DIGITAL CAMERA CONNECTIVITY SOLUTIONS USING THE PICTURE TRANSFER PROTOCOL (PTP)

Petronel Bigioi[1], George Susanu[2], Peter Corcoran[3] and Irina Mocanu[4]
[1]Dept. of IT, National University of Ireland, Galway
[2]Accapella Ltd., Ireland
[3]Dept. of Electronic Engineering, National University of Ireland, Galway, Ireland
[4]Dept. of Computer Science and Engineering, University Politehnica of Bucharest, Romania

Abstract

Interconnectivity of digital camera with other devices is one of the main concerns of consumers and digital camera manufacturers. Interconnectivity features in digital cameras allow more consumer-friendly usage of digital cameras. Moreover, with a suitable application layer software, digital photographs can be sent directly from the camera to a desired target: disk storage, printer, web site, as an e-mail message or web print, using a single, purpose-designed, communication protocol: the Picture Transfer Protocol, PTP.

1 Introduction

Digital photography continues to gain market share from conventional photography. This is due to a number of factors: no development costs; no film costs; easy picture preview before printing or saving the image; easy sharing; portability of images; easy presentation in a variety of electronic & print formats, etc. Despite these benefits conventional photography is still more practical in the eyes of many consumers. Thus, although consumers are converting to digital the rate of growth has slowed very significantly in the last couple of years. This situation exists because the workflow for digital photography, starting from the acquisition process through to the final store, print or share process, remains complex and continues to form a usage barrier for the majority of consumers. In brief, digital photography remains targeted towards skilled PC users.

A couple of years ago each digital camera manufacturer had its own specific communication protocol to access and control a digital still camera. This method had a number of disadvantages: the camera manufacturer had to provide device drivers for all the operating systems and hardware platforms that they wanted to support and this added costs to the digital camera selling price. Further the end-users were expected to have a certain level of technical ability in order to understand the whole process of digital photography.

Even if the above method still exists, nowadays, the most popular approach is to make the digital camera look like a storage device whenever it is attached to a PC or an embedded system. Even if this method has become the present norm and more and more camera manufacturers adopt it, a number of downsides and limitations have already started to emerge. Firstly the digital camera is only able to deliver pictures when attached to a mass storage reader device. Secondly, the camera becomes a peripheral, or slave device to the PC: an upload process from the digital camera to the PC or receiving device can not be initiated using the mass storage approach. Thus an automation of the digital photography workflow is not practical. Thirdly there is no way to control the digital camera using the mass storage solution because it appears to the PC only as a passive storage device.

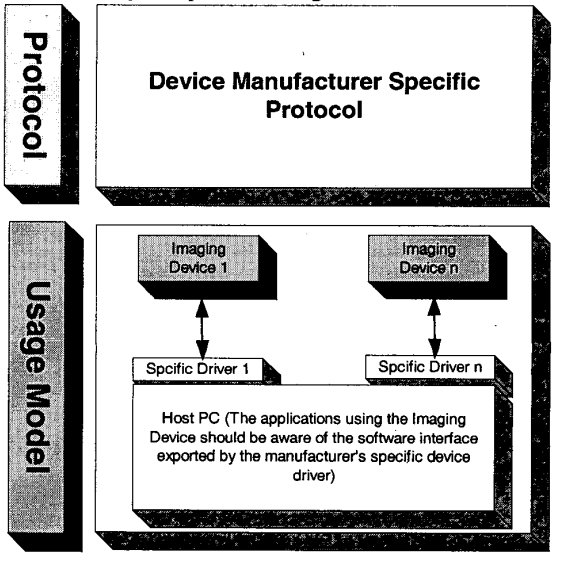


Fig 1: DSC Communication Protocols - Past

This paper will describe a recent standards effort (PIMA 15740) from the *Photographic & Imaging Manufacturer's Association* for connectivity of digital still photography devices. This standard is known as the *Picture Transfer Protocol* (PTP) for short. The intention of the PTP standard is to replace and unify the communication protocols between still imaging devices and other receiving devices. Most imaging devices include hardware interfaces that can be used to connect to a host computer or other imaging devices, such as a printer. A number of new, high-speed interface transports have recently been developed, including IrDA, USB, and IEEE1394. This standard is designed to cover the requirements for communicating with still imaging devices over a variety of transports. This includes communications with any type of

Manuscript received June 24, 2002

devices, including host computers, direct printers and other still imaging devices over suitable transports. The requirements include standard image referencing behavior, operations, responses, events, device properties, datasets, and data formats to ensure interoperability.

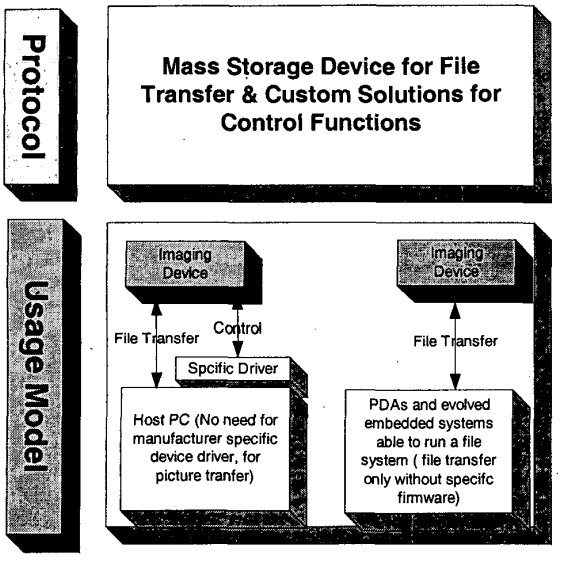


Fig 2: DSC Communication Protocols - Present

Standardizing the operations and data requirements for still imaging devices through a standard such as PTP will assist transport implementers, platform aggregators, service providers and device manufacturers by providing a common ground for interface support. It will also assist developers of host software and image receiving devices by ensuring that their products can interface with many different imaging devices from different manufacturers, and assist users by ensuring that the imaging devices they purchase will inter-operate with those of different manufacturers.

One of most interesting facts about the PTP standard is that provides optional operations, formats and defined extension mechanisms, which will allow digital camera manufacturers to use the communication standard even if they want to implement custom behavior for their imaging devices. This standard has been designed to appropriately support popular image formats used in digital still cameras, including the EXIF and TIFF/EP formats defined in ISO 12234-1 and ISO 12234-2, as well as the Design Rule for Camera File System (DCF) and the Digital Print Order Format (DPOF).

This paper will provide detailed explanation of the following main issues:

- Description of the PTP as a common protocol for any device to exchange images with still imaging device, either by retrieving images from it or by sending images to it.
- Usage models (push and pop models) and usage scenarios (a number of typical usage scenarios for the two typical usage models).
- PTP transport requirements and practical examples using the USB transport. Operating systems support for PTP USB devices.
- Proposal for PTP implementations over wireless transports. Transport specific issues and solutions.
- Imaging devices as Internet connected devices using PTP mapped on TCP/IP protocol. Firewalls, authentication and security issues and solutions.

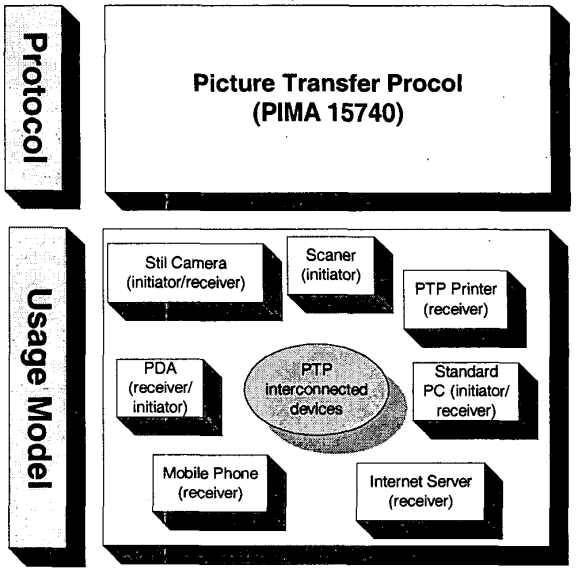


Fig 3: DSC Communication Protocols – Future

2 PTP Description

This section describes the PTP and the main guidelines followed by this protocol.

2.1 PTP Device Roles

Rather than having a host master to a slave device protocol description or a peer to peer description, the PTP refers to the components engaged in a picture transfer as *Initiator* and *Responder*. The PTP defines the *Initiator* as being the device that initiates the connection - issues the *OpenSession* PTP command - while the *Responder* is defined as the device that responds to operation requests such as the *OpenSession* request.

Devices, in the PTP model, can be *Initiators*, *Responders* or both. For instance, a PC can be configured only as an *Initiator* device while a USB camera can be only a *Responder*. Similarly, a Bluetooth camera, that opens a connection to a Bluetooth/PTP printer and pushes pictures for print, can be only an *Initiator* while the corresponding

printer can be only a *Responder*. However, a digital camera that can connect to other digital cameras and is able to both initiate and receive a PTP session will have to be capable of behaving both as *Initiator* and *Responder*.

Usually, the *Initiator* will have a form of graphical user interface, that the user can see/browse thumbnails, select and chose an appropriate control action, and so on. Moreover, the *Initiator* device has to implement the device enumeration and transport mapping (in the case that multiple, PTP-compliant transports are supported), all in a transport specific manner. Typically, a *Responder* will not have a graphical user interface or multiple transport support.

2.2 PTP Sessions

In order for two PTP devices to exchange information about pictures or metadata, a PTP session has to be established. A session is a logical connection between the PTP devices, over which the object identifiers, or *ObjectHandles*, and storage media identifiers, or *StorageIDs*, are persistent. A session is considered opened after the *Responder* returns a valid response to the *OpenSession* operation requested by the *Initiator*. A session is closed after the *CloseSession* operation is completed or the transport closes the communication channel, whichever occurs first.

The only operation or data traffic allowed outside the session is the GetDeviceInfo operation and the *DeviceInfo* dataset. A device can issue/accept a *GetDeviceInfo* operation outside a session. A session is needed in order to transfer descriptors (*StorageInfo*, *ObjectInfo*, etc), images or any other objects between devices. Any data communicated between devices is considered valid unless a specific event occurs specifying otherwise.

Each session is represented by a unique 32 bit identifier (UINT32) that is assigned by the *Initiator* using the OpenSession operation request and it has to be non-zero.

2.3 Protocol Model (Transactions)

The PTP is specified using the transaction model. A transaction is composed of a request operation, followed by an optional data transfer and a response. Each transaction has an identifier (*TransactionID*) that is session unique and comprise of a 32 bit unsigned number (UINT32). The transaction IDs are a sequence of numbers starting with 0x00000000 (for the *SessionOpen* transaction) and increasing with every following operation. With a few exceptions, all the transactions are synchronous and atomic operations, having blocking execution within the session. Devices that support multiple sessions must be able to keep the sessions opaque and asynchronous to each other.

The asynchronous transactions (i.e. *InitiateCapture*) are treated as synchronous in the initial phase (when specifying the operation), when a response indicating that the operation request has been successful or not is enough. Asynchronous events are used to handle the communication initiated by such operations (i.e. the availability of new objects becoming available on the device's storage as result of an *IntiateCapture* operation). The completion of an asynchronous transaction is signaled by a specific event (i.e. *CaptureComplete* event should be issued). If the *Initiator* issues another asynchronous operation while a previous one is still in progress, the device should issue a *Device_Bussy* response.

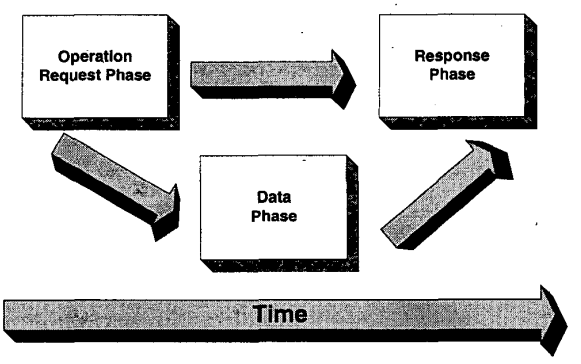


***Fig 4:** PTP Transaction Phases*

In the PTP the transactions consist of three phases. The data phase is an optional phase and up to the operation it can be present or not. If the data phase is present, the data can be sent either from the *Initiator* to the *Responder* or from the *Receiver* to the *Initiator*, but it may not consist of data transferred in both directions. Only one transaction at a time can take place within a session.

Field	Size (bytes)	Type
OperationCode	2	UINT16
SessionID	4	UINT32
TransactionID	4	UINT32
Parameter 1	4	ANY
Parameter 2	4	ANY
Parameter 3	4	ANY
Parameter4	4	ANY
Parameter 5	4	ANY

Operation Dataset

Field	Size (bytes)	Type
ResponseCode	2	UINT16
SessionID	4	UINT32
TransactionID	4	UINT32
Parameter 1	4	ANY
Parameter 2	4	ANY
Parameter 3	4	ANY
Parameter4	4	ANY
Parameter 5	4	ANY

Response Dataset

***Table 1**: Request and Response Datasets*

2.3.1 Operation Request Phase

In this phase the operation request dataset is transferred from the *Initiator* to the *Responder*. This dataset is given in ***Table 1***.

2.3.2 Data Phase

The data phase is optional and is used to transmit a data set that is larger than may be accommodated by the operation or response phases. This phase is used to transfer information that is not specified by small data types (i.e. typically the transfer of a binary image is achieved during

this phase). During the data phase, the information is transferred either from *Initiator* to *Responder* or from *Responder* to the *Initiator*, but never in both directions.

2.3.3 Response Phase

In this phase the response dataset is transferred from the *Responder* to the *Initiator*. The response dataset is very similar to the operation request dataset and it is presented in ***Table 1***.

2.4 Vendor Extensions

Imaging devices manufacturers can extend the PTP command set by defining their own commands, events and properties. Of course, only vendor specific software will take advantage of this vendor specific functionality. The VendorExtensionID and the VendorExtensionVersion are fields in DeviceInfo structures that uniquely identify the vendor extensions. A device has to check those fields before using a vendor extended operation, event or property. The VendorExtensionID is assigned by PIMA while VendorExtensionVersion is maintained internally by each manufacturer.

3 Usage Models

The purpose of this section is to describe how PTP devices can interact and what exactly the flow of operations is in each usage model.

The PTP can be associated with a dynamic master/slave protocol, where the master role is undertaken by the *Initiator* device while the slave role is assigned to the *Responder*. The *Initiator* determines the flow of operations while the *Responder* can issue responses to the operations as well as events. The events can be associated with an operation, but they can appear completely asynchronously as well, so the *Initiator* should be prepared to deal with events that occur outside of the boundaries of a transaction.

3.1 Pull Scenarios

In the pull mode, the *Initiator* retrieves the objects from the *Responder* and it is usually the way that the PTP communication is implemented in a device that has a rich user interface.

	Initiator Action	Parameter1	Parameter2	Parameter3	*Responder* Action
1	*GetDeviceInfo*	0x00000000	0x00000000	0x00000000	Returns *DeviceInfo* dataset
2	*OpenSession*	*SessionID*	0x00000000	0x00000000	Opens a PTP session
3	*GetObjectHandles*	0xFFFFFFFF	0xFFFFFFFF	0x00000000	Returns Object Handles in a *ObjectHandlesArray* structure
4	GetObjectInfo	*ObjHandle* 1	0x00000000	0x00000000	Returns *ObjectInfo* dataset for ObjHandle 1
5	Repeat Step 4 for each ObjectHandle	*ObjHandle* n	0x00000000	0x00000000	Returns *ObjectInfo* dataset for *ObjHandle* n
6	*GetObject*	*ObjHandle* 1	0x00000000	0x00000000	Returns object 1 data
7	Repeat Step 6 for each *ObjectHandle*	*ObjHandle* m	0x00000000	0x00000000	Returns the object m data
8	*CloseSession*	0x00000000	0x00000000	0x00000000	Closes the session

Table 2: *Getting Images from a PTP Device*

This usually involves three steps that the user has to go through in order to complete the operation: select a PTP device to communicate with (if there are multiple devices in the proximity), download and select thumbnails (from the remote device) and download the selected pictures. A typical scenario for this usage model would be where the *Initiator* requests all images objects from the *Responder*, ignoring other objects (non-images) and associations. This is described in ***Table 2***.

Another scenario would be where the *Initiator* requests all thumbnails from the *Responder*, ignoring associations and non-images objects. This is described in ***Table 3***.

	Initiator Action	Parameter1	Parameter2	Parameter3	*Responder* Action
1	*GetDeviceInfo*	0x00000000	0x00000000	0x00000000	Returns *DeviceInfo* dataset
2	*OpenSession*	SessionID	0x00000000	0x00000000	Opens a PTP session
3	*GetObjectHandles*	0xFFFFFFFF	0xFFFFFFFF	0x00000000	Returns the *ObjectHandlesArray*
4	*GetObjectInfo*	ObjHandle 1	0x00000000	0x00000000	Returns *ObjectInfo* dataset for ObjHandle 1
5	Repeat Step 4 for each *ObjectHandle*	ObjHandle n	0x00000000	0x00000000	Returns *ObjectInfo* dataset for *ObjHandle* n
6	*GetThumb*	ObjHandle 1	0x00000000	0x00000000	Returns the object 1 data
7	Repeat Step 6 for each *ObjectHandle*	ObjHandle n	0x00000000	0x00000000	Returns the object n data
8	*CloseSession*	0x00000000	0x00000000	0x00000000	Closes the session

Table 3: *Get all Thumbnails from a PTP Device*

3.2 Push Scenarios

Push mode consists of an *Initiator* sending one or more objects to the *Responder*. Usually, the *Initiator* has a rich user interface (i.e. PC or digital camera). The *Responder* receives the pictures and other data from the *Initiator*. This mode is appropriate for transfers from a device with a rich UI to devices that don't have image display capabilities. For instance, this is useful for direct print operations (over either USB, 1394 or Bluetooth).

	Initiator Action	Parameter1	Parameter2	Parameter3	*Responder* Action
1	*GetDeviceInfo*	0x00000000	0x00000000	0x00000000	Returns *DeviceInfo* dataset
2	*OpenSession*	SessionID	0x00000000	0x00000000	Opens a PTP session
3	*Optional Vendor command or property setting*	Specific	Specific	Specific	Configures the *Responder* for specific operations on the incoming pictures
4	*SendObjectInfo*	0x00000000	0x00000000	0x00000000	Returns StorageID, Parent ObjHandle, ObjHandle
5	*SendObject*	0x00000000	0x00000000	0x00000000	Writes data to store
6	Repeat Step 3,4 for each *ObjectHandle*	-	-	-	-
7	*CloseSession*	0x00000000	0x00000000	0x00000000	Closes the session

Table 4: *Push Scenario*

A typical push scenario is the one illustrated in ***Table 4***, where the *Initiator* transfers a number of images to a default location on the *Responder*. Prior to data transfer, the *Initiator* can execute a vendor specific command that can configure the *Responder* for a specific action that has

to take upon the incoming pictures. For example, if the *Responder* is a printer, than this operation can set the print format. Alternatively, vendor specific properties can be used for the same purpose.

4 USB Transport

In this section we describe an example implementation of PTP over the USB transport. This is a typical implementation for a USB digital camera.

4.1 Overview of endpoints

PTP over USB uses a number of endpoints (source or sink of data on USB devices) to provide the required functionality.

4.1.1 Default endpoint

The default endpoint has the address: 0x00 and it is a control (default) type endpoint.

The default endpoint is usually used for standard USB requests and for class specific requests. Class specific requests are:

- ***Device Reset Request*** (out) - used by the host to reset the device.
- ***Get Device Status Request*** (in) - used by the host to determinate the device status after the host cancels a transaction or an endpoint becomes stalled.
- ***Cancel Request*** (out) - used by the host to cancel the current transaction.
- ***Get Extended Event Data*** (in) - used by host to retrieve the event data from device that is too big for a standard (PTP) event from the interrupt endpoint. All PTP events must be received from the interrupt endpoint.

4.1.2 Data-In endpoint

The Data-In endpoint can have any address in range 0x81-0x8F, but Interrupt endpoint address, It is a Bulk In type endpoint. It is used for responses and the data-in phases of the transaction, from the device to the host.

4.1.3 Data-Out endpoint

The Data-Out endpoint can have any address within the range 0x01 - 0x0F. It is a Bulk Out type endpoint. It is used for data-out and operation phases of the transaction, from the host to the device.

4.1.4 Interrupt endpoint.

The Interrupt endpoint can have any address in range 0x81 - 0x8F but Data-In endpoint address. It is an Interrupt type endpoint and it is used to transfer for event data from device to host.

4.2 Mapping PTP to USB transport

USB transport standard for still digital cameras supports only single session mode. Even if device is capable of supporting multiple sessions the USB transport protocol is not able to handle more than one session (the *Initiator* can't open concurrent session to the *Responder*). This does not affect the service quality because multiple sessions do not provide many benefits to a PC user and, on the other hand, single session support is much simpler for a device implementation.

The PTP communication protocol implies five different kinds of communication "primitives": Operation Request Phase of a transaction, Data-In Phase of a transaction, Data-Out Phase of a transaction, Response Phase of a transaction and Events. All this primitives are encapsulated and transported through USB in "containers". The containers that are part of a transaction (transaction phases) are transferred synchronously relative to each other through bulk endpoints. The Event Container is transferred asynchronously relative to transaction containers through an interrupt endpoint.

Byte offset	Length (bytes)	Field Name	Description
0x00	4	Container Length	The whole size of the container from offset 0x00 including payload
0x04	2	ContainerType	One of the following values: • 0 (undefined) – must not be used; • 1 (CommandBlock) – maps a PTP Operation Request Dataset; • 2 (DataBlock) – maps the Data-In and Data-Out phases of a transaction; • 3 (ResponseBlock) – maps PTP Response Dataset; • 4 (EventBlock) – maps PTP Event Dataset
0x06	2	Code	The value of PTP Operation Code, Response Code or Event Code
0x08	4	TransactionID	The value of associated PTP transaction
0x0C	?	Payload	The content of the payload depends on ContainerType field

Table 5: Generic USB Container Structure

This approach limits the host to act as an *Initiator* and device as a *Responder*. Also events can be sent from the device to host only (this is suitable for most events). There is a special case of a *Cancel Transaction* Event, which is sent from the host to the device. This event is transferred through the default endpoint as a class specific request. Note that even though the *Cancel Transaction* Event is handled separately by the USB transport, the support for these events is required and the corresponding code (0x4001) must be present in *Device Info Dataset*.

On the other hand, the device may cancel the transaction by stalling the corresponding endpoint rather than sending the *Cancel Transaction* Event. This is a very good example of transport specific implementation of the canceling mechanism specified by the PTP layer. USB Transport defines also other class-specific requests issued to the default endpoint.

4.3 Containers

A Container is a USB Transport structure used to encapsulate and transport PTP communication primitives. The PTP "primitives" that are carried over using this USB structure, are defined in ***Table 5***.

4.4 Common Implementation Mistakes

During investigations of different implementation of PTP transport for USB several issues have been found at both *Initiator* and *Responder* sides. Most of these arise because of inconsistencies in the specification of the USB transport. Others are just violations of this specification. Because PTP is the first standard protocol for imaging devices and is not mature enough, we have to deal with the consequences. The cases listed below are not necessarily a sign of a bad design in PTP itself. In some cases the specification needs to be refined to clarify specific cases that are presently implemented differently by different device vendors.

4.4.1 Cancellation problems

Some camera manufacturers do not implement transaction cancellation properly. The guide here is that USB has a transport specific implementation of cancel by means of the class specific *Cancel Request* which is not taken into consideration by most PTP implementations. As a result, PTP clients, such as the Windows Imaging Architecture (WIA) mini-driver, wait until a transaction is completed and discard the results. This can lead to significant delays after the user cancels the operation. On the other hand cancellation from *Responder* side is achieved by stalling the USB pipe and setting a *Transaction_Cancelled* response code as a device status. The majority of *Initiator* implementations treat this as a *Responder* error. In general this is the correct approach because the transaction did not succeed, but in some cases the *Initiator* may refine the reason for failure as a cancellation to differentiate it from cases when the transaction failed because of transport error.

4.4.2 Incorrect error reporting techniques

Many of camera implementations have a "habit" of reporting PTP errors by using a USB level approach. In such cases they usually stall the pipe and set the appropriate response code in device status; cancel a transaction (also by stalling the pipe and reporting a *Transaction_Cancelled* code); stall the pipe without identifying a reason (in this case the camera often needs a reset); or even reset the USB connection. The correct approach is to use the Response Phase to report the appropriate PTP error code.

4.4.3 Undefined Parameters – Transport Issues

USB transport says that undefined parameters of operation, response and event datasets need not be included in a container which incorporates a "number of parameters" field. However the undefined parameters may also be transferred in containers with zero values as documented in the PTP specification. The receiver of the container should plan to handle both cases.

4.4.4 USB NULL packet not handled

Even if the USB containers specify their length in the header, the end of the container is still indicated by a short or a NULL packet. When the container is not a multiple of the USB packet length, then a short packet is received. In this case there is no need for a NULL packet transmission. The NULL packet must be explicitly issued and handled if the container length is an exact multiple of the USB packet length. Some implementations do not handle the NULL packets properly.

4.4.5 Reset problems

Some camera implementations do not reset the PTP session after a *Reset Request* or do not handle this request at all. That usually leads to a *Session_Already_Opened* error which is not expected by the *Initiator* after a reset.

4.4.6 Timeout issues

Neither PTP nor the USB transport specification provides any timing requirements between transaction phases or inter-phase transfers. It is clear that a camera implementation of PTP must know how to deal with cases when the *Initiator* is not responding for a long time during a transaction and take appropriate action (e.g. put a message on OSD and reset the USB connection), but this should be done after a reasonably long timeout (probably not less than a minute). With too short a timeout the *Initiator* could be suspended by the operating system for a couple of seconds and the PTP transaction will be broken before the *Initiator* has a chance to regain control.

5 Bluetooth Transport

This section describes a proposal to map the PTP over a Bluetooth transport.

5.1 BPTP

As mentioned before, in the PTP there are five main primitives that can be identified. These are: operation requests, operation responses, data in, data out and events.

While the first four primitives are part of the transaction model described in one of the earlier sections, the fifth one, events, is a completely asynchronous one during the PTP session. Therefore, the transport should provide a minimum of two logical channels to the PTP, for an easy mapping. Bluetooth transport has the ability to provide this at the L2CAP (Logical Link Control and Adaptation Protocol) level. So, the natural level to interact with the bluetooth protocol stack is the L2CAP layer.

All the PTP communication should go over two L2CAP logical data channels, one dedicated for the event communication and the other for command requests, responses and data traffic. The L2CAP layer is providing high level protocol multiplexing and packet segmentation and reassembly (up to a maximum packet size). It will not be enough just to send PTP data over L2CAP since the size

of the data may exceed the maximum L2CAP packet size. Usually, the L2CAP maximum transmission unit (MTU) will be exported using an implementation specific service interface, while the minimum MTU size is 48 bytes and it should be accommodated by any L2CAP implementations.

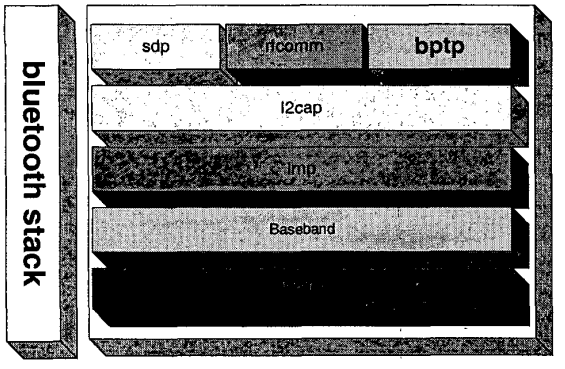


Fig 5: Bluetooth Stack

It is the responsibility of the higher layer protocol to limit the size of the packets sent to the L2CAP layer bellow the MTU limit. Therefore, in order to map the PTP over bluetooth at this level, we need to define a simple protocol specification (BPTP – **B**luetooth **PTP**) that will deal with this kind of issues (see ***Fig 5*** for its place in the bluetooth protocol stacks architecture).

BPTP expects from L2CAP a reliable transport layer, error free communication channels. It is a packet based transport protocol. All the communications between the two bluetooth imaging devices will go over two L2CAP logical data channels. The events (PTP event datasets) are transported separately from the Operation requests/responses and data, because of their asynchronous nature. Using a separate L2CAP logical channel for event mechanism (therefore using an out of band communication channel), the implementation of the protocol will be very much simplified.

The BPTP should perform segmentation and reassembly of the application data (PTP data structures) whenever this data exceeds the MTU negotiated by the L2CAP layers.

5.1.1 Command/Data Transport Channel

The BPTP Command/Data transport channel is based on a L2CAP logical data channel. This channel is dedicated to the data transfer during a PTP transaction: operation request phase, data phase (either data in or data out phase) and response phase.

Since the size of data transferred in the data phase could exceed the minimum MTU size of L2CAP (48 bytes) it is the only type of data that may require segmentation and reassembly. Moreover, in order to ensure an error-free operation, this data is subject to flow control mechanism as well. This channel is opened and configured by the *Initiator* PTP device and is identified by the local L2CAP logical channel identifier (LCID).

5.1.2 Event Transport Channel

The BPTP Event transport channel is based on a L2CAP logical data channel. This channel is dedicated to PTP Event transport. No segmentation or reassembly is needed for this channel, since all the events have a fixed size not exceeding 22 bytes (less than the minimum L2CAP MTU which is 48 bytes). No flow control mechanism is required for this channel. This channel is opened and configured by the *Initiator* PTP device. This channel is identified by the L2CAP local logical channel identifier (LCID).

5.2 Transport Channel Management

The PTP layer (in the top of BPTP) should be abstracted from the transport mechanism used (i.e. the existence of the transport specific Command/Data and Event channels). Therefore, the BPTP layer will rely on the existence of two established L2CAP layer channels between the devices, prior to transporting any PTP structures.

5.2.1 Transport Channels Establishment

In the communication between *Initiator* and *Responder* it is the responsibility of the *Initiator* to begin the device connection and establish a BPTP transport channel to the *Responder*.

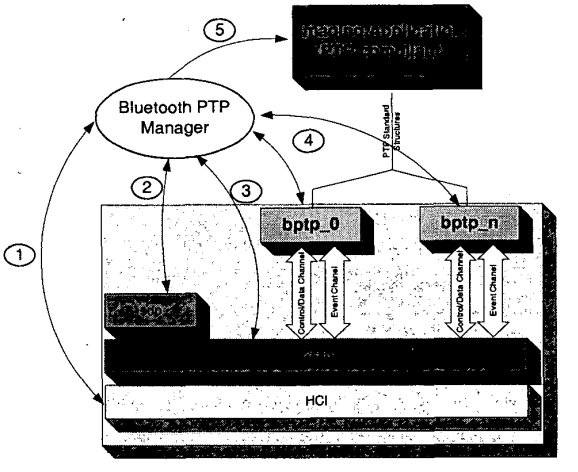


Fig 6: Suggested Behavior for Bluetooth PTP Initiator

Our approach to this is presented in ***Fig 6***, where we introduce a new software component - the Bluetooth/PTP manager, which can be an extension of the standard Bluetooth stack manager. This component performs the following tasks:

1. *Device inquiry*, network search for any local Bluetooth devices.
2. *Device service discovery* on each physical device discovered by the inquiry operation.

3. Once a remote device is identified as a PTP imaging device, the Bluetooth/PTP manager tries to create the L2CAP transport channels with the remote device.
4. If the bluetooth PTP manager successfully creates the Control/Data channel and the Event channel, then it will pass the associated LCIDs to the BPTP object dealing with the PTP, which will create a new imaging device in the local system
5. An optional notification to a registered PTP aware imaging application can also be performed.

Because the LCID numbers are allocated dynamically by L2CAP layer, the logical channels have to be open in sequence to distinguish between the Command/Data and Event transport channels. The Event transport channel shall be established before the Command/Data channel establishment is requested. It is important that the established sequence of channels to be used for the BPTP transport is known by both the *Initiator* and *Responder*. Thus if an attempt to initiate the Command/Data transport channels fails then the established Event transport channel should be closed. The *Initiator* device may re-try to establish the BPTP transport channels later (under some time constraints rules).

5.2.2 Configuration of Transport Channels

The L2CAP logical data channels used by BPTP should have their parameters configured as follows:

- Flush Timeout parameter should be set to 0xFFFF (infinite) value, unless the BPTP implementation supports transaction packet re-transmission so that channel reliability can be achieved.
- MTU parameter is implementation and device specific and can be set for each BPTP transport channel to different values. However, the minimum value should be at least 48 bytes.
- QoS parameter is implementation specific and the Quality of Service support in BPTP is optional.

5.2.3 Shutdown of Transport channels

It is the *Initiator*'s responsibility (BPTP layer) to close all the allocated communication channels whenever the remote device is not in the bluetooth range. The L2CAP channels will coexist during the presence of the *Responder* in the bluetooth neighbourhood. A communication failure with a *Responder*, resulting in a session communication failure, will generate the shutdown of the communication channels.

5.3 BPTP Packet Format

Beside the typical PTP data structures, a new type of packet is being defined: flow control packet. It is used as mechanism to synchronise the data transfer between the BPTP layers, in order to avoid the blocking of L2CAP layer with BPTP packets in the case that one of the layers is not receiving/transmitting properly.

Field	Size (bytes)	Data Type	Description
Packet Type	1	UINT8	One of the values: • 0x00 - Invalid Value; • 0x01 - Operation Request Packet • 0x02 - Operation Response Packet • 0x03 - Data Packet) • 0x04 - Flow Control Packet • 0x05 to 0xFF - Reserved
SessionID	4	UINT32	• 0x00000000 – valid only for OpenSession and GetDeviceInfo transactions • 0x00000000 to 0xFFFFFFFE -valid Session IDs • 0xFFFFFFFF – reserved
Length	2	UINT16	The size of the payload contained in the packet
Paylod	?	?	Dependent on the Packet Type

Table 6: BPTP Packet Format

6 TCP/IP Transport

This section describes existing issues of implementing PTP over TCP/IP and possible solutions. The ideal layer for mapping the PTP over TCP/IP protocols stack would be TCP. That is because the stream sockets provide a reliable way of communication between two networking devices (i.e. the *Initiator* and *Responder*). Moreover, the possibility to open multiple concurrent sockets to the same device would provide the perfect support for Command/Data/Response and Event PTP communication primitives.

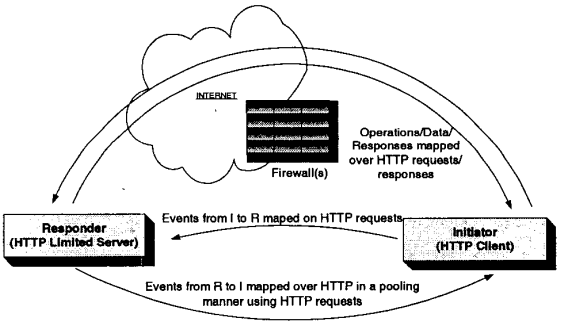


Fig 7: PTP over HTTP

However, in a real world situation, most of the TCP/IP ports are closed by different forms of security measures. e.g. firewalls, or different constrains caused by availability of reduced number of routable IP's (proxies). Therefore pure TCP/IP mapping for the PTP will work only in network setups where both the *Initiator* and *Responder* will have routable IP's and will not be separated by any firewall.

Usually, a firewall allows only traffic for known communication protocols (i.e. known ports are left open), the most common being FTP, HTTP and SSH. ***Fig 7*** presents briefly the idea of mapping the PTP over the

HTTP protocol. Using HTTP protocol is very attractive from two points of view: the traffic will go through any firewall out there and the *Initiator* won't have to have a routable IP address. The only requirement is that the *Responder* should have a routable IP address and implement mini-HTTP server functionality (support for the basic POST and GET functions from the HTTP 1.1 protocol specifications).

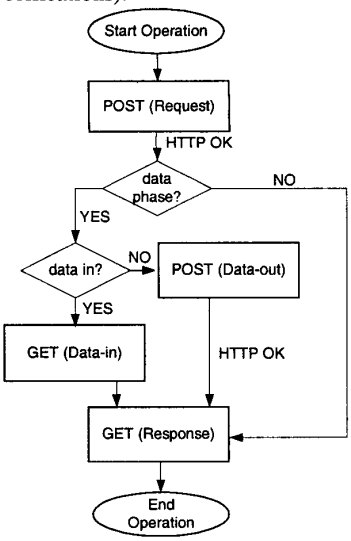


Fig 8: PTP Transaction over HTTP

The PTP operation request dataset structure will be transported from the *Initiator* to the *Responder* using a standard POST HTTP method. The HTTP OK response (200) will trigger on the *Initiator* to perform the remaining phases to complete the transaction. ***Fig 8*** describes the proposed PTP transaction mapping over HTTP. The events from the *Initiator* to the *Responder* will be carried out by a HTTP POST method, while the Events from the *Responder* to the *Initiator* will be carried over by the HTTP GET method, which will be called by the *Initiator* in a pooling fashion (i.e. from an Event thread). The *Responder* will need to export transactions URI that will be used by the *Initiator* to make the differentiation between the synchronous transactions and asynchronous events). A future paper will describe in detail the TCPPTP transport protocol (see ***Fig 9***) and the way this protocol will encapsulate the specific PTP structures and data flow.

Despite the fact that the TCP/IP transport for PTP opens a number of nice possibilities and new usage scenarios for an imaging device, the PnP features of such an approach are lost. The need for a TCP/IP Imaging Manager emerges, a few of its roles being: remote device URI specification (or IP), device discovery, authentication, the creation of TCPPTP layer and remote device presentation in the local system as an imaging device. Optionally, the TCP/IP Imaging manager can notify an imaging application about the presence of an imaging device in the local system.

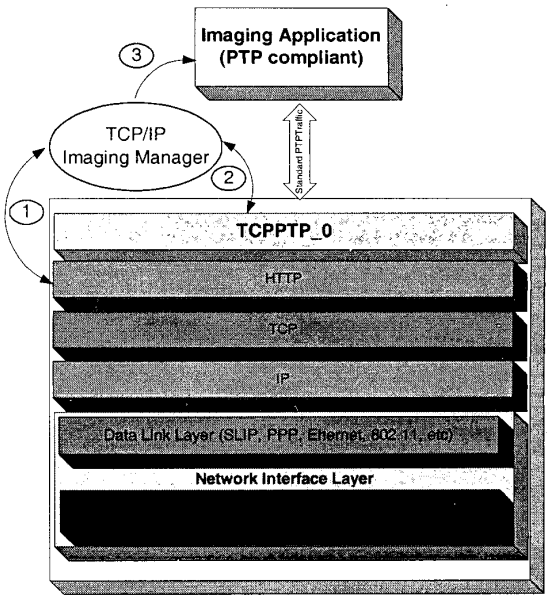


Fig 9: PTP Layer Protocol in the TCP/IP Protocol Stack

7 Conclusions

The PTP seems to become the imaging transfer protocol of choice for many digital camera manufacturers. As we tried to convey in this paper, PTP is an extremely flexible imaging protocol, designed to interconnect imaging devices. Despite being a powerful and complete protocol, it still lacks certain features for new emerging devices (such as digital camera with streaming of video and audio capabilities). This section aims to identify potential problems that the PTP might have and proposes quick fixes to those problems.

7.1 High Capacity Storage Problems

Today the storage capacity of digital media is increasing at vertiginous rates. Digital media of 1GB capacity is a reality; that is the storage that can easily accommodate thousands of pictures. One of the few limitations of the PTP, as it is specified today, is that is not allowing partial browsing for the object handles from the same category (i.e. objects handle for JPG file types). This can result in long initialization times or even long update times after storage changes in poorly implemented *Initiators*.

A quick fix to this problem could be window based browsing of object handles from the same category (i.e. request of the first say 10 object handles and then the next 10 and so on).

7.2 Multiple Session Problems

As it is, the PTP standard says that it is possible for *Responder* to support multiple sessions at the same time though it is not required to support more than one. On the other hand one of PTP goals is that it is very simple to implement even in an embedded environment with limited resources. The multi-session approach can require more processing power and resources, such as a multitasking environment. So the problem is that even the *Responder* supports multi-session (and in some cases it can be useful) the generic PTP *Initiator* cannot assume this in the case when multiple applications will access the camera resources at the same time. This problem is solved very well by serializing transactions by a PTP manager at the *Initiator*.

In order for the serialization not to generate long response time delays, the PTP manager should use the *GetPartialObject* method to download the picture from the *Responder*. As a suggestion to a standard extension we would recommend the implementation of *SendPartialObject* as a standard operation as well.

As it is now, in a multi-session implementation of the PTP, the *Initiator* tries to open a session with the *Responder* by specifying the SessionID in the operation request. The responder will answer OK if the specified SessionID is free. Otherwise, the *Responder* will answer Session_Already_Open or Device_Bussy if no session is available. The *Initiator* has to make a number of OpenSession requests until will find a free session. This could be time consuming when the remote device is not directly attached to the *Initiator* and a different transport than USB is used.

The alternative to this problem would be for the *Responder* to return in the response, as one of the response's parameter, an alternative SessionID (the first available), or Device_Bussy if no session is available.

7.3 Incorrect Utilization of Error Codes

This is more of an implementation problem rather than a problem of the protocol itself. Many implementers of the PTP incorrectly interpret the meanings of different error codes returned by the imaging devices. E.g. *GetObject*, (Association Object) operation returns *Invalid_ObjectHandle* instead of *Invalid_Parameter* (since the association object doesn't have other associated data then the object info).

A solution to these types of problems would be a revision of the protocol specifications, where such cases should be described in more details.

7.4 Normal *Responder* Operation: Invalid States

A good example of that kind of problems is where an *Initiator* opens a session with a given *Responder*, does some data exchange and exits without closing the session. In this case, a CloseSession after a Session_Already_Open error will result in a failure since the *Responder* will expect the TransactionID in sequence, but the new *Initiator* will not have the expected TransactionID. A quick fix for this situation would be to specify that the *Responder* will accept *CloseSession* with a TransactionID=0x00000000.

7.5 Handling of Special Value Parameters

A very common case in existing implementations is where the *Responder* reports that it implements a certain parameter, but it doesn't handle the special cases for that particular parameter properly.

A very good example is the *GetObjectHandle* operation that is accepting as second optional parameter the object format code (that specifies the type of object the operation should return handles for). The *Responder* should either return *Specification_By_Format_Unsuported* error code or implement the full functionality. This parameter has a specified special value of 0xFFFFFFFF that, if issued, the *Responder* should return object handles for all image objects. In reality, most of the *Responders* will return all the object handles ignoring the parameter value.

A fix for this problem would be to make impossible the partial implementation of given parameters specification. In this case, the object format code parameter would be either fully supported or unsupported.

7.6 Future Work

As future work, the authors intend to implement the Picture Transfer Protocol over Bluetooth transport and verify that the protocol is fully suited for wireless types of transports.

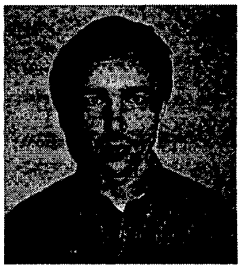
8 References

PIMA 15740, Photographic and Imaging Manufacturers Association Inc., 2000.

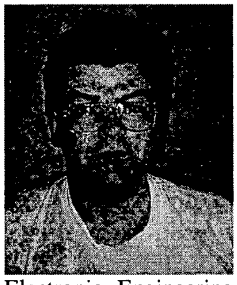
USB Still Image Capture Device Definition, USB Device Working Group, 2000.

Bluetooth Specification Version 1.1, Bluetooth SIG, February 2001

9 Author Biographies



Petronel Bigioi received his B.S. degree in Electronic Engineering from "Transilvania" University from Brasov, Romania, in 1997. At the same university he received in 1998 M.S. degree in Electronic Design Automation. He received a M.S. degree in electronic engineering at National University of Ireland, Galway in 2000. Currently he is lecturing embedded systems at National University of Ireland, Galway. His research interests include VLSI design, communication network protocols and embedded systems.



Peter Corcoran received the BAI (Electronic Engineering) and BA (Math's) degrees from Trinity College Dublin in 1984. He continued his studies at TCD and was awarded a Ph.D. in Electronic Engineering for research work in the field of Dielectric Liquids. In 1986 he was appointed to a lectureship in Electronic Engineering at UCG. His research interests include microprocessor applications, environmental monitoring technologies. He is a member of I.E.E.E.

George Susanu received his BS degree in microelectronics from "*Kishinev Politechnical Institute*", Kishinev, Republic of Moldova. With an experience of eight years in RTOS and embedded systems, having a wide experience in C/C++ programming he currently is working as R&D senior engineer with Accapella Ireland Ltd. His research areas include real time operating systems and device connectivity.

Irina Mocanu received her B.S. degree in Computer Science from University Politehnica of Bucharest, Romania, in 1996. At the same university she received in 1997 M.S. degree in Real Time Operating Systems for embedded systems. Currently she is teaching assistant at University Politehnica of Bucharest, Romania. She is currently teaching data structures and algorithms, computer graphics and formal languages. Her research interests include multimedia, databases and computer graphics.