`
`0x01`	1-byte signed integer -128 to 127

Y Decimals Descriptor Block

The *Y Decimals Descriptor block* specifies the number of decimal places which should be displayed for the matrix values. This does not shift the decimal place of the underlying number value, but how it is displayed in terms of digits after any decimal point. The *Descriptor Identifier* used to indicate the Mantissa Descriptor Block is `0x44`.

DataSize	Contains
uint32	*Decimals* size, value `0x01`
`0x01`	1-byte integer decimals value 0 to 127

Y Mantissa Descriptor Block

The *Y Mantissa Descriptor block* specifies a power ten multiplier to use when displaying the matrix values. For example, if the integer value is 42 and the mantissa is 3, the displayed value is 42000. If the mantissa is –2, the same number value would be displayed as 0.42. The *Descriptor Identifier* used to indicate the Mantissa Descriptor Block is `0x4D`.

DataSize	Contains
uint32	*Mantissa* size, value `0x01`
`0x01`	1-byte signed integer -128 to 127

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example – List type

A server sends an list type ASCII matrix control block with 1-byte integer values, 3 rows (only 2 valid), 2 columns, default formatting options, X Axis title ‘Rows’, Y Axis title ‘Cols’, named “Chart” as follows:

Bytes in order of transmission (hex)	Meaning
`3D,00,00,00`	Size of fields to follow (61 bytes)
`4D,00`	Control type, value `0x004D`
`00,00,02,00`	Specifies control flags `CTL_MTX_DATA_LIST`, `CTL_MTX_Y_1BYTE`
`44,41,54,31`	Control ID `0x31544144`
`0A,00,00,00`	Control value size = 10

`01,02,CD,`
`04,03,CD,`
 	
`02,00,00,00`
	Control value
	(col 0)
	(col 1)
	(no row-specific data)
	(`NumValid`)

`03,00,00,00`	3 descriptor identifiers follow, 1 byte each
`5A,61,41`	Descriptor identifiers: Name, X Title, Y Title
`05,00,00,00`	Size of name descriptor data in bytes
`43,68,61,72,74`	Text “Chart” in ASCII
`04,00,00,00`	Size of X Title in bytes
`52,6F,77,73`	X Title “Rows” in ASCII
`04,00,00,00`	Size of Y Title in bytes
`43,6F,6C,73`	Y Title “Cols” in ASCII

In the above table, the `CD` values represent unused bytes and could be any value.

Worked example – XY / Date-Time type

A server sends an XY type ASCII matrix control block with 1-byte integer values, 3 rows (all valid, offset to start at row 2), 1 column, 2-byte row values, default formatting options, no axis titles or name specified, as follows. See Number Control Block for worked example of formatting. The Date-Time type follows the XY type except that the row values are each 8-byte Date-Time values.

Bytes in order of transmission (hex)	Meaning
`15,00,00,00`	Size of fields to follow (21 bytes)
`4D,00`	Control type, value `0x004D`
`00,00,0D,01`	Specifies control flags `CTL_MTX_DATA_XY`, `CTL_MTX_Y_1BYTE`, `CTL_MTX_X_2BYTE`
`44,41,54,32`	Control ID `0x32544144`
`0D,00,00,00`	Control value size = 13

`01,02,03,`
`01,00,02,00,03,`
`00,`
`FE,FF,FF,FF`
	Control value
	(col 0)
	(row / X values, 2 bytes each)
	(`NumValid` = -2)

`00,00,00,00`	0 descriptor identifiers follow

Worked example – Labels type

A server sends an Labels type ASCII matrix control block with 1-byte integer values, 3 rows (all valid, offset to start at row 2), 1 column, row labels ‘One’, Two, ‘Three’, default formatting options, no axis titles or name specified, as follows. (See Number Control Block for worked example of formatting.)

Bytes in order of transmission (hex)	Meaning
`2B,00,00,00`	Size of fields to follow (43 bytes)
`4D,00`	Control type, value `0x004D`
`00,00,02,00`	Specifies control flags `CTL_MTX_DATA_LABELS`, `CTL_MTX_Y_1BYTE`
`44,41,54,32`	Control ID `0x32544144`
`15,00,00,00`	Control value size = 21

`01,02,03,`
`4F,6E,65,00,`
`54,77,6F,00`
`54,68,72,65,65,`
`00,`
`FE,FF,FF,FF`
	Control value
	(col 0)
	(row labels as Zero Interspersed String List, 2 bytes each)
	
	(`NumValid` = -2)

`00,00,00,00`	0 descriptor identifiers follow

Password Control Block

The Password Control Block stores a binary open/closed value which the client can only put into the open state by specifying the correct password. If so configured, the password may be modifiable once open.

Passwords provide a security function whose primary use is to prevent unwanted access to a user interface. A change of state of a password control would usually result in the server sending a New Control Panel From Server message; however, it entirely up to the server what it does.

- The client cannot directly change the state of the password control. It can only do so by transmitting a password and the server verifying it to be correct. The server never transmits passwords.
- Some client devices (cellphones) can only enter the digits 0-9 in passwords.
- In ASCII based systems, the maximum password length is 33 characters. In Unicode systems, it is 16 characters. (Zero terminator not included.)

The Password Control Block was implemented in protocol version 2.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0050`
`uint32`	Generic control flags plus *Password Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, `0x00000001`
`0x01`	*Control Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Password Value

The password control value is a single byte. `0x00` represents closed and `0xFF` represents open.

Password Flags

The following control flags are specific to the password control:

Name / `dwFlags` constant	Meaning
`CTL_PWD_ MODIFIABLE 0x00010000`	Client is permitted to change the password.
`CTL_PWD_ AUTOCLOSE 0x00020000`	Server always returns the Password control to the closed state when a client disconnects

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the password relates to. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Section control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an open, automatically closing, ASCII "Pwd" password control block as follows:

Bytes in order of transmission (hex)	Meaning
`1B,00,00,00`	Size of following fields (27 bytes)
`50,00`	Control type, value `0x0050`
`00,00,02,00`	Specifies control flag `CTL_PWD_AUTOCLOSE`
`50,77,64,00`	Control ID `0x00647750`
`01,00,00,00`	Control value size = 1
`FF`	Control value (open)
`01,00,00,00`	1 descriptor identifier follows, 1 byte long
`5A`	Descriptor identifier: Name
`03,00,00,00`	Size of name descriptor data in bytes
`50,77,64`	Text "Pwd" in ASCII

List Control Block

The List Control Block stores a 4-byte signed integer, *i.e.* in the range –2,147,483,648 to +2,147,483,647. It represents the current selected item in a list, or -1 for no item selected.

A client may not modify the list items; it can only choose which is selected. A server may modify the list items, but only by sending a *New Control Panel From Server* message.

The List Control Block was implemented in protocol version 2.0 and remains current. Each *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x004C`
`uint32`	Generic control flags plus *List Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `0x00000004`
`0x04`	*List Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

List Value

The List value is a 4-byte integer, transmitted least significant byte first (as are all multi-byte integers in FlexiPanel). It represents the current selected item in a list (zero based, so the first item is 0), or -1 for no item selected.

List Flags

The following control flags are specific to the number control:

Name / `dwFlags` constant	Meaning
`CTL_ LST_ NONULLSELECT 0x00010000`	No item selected is an allowable state.

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the number control value represents. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Number control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Item Count Descriptor Block

The *Item Count Descriptor block* specifies how many items are in the list. The *Descriptor Identifier* used to indicate the Format Descriptor Block is `0x6E`.

DataSize	Contains
uint32	*Item Count* size, value `0x04`
`0x04`	4-byte integer item count

Items Text Descriptor Block

The *Items Text Descriptor block* specifies the text for each item in the list as a Zero Interspersed String List (see **Error! Reference source not found.**, page **Error! Bookmark not defined.**). The *Descriptor Identifier* used to indicate the Minimum Value Descriptor Block is `0x49`.

DataSize	Contains
uint32	*Items Text* string list size, value `sizZISL`
`sizZISL`	*Items Text* Zero Interspersed String List, ASCII or Unicode as previously specified

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an ASCII list control block with item values “One”, “Two”, Three”, the last is selected, named “List” as follows:

Bytes in order of transmission (hex)	Meaning
`3B,00,00,00`	Size of fields to follow (59 bytes)
`4E,00`	Control type, value `0x004E`
`00,00,00,00`	No control flags specified
`4C,69,73,74`	Control ID `0x7473694C`
`04,00,00,00`	Control value size = 4
`02,00,00,00`	Control value = 2
`03,00,00,00`	3 descriptor identifiers follow, 1 byte each
`5A,6E,49`	Descriptor identifiers: Name, Item Count, Items Text
`04,00,00,00`	Size of name descriptor data in bytes
`4C,69,73,74`	Text “List” in ASCII
`04,00,00,00`	Size of Item Count in bytes
`03,00,00,00`	Item Count
`0E,00,00,00`	Size of Items Text (14 bytes)
`4F,6E,65,00,` `54,77,6F,00` `54,68,72,65,65,` `00`	List Items Text (“One”) (“Two”) (“Three”)

Message Control Block

The Message Control Block stores a text message which can be displayed at request of the server. No response is necessarily expected from the client, but the user should not be able to continue to operate the user interface until the message is dismissed. From version 3.0, the client may offer the Windows standard Message Box response buttons and will inform the server which was pressed.

The message control information is sent in the *New Control Panel From Server* and *Control Update From Server* message. It is not displayed until *Control Properties Update From Server* message is sent with the `CTL_INVISIBLE` flag not set.

The Message Control Block was implemented in protocol version 2.0 and remains current. Each *message control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0058`
`uint32`	Generic control flags plus *Matrix Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, `nBytes`

nBytes	*Control Value*
uint32	Size nDsc of *Control Descriptor Header* field, in bytes
nDsc	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then nDsc *Descriptor Blocks*	

Message Value

The control value is zero terminated text to appear when the message is shown. The length in characters must be less than or equal to the maximum number of characters as specified in the Characters Descriptor.

Message Flags

The following control flags are specific to the message control:

Name / dwFlags constant	Meaning
CTL_MSG_ICON_ NONE 0x00010000*	Display no icon.
CTL_MSG_ICON_ STOP 0x00020000*	Display Stop icon.
CTL_MSG_ICON_ EXCLAMATION 0x00030000*	Display Exclamation icon.
CTL_MSG_ICON_ QUESTION 0x00040000*	Display Question Mark icon.
CTL_MSG_ICON_ INFORMATION 0x00050000*	Display Information icon.
CTL_RESP_NONE 0x00000000†	Do not generate a client reply (specify this for pre-version 3.0 compatibility.)
CTL_RESP_OK 0x00100000†	Display OK response button.
CTL_RESP_ OKCANCEL 0x00200000†	Display OK, Cancel response buttons.
CTL_RESP_ RETRYCANCEL 0x00300000†	Display Retry, Cancel response buttons.
CTL_RESP_YESNO 0x00400000†	Display Yes, No response buttons.
CTL_RESP_ YESNOCANCEL 0x00500000†	Display Yes, No, Cancel response buttons.
CTL_RESP_ABORT RETRYIGNORE 0x00600000†	Display Abort, Retry, Ignore response buttons.

Exactly one of each group of flags marked * and † should be specified.

Characters Descriptor Block

The *Characters Descriptor Block* specifies the maximum number of characters of any message used by that control, including zero terminator. The value is characters, i.e. if the server is Unicode, twice the number of bytes will be required to store messages than characters specified in the Character Descriptor Block. The *Descriptor Identifier* used to indicate the Characters Descriptor Block is 0x6D.

DataSize	Contains
uint32	*Characters* size, value 0x00000004
0x04	Number of characters (not bytes), including zero terminator.

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the message relates to. It may be used in the title bar of the message box. The *Descriptor Identifier* used to indicate the Name Descriptor Block is 0x5A.

DataSize	Contains
uint32	Size nNam of *Name* in bytes
nNam	Section control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be nNam.

Color Descriptor Block

A client is unlikely to do anything with a *Color Descriptor Block* specified for a message control, but there's no harm in sending one.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends a Unicode "Busy!" message control block as follows. The text buffer is large enough for messages of up to 8 characters plus zero terminator:

Bytes in order of transmission (hex)	Meaning
`31,00,00,00`	Size of following fields (49 bytes)
`58,00`	Control type, value `0x0058`
`00,00,05,00`	Specifies control flags `CTL_RESP_NONE` and `CTL_MSG_ICON_ INFORMATION`
`4D,73,67,00`	Control ID `0x0067734D`
`12,00,00,00`	Control value size `nBytes` = 18 bytes (nine Unicode characters including zero terminator)
`42,00,75,00, 73,00,79,00, 21,00,00,00, CD,CD,CD,CD, CD,CD`	Control value "Busy!" in a buffer large enough for 8 characters. (`CD` values could be anything.)
`02,00,00,00`	2 descriptor identifiers follow of 1 byte each
`5A,6D`	Descriptor identifiers: Name, Characters
`03,00,00,00`	Size of name descriptor data in bytes
`4D,73,67`	Name "Msg" in ASCII
`04,00,00,00`	Size of Characters in bytes
`09,00,00,00`	Characters, = `nBytes` / 2

In the above table, the `CD` values represent unused bytes and could be any value.

Blob Control Block

The Blob Control stores a quantity of binary data. It is intended primarily for the customized transfer of data between specific servers. It may also be used to pass a URL (i.e. web page address) to a client. The client should then, if able, provide a button which launches a web browser to retrieve that web page.

A typical use of the blob control is to provide a link to a corporate web site.

A client is not required to support the Blob control but must continue to function without error if it receives one.

The Blob Control Block was implemented in protocol version 2.0 and remains current. Its use for custom data transfer is not defined in any way.

Each *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x004F`
`uint32`	Generic control flags plus *Blob Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size `sBlob`, equal to the number of binary data *bytes*.
`sBlob`	*Blob Value*
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Blob Value

If the `CTL_BLOB_HINT_URL` blob flag is specified, the value is a URL. This may be ASCII or Unicode, according to the *Control Flags.* `sBlob` bytes are always transmitted, regardless of current size of the blob.

If CTL_BLOB_TEXT is defined, the client may need to convert the text to its native format (ASCII / Unicode) before processing.

If the CTL_BLOB_HINT_NONE blob flag is specified, the data content is unrestricted and may be used for custom server / client combinations.

Blob Flags

The following control flags are specific to the text control:

Name / dwFlags constant	Meaning
CTL_BLOB_TEXT 0x00010000	Blob is composed of text characters in the native text format (ASCII / Unicode) of the server.
CTL_BLOB_ HINT_NONE 0x00000000	Data content is unrestricted
CTL_BLOB_ HINT_URL 0x00100000	Data is a zero-terminated URL. (CTL_BLOB_TEXT should also be defined.)

Name Descriptor Block

The *Name Descriptor Block* defines the title text which describes what the blob data represents. If a URL launch button is provided, this text will likely appear on the button. The *Descriptor Identifier* used to indicate the Name Descriptor Block is 0x5A.

DataSize	Contains
uint32	Size nNam of *Name* in bytes
nNam	Text control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be nNam.

Length Descriptor Block

The *Length Descriptor Block* specifies the maximum number of bytes which may be required to store Blob data. It is required. The *Descriptor Identifier* used to indicate the Length Descriptor Block is 0x6D.

DataSize	Contains
uint32	*Length* size, value 0x00000004
0x04	Maximum number of bytes

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is 0x63.

DataSize	Contains
uint32	*Color* size, value 0x00000004
0x04	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends an ASCII URL "www.flexipanel.com" blob control block as follows:

Bytes in order of transmission (hex)	Meaning
2F,00,00,00	Size of fields to follow (47 bytes)
4F,00	Control type, value 0x004F
00,00,11,00	Specifies control flags CTL_BLOB_HINT_URL and CTL_BLOB_TEXT
42,6C,6F,62	Control ID 0x626F6C42
13,00,00,00	Control value size = 19
77,77,77,23,66, 6C,65,78,69,70, 61,6E,65,6C,23, 62,6F,6D,00	Control value "www.flexipanel.com" in zero terminated ASCII
02,00,00,00	1 descriptor identifier follows, 1 byte long
5A	Descriptor identifier: Name
08,00,00,00	Size of name descriptor data in bytes
42,6C,6F,62,00	Text "Blob" in ASCII

Flies Control Block

The Flies Control allows files to be transferred between server and client. It is intended primarily for the customized transfer of data between specific servers. It may also be used to pass web pages and files (e.g. images) to a client. The client should then, if able, provide a button which launches a web browser to display the files as a web page.

The Files control differs from the Blob control in that the data is transferred on request only. The *Control Update From Client* and *Control Update From Server* messages are used to request the files. The files are actually transferred in the *Files From Server* and *Files From Client* messages.

If a server provides a files control it does not need to be able to accept files. If it can't, it simply never requests them. While the mechanism exists to send files from client to server within in the protocol, this has not actually been implemented on any clients or servers because a specific need has not yet arisen.

A client is not required to support the Files control but must continue to function without error if it receives one.

A typical use of the files control is to serve up web pages locally where an internet cannot be guaranteed. For example, a product's entire instruction manual may be pre-loaded with the product and uploaded on demand.

The Files Control Block was implemented in protocol version 2.0 and remains current. Each *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0046`
`uint32`	Generic control flags plus *Files Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size = 0
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Files Value

The files values has zero size since the files are transferred on request only.

The client uses the *Control Update From Client* to indicate that it wishes the files to be transferred.

The server uses the *Control Update From Server* to indicate that it wishes the files to be transferred. The value block is empty.

Files Flags

The following control flags are specific to the text control:

Name / `dwFlags` constant	Meaning
`CTL_FILE_HINT_NONE` `0x00000000`	Files are completely undefined for use with custom server / client combinations
`CTL_FILE_HINT_HTML` `0x00100000`	Files are acceptable to a browser.

If `CTL_FILE_HINT_HTML` is specified, the files should all be stored in the same directory and then a web browser should be directed to the first file. In this way, the server can serve up web pages locally without the need for an internet connection. The first file serves as the home page. The other files may be linked web pages or other supporting files such as images.

Name Descriptor Block

The *Name Descriptor Block* defines the title text which describes what the files data represents. If a browser launch button is provided, this text will likely appear on the button. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Text control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Color Descriptor Block

The *Color Descriptor block* requests coloring for the control. The server is not required to request a color. Not all clients can provide colors. The *Descriptor Identifier* used to indicate the Color Descriptor Block is `0x63`.

DataSize	Contains
uint32	*Color* size, value `0x00000004`
`0x04`	Microsoft RGB value 0x00BBGGRR.

Worked example

A server sends a files control with the ASCII name "Instructions" as follows:

Bytes in order of transmission (hex)	Meaning
`23,00,00,00`	Size of fields to follow (35 bytes)
`46,00`	Control type, value `0x0046`
`00,00,10,00`	Specifies control flag `CTL_FILE_HINT_HTML`
`46,69,6C,65`	Control ID `0x656C6946`
`00,00,00,00`	Control value size = 0
`02,00,00,00`	1 descriptor identifier follows, 1 byte long
`5A`	Descriptor identifier: Name
`0C,00,00,00`	Size of name descriptor data in bytes
`49,6E,73,74,75, 63,74,69,6F,6E, 73,00`	Text "Instructions" in ASCII

Image Control Block

The Image Control Block stores a color image. It could be used to represent a company logo and, on some clients, may be clickable. A client's ability to render an image may be limited.

The Image 3.0 and remains current. The *control block* consists of:

DataSize	Contains
`uint32`	Total size of all fields in the *control block* except this field but including all descriptor fields, in bytes
`uint16`	Control Type, value `0x0049`
`uint32`	Generic control flags plus *Latch Flags* described below
`uint32`	Control Unique ID
`uint32`	Control Value Size, value `ImgSz`
`ImgSz`	Image, BMP or GIF format
`uint32`	Size `nDsc` of *Control Descriptor Header* field, in bytes
`nDsc`	*Control Descriptor Header*, one byte per *Descriptor Block* to follow. Each byte is a *Descriptor Identifier* that indicates the contents of the *Descriptor Block. Descriptor Blocks* follow in the same order as their identifiers
then `nDsc` *Descriptor Blocks*	

Image Value

The image value format depends on the image format flag; currently only `CTL_IMG_FMT_BMP` and `CTL_IMG_FMT_GIF` are defined. The `CTL_IMG_FMT_BMP` format is the standard Microsoft BMP file format. The `CTL_IMG_FMT_GIF` format is the standard Compuserve GIF file format.

`CTL_IMG_FMT_BMP` images are not compressed but are easiest for server or client to modify.

Image Flags

The following control flags are specific to the image control:

Name / `dwFlags` constant	Meaning
`CTL_IMG_FMT_BMP` `0x00100000`	Data is in uncompressed BMP format
`CTL_IMG_FMT_GIF` `0x00200000`	Data is in compressed GIF format
`CTL_IMG_` `CLICK_MSG` `0x00010000`	A client update should be generated if the user clicks on the control; the image cannot be modifiable
`CTL_IMG_` `MODIFIABLE` `0x00020000`	The control is modifiable by the client

Name Descriptor Block

The *Name Descriptor Block* defines the title text that describes what the image represents. The *Descriptor Identifier* used to indicate the Name Descriptor Block is `0x5A`.

DataSize	Contains
uint32	Size `nNam` of *Name* in bytes
`nNam`	Image control name in ASCII or Unicode as specified in message header.

The name may or may not contain a zero terminator. If it does not, its length should be assumed to be `nNam`.

Color Descriptor Block

Any color descriptor block is ignored.

Worked example

A server sends a `0x1000` byte GIF pixel image with no name as follows:

Bytes in order of transmission (hex)	Meaning
`12,10,00,00`	Size of fields to follow (`0x1012` bytes)
`42,00`	Control type, value `0x0049`
`00,00,21,00`	Specifies control flags `CTL_IMG_FMT_GIF` and `CTL_IMG_CLICK_MSG.`
`49,4D,47,00`	Control ID `0x00474D49`
`00,10,00,00`	Control value size `0x1000`
`(0x1000 bytes)`	GIF Image
`00,00,00,00`	0 descriptor identifiers follow

Worked example – entire message

The following New Control Panel From Server message sends a latch control (ASCII form) and a number control.

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`06,00`	New Control Panel message ID
`00,00`	Flags – none
`02,00,00,00`	2 control blocks follow
`1E,00,00,00`	Size of latch control block
`42,00`	Control type, value `0x0042`
`00,00,00,00`	Control flags – none
`54,53,52,00`	Control ID `0x00525354`
`01,00,00,00`	Control value size = 1
`00`	Control value (off)
`03,00,00,00`	1 descriptor identifier follows
5A	Descriptor identifiers: Name
`0E,00,00,00`	Size of name descriptor data in bytes
`4C,61,74,63, 68,00`	Text “Latch” in ASCII Radio 1
`3C,00,00,00`	Size of number control block
`4E,00`	Control type, value `0x004E`
`0D,00,13,00`	Specifies control flags `CTL_NUM_MODIFIABLE`, `CTL_NUM_MIN`, and `CTL_NUM_FIXEDPOINT`
`54,00,00,00`	Control ID `0x00000054`
`08,00,00,00`	Control value size = 4
`94,01,00,00`	Control value 404
`05,00,00,00`	5 descriptor identifiers follow, 1 byte each
`5A,6E,64,6D, 63`	Descriptor identifiers: Name, Minimum, Decimals, Mantissa, Color
`03,00,00,00`	Size of name descriptor data in bytes
`4E,75,6D`	Text “Num” in ASCII
`04,00,00,00`	Size of minimum in bytes
`00,00,00,00`	Minimum value
`01,00,00,00`	Size of decimals in bytes
`02`	Decimals value
`01,00,00,00`	Size of mantissa in bytes
FE	Mantissa value (-2)
`04,00,00,00`	Size of color descriptor data in bytes

FF,00,00,00	Bright red latch requested

Control Update From Client

The Control Update From Client informs the server when the client is attempting to modify a control's value.

The Control Update From Client message is sent whenever the user modifies a control. Non-modifiable controls such as the matrix control do not send this message. To be hack-proof, the server should also verify that the client is permitted to modify a value before processing this message. This means verifying that:

- The control ID is valid.
- The new data is valid.
- The control is modifiable.
- The control is visible. (This is important, since a client may attempt to modify a control which is no longer being displayed is protected by a locked password.)

The Control Update From Client message was implemented in protocol version 2.0 and remains current.

The message starts with a header with message constant `0x0007`. The body that follows will vary depending on the controls being modified. The general format is:

DataSize	Contains
uint32	number `nCtl` of *Client Update Blocks* to follow
then `nCtl` *Client Update Blocks*	

Each Client Update Block contains new data for an updated control. Only those controls which have been modified need to be transmitted. The general format is:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `sCUpdate` of the data to follow
`sCUpdate`	New control data

The control ID must be used to identify the type of control and therefore the control data format. The specific format for each control will now be described.

Button Client Update Block

The Button Client Update Block indicates that the button has been pressed. It was implemented in protocol version 2.0 and remains current.

Each *button client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000001`
`0x01`	Control data, value ignored

Sending the message indicates that the button has been pressed. The control data value is ignored.

Text Client Update Block

The Text Client Update Block was implemented in protocol version 2.0 and remains current. Only modifiable text controls may send client update messages.

Each *text client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `szLen` of the data to follow (bytes)
`szLen`	New text data

The new text data should be ASCII or Unicode according to the text format specified by the server. The data sent must not be longer than the length specified in the original Text Control Block. If the text string is shorter than the maximum length, a zero terminator must also be included.

Latch Client Update Block

The Latch Client Update Block indicates that the latch control has been modified. It was implemented in protocol version 2.0 and remains current. If the control is part of a radio group, it is the responsibility of the client to send correct client update blocks for all latch controls in the

Each *latch client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value

	`0x00000001`
`0x01`	New control data, value `0xFF` for on and `0x00` for off.

Section Client Update Block

The Section Client Update Block indicates that the section control has been opened or closed. It was implemented in protocol version 2.0 and remains current.

A change of state of a section control would usually result in the server sending a New Control Panel From Server message; however, it entirely up to the server which controls are displayed in each state.

Each *section client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000001`
`0x01`	New control data, value `0xFF` for open and `0x00` for closed.

Date-Time Client Update Block

The Date-Time Client Update Block indicates that a date-time control has been modified. It was implemented in protocol version 2.0 and remains current.

It is important that the server ignore those fields which are specified as non-modifiable, since the client's Date-Time controls may not be able to constrain the fields which are modified.

Each *date-time client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000008`
`0x08`	New control data, as detailed below.

The 8-byte Date-Time value consists of the following fields:

Datatype	Contains	Range
byte	Second	0 – 59
byte	Minute	0 – 59
byte	Hour	0 – 23
byte	Date	1 – 31
Byte	Day of week	0 – 6 Sunday to Saturday respectively 7 = Unknown
byte	Month	1 – 12
uint16	Year	0 – 65535

The Date-Time control is a little unusual in that both the server and client may wish to update the control at once. The situation arises because a real-time clock date-time control will be updated by the server every second. At the same time, the client may wish to set the clock time. Client software needs to ignore server updates while the client is in the process of modifying the value: real-time clock date-times controls should be tested when designing a FlexiPanel client.

Number Client Update Block

The Number Client Update Block indicates that a number control has been modified. It should only be sent for modifiable number controls. It was implemented in protocol version 2.0 and remains current.

A mantissa feature provides decimal shifting function for data display on the client. Client Update Block data, however, should contain the underlying 4-byte signed integer value. If maximum or minimum values have been specified, the server must verify that the new value is within range.

Each *number client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000004`
`0x04`	New control data (4-byte signed integer).

Matrix Client Update Block

The Matrix Control is not modifiable by the client and no matrix client update message should be sent.

Password Client Update Block

The Password Client Update Block transmits a password that the user enters, and, optionally a new password. It was implemented in protocol version 2.0 and remains current.

The server verifies the password and takes the following actions:

- If the password is correct, the state of the password control is set to unlocked.
- If the password is correct and the control is modifiable and the new password is not a zero-length string, the password is changed to the new password.
- If the password is incorrect or a zero-length string, the state of the password control is set to locked.

Passwords provide a security function whose primary use is to prevent unwanted access to a user interface. A change of state of a password control would usually result in the server sending a New Control Panel From Server message; however, it entirely up to the server what it does.

Each *password client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000044`
`0x44`	Password entered by user, or zero-terminated string to lock control.
`0x44`	New password entered by user, or zero-terminated string if no change.

Note:

- The client cannot change the state of the password control directly. It can only do so by transmitting a password and the server verifying it to be correct. The server never transmits passwords.
- The password is only changed if the correct old password is sent in the same message and if the `CTL_PWD_MODIFIABLE` flag is set. A password control may have non-modifiable 'master' passwords, but that is up to the server.
- Some client devices (cellphones) can only enter the digits 0-9 in passwords.
- In ASCII based systems, the maximum password length is 33 characters. In Unicode systems, it is 16 characters. (Zero terminator not included.)

List Client Update Block

The List Client Update Block indicates that a list control has been modified. It was implemented in protocol version 2.0 and remains current.

Each *list client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000004`
`0x04`	New control data (4-byte signed integer).

Message Client Update Block

From version 3.0, a message control may send a response indicating which button was pressed. It must not be sent to a server with version number below 3.0 or if the `CTL_RESP_NONE` flag was specified.

Each *message client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000001`
`0x01`	Message response value.

The message value is a single byte, representing the one-based index number of the button which was pressed. For example, if the message box flag `CTL_RESP_YESNOCANCEL`, the index values are 1 for Yes, 2 for No and 3 for Cancel. The response may be 0xFF, meaning that the client was not able to obtain an answer (usually because the client does not support message responses).

Blob Client Update Block

If the `CTL_BLOB_HINT_URL` flag is specified, the client is not expected to modify the Blob value and this message should never be sent; it is intended for custom server / client combinations and its usage is undefined. The Blob Client Update Block was implemented in protocol version 2.0 and remains current.

Each *blob client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `sLen` of the data to follow (bytes)
`sLen`	New blob data

The new blob data must not be longer than the length specified in the original Blob Control Block.

Flies Client Update Block

The Flies Client Update block is transmitted to request that the files are sent from the server. Files themselves are sent in a separate Files From Server message. The block contains no updated control data. The Files Client Update Block was implemented in protocol version 2.0 and remains current.

Each *files client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000000`

Worked example

The following Control Update From Client message sends updated values for a latch control and a number control.

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,01`	Serial number (Pocket PC in this example)
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`07,00`	Control Update From Client message ID
`00,00`	Flags – none
`02,00,00,00`	2 control blocks follow
`54,53,52,00`	Latch control ID
`01,00,00,00`	Control value size = 1
`FF`	Control value (on)
`54,00,00,00`	Number Control ID
`08,00,00,00`	Control value size = 4
`96,01,00,00`	Control value 406

Image Client Update Block

The Image Client Update Block comes in two forms. Either it informs the server that the user has clicked on the image (if the `CTL_IMG_CLICK_MSG` flag was set) or the images has been modified and the new value is being sent (if the `CTL_IMG_MODIFIABLE` flag was set). It was implemented in protocol version 3.0 and remains current.

If the `CTL_IMG_CLICK_MSG` flag was set, the *image client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000000`

If the `CTL_IMG_MODIFIABLE` flag was set, the *image client update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Control Value Size, value `ImgSz = xSize` x `ySize x 3`
`ImgSz`	`xSize` x `ySize` RGB triplets

Control Update From Server

The Control Update From Server informs the client when the server is attempting to modify a control's value.

The Control Update From Server message is sent whenever the server modifies a control. Controls specified as non-modifiable may be modified by the server at any time; the flag pertains to whether the client is allowed to modify the value.

The Control Update From Server message was implemented in protocol version 2.0 and remains current.

The message starts with a header with message constant `0x0009`. The body that follows will vary depending on the controls being updated. The general format is:

DataSize	Contains
uint32	number `nCtl` of *Server Update Blocks* to follow
then `nCtl` *Server Update Blocks*	

Each Server Update Block contains new data for an updated control. Only those controls which have been modified need to be transmitted. The general format is:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `sSUpdate` of the data to follow
`sSUpdate`	New control data

The control ID must be used to identify the type of control and therefore the control data format. The specific format for each control will now be described.

Button Server Update Block

It would not make sense for the server to tell the client that the button had been pressed and so a button server update block should never be sent

Text Server Update Block

The Text Server Update Block was implemented in protocol version 2.0 and remains current.

Each *text server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `szLen` of the data to follow (bytes)
`szLen`	New text data

The new text data should be ASCII or Unicode according to the text format specified in the original New Control Panel From Server message. The data sent must not be longer than the length specified in the original Text Control Block. If the text string is shorter than the maximum length, a zero terminator must also be included.

Latch Server Update Block

The Latch Server Update Block changes the state of a latch control. It was implemented in protocol version 2.0 and remains current.

Each *latch server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000001`
`0x01`	New control data, value `0xFF` for on and `0x00` for off.

If the latch was part of a radio group, the server must correctly set the state of all the controls in the radio group.

Section Server Update Block

The Section Server Update Block changes the state of a section control. It was implemented in protocol version 2.0 and remains current.

A change of state of a section control would usually result in the server sending a New Control Panel From Server message; however, it entirely up to the server which controls are displayed in each state.

Each *section server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000001`
`0x01`	New control data, value `0xFF` for open and `0x00` for closed.

Date-Time Server Update Block

The Date-Time Server Update Block indicates that a date-time control has been modified. It was implemented in protocol version 2.0 and remains current.

If a user is in the process of updating the date-time control, the client may ignore this message and give priority to the value being entered by the user.

Each *date-time server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000008`
`0x08`	New control data, as detailed below.

The 8-byte Date-Time value consists of the following fields:

Datatype	Contains	Range
byte	Second	0 – 59
byte	Minute	0 – 59
byte	Hour	0 – 23
byte	Date	1 – 31
Byte	Day of week	0 – 6 Sunday to Saturday respectively 7 = Unknown
byte	Month	1 – 12
uint16	Year	0 – 65535

Number Server Update Block

The Number Server Update Block indicates that a number control has been modified. It was implemented in protocol version 2.0 and remains current.

A mantissa feature provides decimal shifting function for data display on the client. Server Update Block data, however, should contain the underlying 4-byte signed integer value.

Each *number server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000004`
`0x04`	New control data (4-byte signed integer).

Matrix Server Update Block

The Matrix Server Update Block updates a entire data matrix. It was implemented in protocol version 2.0 and remains current.

Matrix Server Update Block

The *matrix server update block* updates all the data in the matrix and consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `nDat` of the matrix value to follow
`nDat`	Entire matrix value

The `nDat` is defined by the number of rows and columns and the type of data required to represent the X axis. The entire matrix value comprises of the following fields:

- The matrix of integers ‘unwrapped’ into a linear array. All the values in the first column come first, then the next column, etc. Each column of values is known as a Column Sub-Array. Each integer will be signed but may be 1- 2- or 4-byte depending on the matrix flags.

- If the matrix is of the *List* or *Labels* matrix type, no X-axis data is sent.

- If the matrix is of the *XY* matrix type, the X-axis values are sent as an array of signed integers which may be 1- 2- or 4-byte depending on the matrix flags.

- If the matrix is of the *Date-Time* matrix type, the X-axis Date-Time values are sent as an array of 8-byte Date-Time values (see *Date-Time Control Block*, page 16).

- An four byte signed integer `NumValid` indicating the number of rows of the matrix which contain valid data and whether it must be interpreted as an offset circular array:

    - If `NumValid` is zero or positive, the first `NumValid` rows contain valid data and the rest are unspecified.

    - If `NumValid` is negative, all rows contain valid data. The row elements of the matrix have been offset and row `R` of the matrix is located at Column Sub-Array element offset `E`, where

```
E = ( R – NumValid ) % NumRows
```

and `NumRows` is the total number of rows in the matrix. `E` and `R` are zero-based; `%` is the modulus operator; `NumValid` may be no more negative than `–NumRows`. If XY style of Date-Time style, this offset applies to the row data, too. (It does not apply to row labels in the Labels style.)

Password Server Update Block

The Password Server Update Block indicates that a password control has changed state. This may be because the server chose to, or in response to a correct or incorrect password being sent by the client. It was implemented in protocol version 2.0 and remains current.

Each *password server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000001`
`0x01`	New control state, being `0x00` for locked and `0xFF` for unlocked.

List Server Update Block

The List Server Update Block indicates that a list control has been modified. It was implemented in protocol version 2.0 and remains current.

Each *list server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size of the data to follow, value `0x00000004`
`0x04`	New control data (4-byte signed integer).

Message Server Update Block

A server uses the server update block message to change the message box's text. The message is shown, and the icons and buttons modified, using the Control Properties Update From Server page 53.

The Message Server Update Block was implemented in protocol version 2.0 and remains current.

Each *message server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `szLen` of the new text to follow (bytes)
`szLen`	New text

The new text message should be ASCII or Unicode according to the format specified in the original New Control Panel From Server message. The data sent must not be longer than the length specified in the original Matrix Control Block. If the text string is shorter than the maximum length, a zero terminator must also be included.

Blob Server Update Block

The Blob Server Update Block was implemented in protocol version 2.0 and remains current.

Each *blob server update block* consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `sLen` of the data to follow (bytes)
`sLen`	New blob data

The new blob data must not be longer than the length specified in the original Blob Control Block. If the `CTL_BLOB_HINT_URL` flag is specified, the new blob data is a new URL.

Flies Server Update Block

The Flies Server Update block is transmitted to request that the files are sent from the client. Files themselves are sent in a separate Files From Client message. The block contains no updated control data. The Files Server Update Block was implemented in protocol version 2.0 and remains current.

Each *files server update block* consists of:

DataSize	Contains
uint32	Control ID
uint32	Size of the data to follow, value 0x00000000

Image Server Update Block

The Image Server Update Block indicates the image has been modified and the new value is being sent. It was implemented in protocol version 3.0 and remains current.

Each *image server update block* consists of:

DataSize	Contains
uint32	Control ID
uint32	Control Value Size, value ImgSz = xSize x ySize x 3
ImgSz	xSize x ySize RGB triplets

Worked example

The following Control Update From Server message sends updated values for a latch control and a number control.

Bytes in order of transmission (hex)	Meaning
48,EF,F0,74, 72,EF,E6,66	FlexiPanel identifier
00,00	Serial number not used
00,00	Checksum not implemented
02,00	Protocol version 3.0
01,00	Backwardly compatible to 2.0
09,00	Control Update From Server message ID
00,00	Flags – none
02,00,00,00	2 control blocks follow
54,53,52,00	Latch control ID
01,00,00,00	Control value size = 1
00	Control value (off)
54,00,00,00	Number Control ID
08,00,00,00	Control value size = 4
93,01,00,00	Control value 403

Control Partial Update From Server

The Control Partial Update From Server informs the client when the server is attempting to modify an incomplete portion of a control's value.

The Control Update From Server message was implemented in protocol version 3.0 and remains current. It is currently only defined for matrix controls.

The message starts with a header with message constant `0x0015`. The body that follows will vary depending on the controls being updated. The general format is:

DataSize	Contains
uint32	number `nCtl` of *Server Partial Update Blocks* to follow
then `nCtl` *Server Partial Update Blocks*	

Each Server Partial Update Block contains new data for an updated control. Only those controls which have been modified need to be transmitted. The general format is:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `sSUpdate` of the data to follow
`sSUpdate`	New control data

The control ID must be used to identify the type of control and therefore the control data format. The specific format for each control will now be described.

Matrix Server Partial Update Block

The Matrix Partial Server Update Block updates a single row of the matrix.

The *matrix server partial update block* updates a single row of the matrix and consists of:

DataSize	Contains
`uint32`	Control ID
`uint32`	Size `nDat` of the matrix value to follow
`nDat`	Single Row matrix value

The single row matrix value comprises of the following fields:

- The single row of integers as a linear array. The row value from the first column comes first, then the next column, etc. Each integer will be signed but may be 1- 2- or 4-byte depending on the matrix flags.

- If the matrix is of the *list* or *labels* matrix type, no X-axis data is sent.

- If the matrix is of the *XY* matrix type, the X-axis value is sent as a signed integers which may be 1- 2- or 4-byte depending on the matrix flags.

- If the matrix is of the *Date-Time* matrix type, the X-axis Date-Time value is sent as an 8-byte Date-Time value (see Date-Time Control Block, page 16).

- An four byte signed integer `NumValid` indicating the number of rows of the resulting matrix which contain valid data and whether it must be interpreted as an offset circular array:

    - If `NumValid` is zero or positive, the first `NumValid` rows contain valid data and the rest are unspecified.

    - If `NumValid` is negative, all rows contain valid data. The row elements of the matrix have been offset and row `R` of the matrix is located at Column Sub-Array element `E`, where

      ```
      E = ( R – NumValid ) % NumRows
      ```

      and `NumRows` is the total number of rows in the matrix. `E` and `R` are zero-based; `%` is the modulus operator; `NumValid` may be no more negative than `–NumRows`. If XY style of Date-Time style, this offset applies to the row data, too.

- A four byte signed integer `R` indicates which row is to be overwritten with the new row values. If `NumValid` is negative, this value is the Column Sub-Array element `E` to be updated.

Worked example

The following Control Partial Update From Server message sends an updates row of data to an XY matrix control consisting of a 2-byte row X value and three 1-byte cell values.

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`02,00`	Backwardly compatible to 3.0
`15,00`	Control Partial Update From Server message ID
`00,00`	Flags – none
`01,00,00,00`	1 control block follow
`54,53,52,00`	Matrix control ID
`0D,00,00,00`	Control value size = 13
`01,02,03`	Matrix cell values
`04,05`	Matrix X values
`04,00,00,00`	Number of valid rows is 4
`03,00,00,00`	Row to update is 3

Ping From Server

Bluetooth connections may be fail, *e.g.* if the client or server loses power or go out of range. Ping functionality is provided in order to detect this condition and treat it in a fail-safe manner.

If implemented by the server, and if supported by the client (as indicated by its header flags in earlier messages), the server regularly sends *Ping From Server* message, say, once every five seconds. (More frequently than this is problematic because the client may be busy or slow.)

In response to a *Ping From Server* message, the client should send a *Ping Reply From Client* message. If the time comes for a server to send a *Ping From Server* message and it finds that a *Ping Reply From Client* message has not been received in response to the previous Ping From Server message, it should conclude that the link is lost and enter a fail-safe disconnected state and await a new *Greetings From Client* message.

The *Ping From Server* message is sent as a header with message constant `0x0008` and no body. It was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Ping From Server* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`08,00`	Ping From Server message ID
`01,00`	Flags – Unicode flag only

Ping From Client

Bluetooth connections may be fail, *e.g.* if the client or server loses power or go out of range. Ping functionality is provided in order to detect this condition and treat it in a fail-safe manner.

If implemented by the client, and if supported by the server (as indicated by its header flags in earlier messages), the client regularly sends *Ping From Server* message, say, once every five seconds. (More frequently than this is problematic because the server may be busy or slow.)

In response to a *Ping From Client* message, the server should send a *Ping Reply From Server* message. If the time comes for a client to send a *Ping From Client* message and it finds that a *Ping Reply From Server* message has not been received in response to the previous *Ping From Client* message, it should conclude that the link is lost and enter indicate this state to the user, offering the option to reconnect.

The *Ping From Client* message is sent as a header with message constant `0x000A` and no body. It was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Ping From Client* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,01`	Serial number (Pocket PC in this example)
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`0A,00`	Ping From Client message ID
`01,00`	Flags – Unicode flag only

Ping Reply From Server

The server should send a *Ping Reply From Server* message promptly in response to a *Ping From Client* message.

The *Ping Reply From Server* message is sent as a header with message constant `0x000B` and no body. It was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Ping Reply from Server* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`0B,00`	Ping Reply From Server message ID
`01,00`	Flags – Unicode flag only

Ping Reply From Client

The client should send a *Ping Reply From Client* message promptly in response to a *Ping From Server* message.

The *Ping Reply From Client* message is sent as a header with message constant `0x000C` and no body. It was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Ping Reply From Client* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,01`	Serial number (Pocket PC in this example)
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`0C,00`	Ping Reply From Client message ID
`01,00`	Flags – Unicode flag only

Acknowledge From Server

The server should send an *Acknowledge From Server* message promptly in response to a message where the Acknowledge Requested flag was set in the header. Only the following message will be acknowledged: Update Control From Client.

The *Acknowledge From Server* message is sent as a header with message constant `0x000D` and no body. It was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Acknowledge from Server* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`0D,00`	Acknowledge message ID
`01,00`	Flags – Unicode flag only

Acknowledge From Client

The client should send an *Acknowledge From Client* message promptly in response to a message where the Acknowledge Requested flag was set in the header. Only the following messages will be acknowledged: Update Control From Server, New Control Panel From Server.

The *Acknowledge From Client* message is sent as a header with message constant `0x000E` and no body. It was implemented in protocol version 2.0 and remains current.

Worked example

A server sends a Unicode *Acknowledge from Client* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,01`	Serial number (Pocket PC in this example)
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`0E,00`	Acknowledge message ID
`01,00`	Flags – Unicode flag only

New Server

In some systems it is possible that a client may connect to the server Bluetooth layer before the FlexiPanel service has been initiated on the server. In this condition, the original *Greeting From Client* will have been lost and the server will not know it has a client connected.

If the system architecture is such that this problem might arise, the server may send a *New Control Panel From Server* message when it first initializes.

The *New Control Panel From Server* message is sent as a header with message constant `0x000F` and no body.

In response to a *New Control Panel From Server* message, the client should reply with a *Greetings From Client* message. Thus the server knows it is connected to a client.

The *New Server* message makes the FlexiPanel protocol robust to variations in boot-up sequence. It may result in two *Greetings From Client* messages being received by the server.

The *New Server* message was implemented in protocol version 2.2 and remains current.

Worked example

A server sends a Unicode *New Server* message as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`0F,00`	New Server message ID
`01,00`	Flags – Unicode flag only

Control Properties Update From Server

The Control Properties Update From Server message allows the server to update certain flags on a control and also the color of the control. The flags which may be modified are:

- `CTL_INVISIBLE` may change to hide or show a control.
- `CTL_ENSUREVISIBLE` may be set to request that a control be brought into view.
- `CTL_MSG_ICON_` flags.
- `CTL_MSG_RESP_` flags.
- If the color is to be changed, the `CTL_COLCHANGED` flag should be set; otherwise the color value will be ignored by the client.

In particular, the message is sent with the `CTL_INVISIBLE` clear in order to show the message box of a message control. (It does not need to be reset to invisible afterwards.)

The Control Properties Update From Server message was implemented in protocol version 2.0 and remains current.

The message starts with a header with message constant `0x0010`. The body size that follows will vary depending on the number of controls to be updated. The general format is:

DataSize	Contains
uint32	number `nCtl` of *Control Blocks* to follow
then `nCtl` *Control Properties Blocks*	

Control Properties Block

Currently the control properties block is the same for all types of control:

DataSize	Contains
`uint32`	Control Unique ID
`uint32`	Total size of all fields to follow, value `0x00000008`
`uint32`	Control Flags as discussed above
`uint32`	Microsoft RGB value 0x00BBGGRR.

Worked example

The following Control Properties Update From Server message sends updated properties for a latch control and a message control.

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`07,00`	Control Properties Update From Server message ID
`00,00`	Flags – none
`02,00,00,00`	2 control properties blocks follow
`54,53,52,00`	Latch control ID
`08,00,00,00`	Control properties size = 8
`40,00,00,00`	Flags `CTL_COLCHANGED`
`FF,00,00,00`	Set Color to Red
`4D,73,67,00`	Message Control ID
`08,00,00,00`	Control properties size = 8
`00,00,20,00`	Flag `CTL_MSG_RESP_OKCANCEL.` (Invisible flag not set, so message will be shown.)
`00,00,00,00`	Color value (ignored)

Files From Server

The *Files From Server* message sends the files of a file control to a client. It should be sent in response to a *Control Update From Client* message relating to the files control. A client which does not wish to support this control simply never sends the related *Control Update From Client* message.

The *Files From Server* message was implemented in protocol version 2.2 and remains current.

Whether or not the files sent replace any existing files (e.g. the files sent in a previous *Files From Server* message) is up to the server.

The message starts with a header with message constant `0x0011`. The body size that follows will vary depending on the number and size of the files toi be sent. The general format is:

DataSize	Contains
uint32	Control ID of the Files control
uint32	number `nFiles` of files to be sent
then `nFiles` *File Blocks*	

Files Block

A files block is sent for each file being sent:

DataSize	Contains
`uint32`	Length `szlName` of file name field to follow
`szlName`	File name with zero terminator
`uint32`	Size of file data
`lData`	File data

The file name should not include a path since nothing can be assumed about the file system on the client device.

If `CTL_FILE_HINT_HTML` is specified, the files should all be stored in the same directory and then a web browser should be directed to the first file. In this way, the server can serve up web pages locally without the need for an internet connection. The first file serves as the home page. The other files may be linked web pages or other supporting files such as images.

Worked example

The following Files From Server message sends two files `index.htm` (256 bytes) and `image.jpg` (1024 bytes). If the `CTL_FILE_HINT_HTML` flag was originally specified for the control, the client would display the file `index.htm` in its web browser; `image.jpg` might well reference `image.jpg` which it would expect to find in the same directory.

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`11,00`	Files From Server message ID
`00,00`	Flags – none
`46,69,6C,65`	Control ID `0x656C6946`
`02,00,00,00`	Number of file blocks = 2
`0A,00,00,00`	Length of file name 1
`69,6E,64,65, 78,2E,68,74, 6D,00`	File name 1, zero terminated
`00,01,00,00`	File 1 size
`(256 bytes)`	File 1 data
`0A,00,00,00`	Length of file name 2
`69,6D,61,67, 65,2E,6A,70, 67,00`	File name 2, zero terminated
`00,04,00,00`	File 2 size
`(1024 bytes)`	File 2 data

Files From Client

The *Files From Client* message sends the files of a file control to a server. It should be sent in response to a Control Update From Server message relating to the files control. A server which does not wish to support this control simply never sends the related Control Update From Server message.

Whether or not the files sent replace any existing files (e.g. the files sent in a *Files From Server* message) is up to the server.

The *Files From Client* message was implemented in protocol version 2.2 and remains current.

The message starts with a header with message constant `0x0014`. The body size that follows will vary depending on the number and size of the files toi be sent. The general format is:

DataSize	Contains
uint32	Control ID of the Files control
uint32	number `nFiles` of files to be sent
then `nFiles` *File Blocks*	

Files Block

A files block is sent for each file being sent:

DataSize	Contains
`uint32`	Length `szlName` of file name field to follow
`szlName`	File name with zero terminator
`uint32`	Size of file data
`lData`	File data

The file name should not include a path since nothing can be assumed about the file system on the client device.

The `CTL_FILE_HINT_HTML` flag is unlikely to result in the files being displayed; a more typical usage of the message in this case would be to update the web pages which are subsequently displayed in *Files From Server* messages.

Worked example

The following Files From Client message sends two files `index.htm` (256 bytes) and `image.jpg` (1024 bytes).

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74,72,EF,E6,66`	FlexiPanel identifier
`00,01`	Serial number (Pocket PC in this example)
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`14,00`	Files From Server message ID
`00,00`	Flags – none
`46,69,6C,65`	Control ID `0x656C6946`
`02,00,00,00`	Number of file blocks = 2
`0A,00,00,00`	Length of file name 1
`69,6E,64,65,78,2E,68,74,6D,00`	File name 1, zero terminated
`00,01,00,00`	File 1 size
`(256 bytes)`	File 1 data
`0A,00,00,00`	Length of file name 2
`69,6D,61,67,65,2E,6A,70,67,00`	File name 2, zero terminated
`00,04,00,00`	File 2 size
`(1024 bytes)`	File 2 data

Profile Request From Client

Client Profiles Overview

The basic FlexiPanel system passes logical controls which may then be displayed on any remote client and few assumptions are made about that client. It might be a Palm-sized PC, but it could equally be a completely audio implementation which is accessed by phone.

This approach is very flexible but on its own does not exploit each platform to its full capability. For example it does not specify how data is laid out on a screen or whether a line or bar chart should be used to display matrix data.

Consequently, a separate mechanism exists to send an additional, platform-specific *profile* which improves the layout of the user interface each specific client devices. Profile data is strictly optional – all clients must provide useable user interfaces given the basic FlexiPanel data.

Profile data is stored as groups of four 4-byte integers on the server: Device ID, Control ID, Attribute, Value.

The Device ID identifies the client type. The following are currently defined:

Value (hex)	Client platform
`0x00000100`	Pocket PC
`0x00000200`	Windows PC
`0x00000300`	Smartphone

Java phone client and Palm client do not currently exploit profile data. If you develop a client platform and wish to have a Device ID allocated, contact us.

The Control ID stores the control ID to which the attribute and its value apply. If the attribute is not specific to a particular control, the value `0x00000000` is specified.

The Attribute is the feature of the control which is specified by the associated Value. If the Control ID is nonzero, its interpretation will be specific to the type of control and so must be interpreted with reference to the control type.

When a client sends a profile request message, the server will respond with all the Control ID – Attribute – Value triplets for that client's Device ID type.

The actual profile definitions are client-specific and not part of the FlexiPanel protocol definition, other than to specify that they are groups of four 4-byte integers Device ID, Control ID, Attribute and Value. The definitions for Pocket PC, Windows and Smartphones are contained in the following C header files which are included in the FlexiPanel Designer developer's kit:

```
PocketPCProfiles.h
WindowsProfiles.h
SmartPhoneProfiles.h
```

Profile Request Message

The *Profile Request From Client* message asks the server to send profile data relevant to that client. If it has such profile data, it will respond with a *Profile Data From Server* message.

The *Profile Request From Client* message is sent as a header with message constant `0x0012`. It was implemented in protocol version 2.1 and remains current. The message body is:

DataSize	Contains
`0x04`	Device ID

Worked example

A Windows PC sends a Unicode *Profile Request From Client* message as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,02`	Serial number
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`12,00`	Profile Request message ID
`01,00`	Flags – Unicode flag only
`00,02,00,00`	Windows PC Device ID

Profile Data From Server

See the previous section, *Profile Request From Client* for an overview of profile data.

The *Profile Data From Server* message is sent in response to a *Profile Request From Client* message, if the server has profile data relating to that Device ID type.

The *Profile Request From Client* message is sent as a header with message constant `0x0013`. It was implemented in protocol version 2.1 and remains current. The message body is:

DataSize	Contains
uint32	number `nProfs` of profile blocks to be sent
then `nProfs` *Profile Blocks*	

Profile Block

A profile block is sent for each profile group whose Device ID matches the device ID in sent in the body of the Profile Request From Client message:

DataSize	Contains
`uint32`	Control ID
`uint32`	Attribute ID
`int32`	Value

It is permissible to send the message with a profile block count `nProfs` of zero.

Worked example

A server sends a *Profile Data From Server* message containing three profile blocks as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,00`	Serial number not used
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`13,00`	Profile Data message ID
`00,00`	Flags – None
`03,00,00,00`	Number of profile blocks = 3
`00,00,00,00`	Control ID 1 (general feature not control specific = 0)
`00,04,00,00`	Attribute 1 `WIN_SCREEN_HEIGHT`
`00,02,00,00`	Value 1 (512 pixels)
`00,00,00,00`	Control ID 2 (general feature not control specific = 0)
`00,08,00,00`	Attribute 2 `WIN_SCREEN_WIDTH`
`00,03,00,00`	Value 2 (768 pixels)
`4C,00,00,00`	Control ID 3 (Happens to be a latch control)
`50,01,00,00`	Attribute 3 `WIN_ATT_STYLE`
`01,00,00,00`	Value 3 `WIN_CST_LCH_ CHECKBOX` (checkbox style)

Program Device

The client is requesting that the server enter a product specific field programming mode. All communication thereafter will be product specific and will terminate by the server performing a device reset.

No attempt is made to prevent ‘hacking’ field programming attempts. This is the responsibility of the product specific field programming mode.

The *Program Device* message is sent as a header with message constant `0x0081` and no body. It was implemented in protocol version 2.3 and remains current.

Worked example

A server sends a Unicode *Acknowledge from Client* as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,01`	Serial number (Pocket PC in this example)
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`81,00`	Program Device message ID
`01,00`	Flags – Unicode flag only

Device-Specific Data From Client

Message

The *Device-Specific Data From Client* message sends the data relevant to that client. Servers that do not know how to process the message should ignore it.

The *Device-Specific Data From Client* message is sent as a header with message constant `0x0082`. It was implemented in protocol version 3.0.00006 and remains current. The message body is:

DataSize	Contains
`0x04`	`NumByte` Number of bytes in remainder of message
`NumByte`	Message

Worked example

A custom client sends a Unicode *Device-Specific Data From Client* message 12 34 as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74, 72,EF,E6,66`	FlexiPanel identifier
`00,02`	Serial number
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`82,00`	Profile Request message ID
`01,00`	Flags – Unicode flag only
`02,00,00,00`	NumByte = 0x02
`12 34`	Message 12 34

Device-Specific Data From Server

Message

The *Device-Specific Data From Server* message sends the data relevant to that client. Servers that do not know how to process the message should ignore it.

The *Device-Specific Data From Server* message is sent as a header with message constant `0x0083`. It was implemented in protocol version 3.0.00006 and remains current. The message body is:

DataSize	Contains
`0x04`	`NumByte` Number of bytes in remainder of message
`NumByte`	Message

Worked example

A custom client sends a Unicode *Device-Specific Data From Server* message 12 34 as follows:

Bytes in order of transmission (hex)	Meaning
`48,EF,F0,74,` `72,EF,E6,66`	FlexiPanel identifier
`00,02`	Serial number
`00,00`	Checksum not implemented
`02,00`	Protocol version 3.0
`01,00`	Backwardly compatible to 2.0
`83,00`	Profile Request message ID
`01,00`	Flags – Unicode flag only
`02,00,00,00`	NumByte = 0x02
`12 34`	Message 12 34

FlexiPanel Protocol Revision History

The table below details the evolution of the FlexiPanel protocol. Version numbers indicate back-compatible changes which are significant enough to merit compliance with the new feature. For example, version 3.0 requires that unrecognized control styles and messages are managed gracefully: to claim compliance with version 3.0, these features must be supported.

Client-specific profiles features are optional and not listed here.

Version	***Detail***
1.0	Original infrared protocol
2.0	Bluetooth protocol initial release
2.2	Matrix control added
2.3	*Name Descriptor* expected on all controls for display purposes.
3.0	Unrecognized or unsupported messages, controls and descriptors must be ignored without fault and user must be aware that client is not providing full service.
3.0	Partial Update From Server message added
3.0	Servers must support authentication
3.0	Message Box can return a value
3.0	Image control added
3.0.00006	Device-specific data messages added

Protocol Errata History

The table below details corrections to this the documentation and when the correction was made.

Date	***Detail***
5-May-05	Profile request from client message body amended
24-Sep-05	Control update from server message constant previously wrong, corrected to 0x0015

Glossary and Notation

Hex Notation

Throughout this document, numbers with an ‘0x’ prefix should be assumed to be in hex. For example, 0xFF is completely equivalent to decimal 255.

In some partners’ documentation, a ‘$’ prefix is used in place of an ‘0x’ prefix. ‘$FF’ is equivalent to ‘0xFF’ and decimal 255.

In some partners’ documentation, a ‘H’ suffix is used in place of an ‘0x’ prefix. ‘FFH’ is equivalent to ‘0xFF’ and decimal 255.

Data Types

Data types are C standard data types; no floating point is used. C standard notation and calling conventions are assumed. Integers are explicitly defined as:

bool – unsigned char, zero for *false,* otherwise *true*

byte – 8 bit unsigned integer

int16 – 16 bit signed integer

int32 – 32 bit signed integer

signed char – 8 bit signed integer

uint16 – 16 bit unsigned integer

uint32 – 32 bit unsigned integer

unsigned char – 8 bit unsigned integer

word – 16 bit unsigned integer

Glossary

> *symbol* – Navigation drilldown to a particular item in a piece of software. A list of phrases separated by **>** symbols means: go to the program, menu or tab indicated by the first phrase, look for the second phrase and select it, look for the third phrase and select it, etc, until you find the item at the end of the list.

Big-Endian – see *Endian.*

Buffer – A linear region of memory designed for storing data entering from or departing to a communications channel.

Circular buffer – A ‘first-in-first-out’ buffer which wraps around. It has a start pointer indicating when the next byte to be dispatched (i.e. read or transmitter) and an end pointer indicating the last piece of data to be dispatched. The start pointer advances when its data is dispatched; the end pointer advances when new data arrives. When either pointer reaches the end of the buffer it starts at the beginning again. If the end pointer catches up with the start pointer, the buffer is full and a buffer *overrun* occurs.

<CR> – The ASCII carriage return character 0x0D.

FlexiPanel client – Hardware or software that creates a control panel when requested to by a FlexiPanel server.

Endian – Refers to which byte is stores / transmitted first in multibyte integers. In *Little-Endian* format, bytes are in increasing order of significance, least significant byte first. In *Big-Endian* format, bytes are in increasing order of significance, least significant byte first. In general, Flexipanel Ltd uses little-endian format, but there are exceptions.

FlexiPanel server – Hardware or software that requests a control panel to be created on a FlexiPanel client.

IC – Integrated Circuit.

KIPS – thousand instruction cycles per second.

<LF> – The *ASCII* line feed character 0x0A.

Little-Endian – see Endian.

LSB – least significant bit or byte, depending on context.

MIPS – million instruction cycles per second.

MSB – most significant bit or byte, depending on context.

OS – Operating System.

Overrun – A circular buffer overruns if an attempt is add more data to it when it is full (see definition of *circular buffer).*

PWM – Pulse Width Modulation.

Underrun – A circular buffer underruns if an attempt is made to dispatch data from it when it is empty.

Unicode – Two-byte integer array representing text characters. ASCII characters keep the same values in the Unicode character set.

User – The person using the finished product (as opposed to the *Developer).*

Zero Terminator – A zero-valued character used to indicate the end of a string of characters.

Legal Notices

If any of this is not clear, contact FlexiPanel Ltd for clarification.

General

FlexiPanel technology should not be used in life critical devices without the permission FlexiPanel Ltd.

FlexiPanel Ltd makes every effort to ensure, but cannot warrant, that its products and documentation are without errors and omissions. However FlexiPanel Ltd does not accept liability for consequent loss or injury as a result of using its products or interpreting its documentation. FlexiPanel Ltd will not be responsible for any third party patent infringements arising from the use of its products.

FlexiPanel Ltd reserves the right to make changes to its technology and documentation in order to improve reliability, function or design.

Software Libraries

FlexiPanel Ltd provides software such as the Toothpick Services library exclusively for use with products made by FlexiPanel Ltd. It is not permitted to use the libraries except with products made by FlexiPanel Ltd. It is not permitted to reverse engineer the security features designed to ensure that the library only works with products made by FlexiPanel Ltd.

FlexiPanel Protocol

The FlexiPanel protocol and the products which use it are protected by pending patents and copyright law.

The FlexiPanel protocol allows *servers* to create user interfaces on remote *clients.*

Client software and products are freely distributable as far as we are concerned and you can do with them what you like. You can also freely produce your own client software and products which use the FlexiPanel protocol.

We make a living from licensing the FlexiPanel servers and providers of FlexiPanel server products must pay us an agreed license fee. If you buy FlexiPanel hardware products from FlexiPanel Ltd, this license is implicit. You may, under license, also make your own hardware or software FlexiPanel server products – contact us for details.

Contact Details

The FlexiPanel Protocol is owned and designed by FlexiPanel Ltd. For technical support, contact us at:

FlexiPanel

FlexiPanel Ltd
2 Marshall St, 3rd Floor
London W1F 9BB, United Kingdom
www.flexipanel.com
email: support@flexipanel.com