Device Communications Interface™ (DCI) Protocol Specification

Advanced Software Radio™

Locked on Location

http://www.loctronix.com

MANAGED DOCUMENT	
Document Version:	1.0
Filename:	Device Communications Interface Protocol Sp
Last Revised	3-Dec-13

Document Control

Revision History

Version	Date	Description	Author
0.50	*7/20/2007*	Initial specification draft	M. Mathews B. Williams
1.00	*3/12/13*	Restructured and eliminated previous standard messages in favor of new WCA messages to reduce scope of support for basic I/O. Removed all custom device messages. Custom messages to be specified in separate specification documents. These modifications are not compatible with previous versions of DCI.	M. Mathews
1.00	*3/28/13*	Added flags field to Type Data Query (20,93). Added binary image specification to BIT Query (21,83) and additional flag to support listing of available images.	M. Mathews
1.00	*4/19/13*	Fixed bug in spec for Typed Data 20,13. Added endian message (20,06) to allow protocol to support little endian format. Default is still big-endian.	M. Mathews
1.00	*4/25/13*	Added idComponent to 21,04 message to create consistency in all WCA messages.	M. Mathews.
1.00	*6/6/13*	Added BIT status response (21,06) message to better coordinate transfers	M. Mathews
1.00	*9/23/13*	Updated WcaPropQuery (21,81) message to include specifications of type identifiers.	M. Mathews.
1.00	*12/3/13*	Added BIT Transfer cancel status indicator to enable peer to stop when another cancels.	

.

Contents

1. Introduction

This document details Loctronix SCP Device Communications Interface (DCI) protocol used for communicating with SCP enabled devices. DCI is a lightweight message based communications interface intended for control and data transfer of information. Protocol may be extended and adapted for other devices and can be used by partners in development of their applications.

1.1 Intended Use

The DCI protocol is the proprietary property of Loctronix Corporation. Any unauthorized use of the protocol is strictly prohibited. The specification is managed by Loctronix and may change without notice. Every attempt will be made to maintain backward compatibility.

2. DCI Message Structure

SCP enabled devices must communicate with networked receivers either processing commands or transferring data. Typically, the networks will be a TCP/IP based implementation, where transport level framing is handled by the communications stack. The base implementation of DCI is meant as a simple atomic messaging protocol supporting multiple conversations with a device or host to ensure coordinated communications.

The protocol is designed to minimize coding and data processing resources. At the heart of the protocol is the DCI message, which comprises a header and message payload. The size and structure of the header is dependent upon the particular version of the protocol.

DCI Message headers and encodings are in big endian format. Payloads may have big- or little-endian formats as agreed by the source and sink. By default big-endian is the format for all payload data. To specify little endian format, the protocol implementation may require notification using the framework endian message. Loctronix C++ and C# libraries support both payload endian formats. Loctronix compact embedded C libraries support the native processor endian format only.

2.1 Version 1 Message Structure

Version 1 is the basic implementation of the DCI protocol, it uses a 4 byte header to describe the payload contents.

1 byte	1 byte	2 bytes	N bytes
Flags	**Seq**	**Id**	**Payload**

Where:

Flags = Packet identification flags

Seq = Send / Acknowledge sequence fields

Id = Message identifier MSB is message category. LSB is the specific message type.

Payload = Message Payload: N Bytes size is based on particular message definition.

2.1.1 Flags

Flags identify protocol message structure and sender intentions.

3 bits	1 bit	2 bits	2 bits
Ver	**Ack**	**Encoding**	**ConvID**

Where:

Ver = Protocol version number. Value should be 1.

Ack **=** Sender request acknowledgement of message. If 0, message acknowledgement is not required. If 1, message must be acknowledged either by **ackid** in a response message or explicit transmission of the Ack message.

Encoding = Payload encoding method, 0 if no encoding is used. See Section 3 for details on encoding payload data.

ConvID = Conversation Identifier (0, 1, 2, or 3). Used to indicate specific conversations. If not using conversation capability, value should be 0.

2.1.2 Seq definition

The **Seq** field is a single byte containing the four-bit send sequence number and the four-bit acknowledge sequence number. The bit definitions are as follows:

4 bits	4 bits
seqid	**Ackid**

The **seqid** field is a four-bit sequence number assigned to a message being sent. This number is used by the message recipient for two purposes:

1. Recognize a new message. If the sequence ID is the same as the previous it is a retransmission

2. Generate a new **ackid** number to send out as acknowledgment of receiving a valid packet.

The **ackid** field is a four-bit sequence number equal to the **seqid** number of the most recent, valid received message. The **ackid** number is used to acknowledge successful reception of a message.

When a connection is first established between two recipients, the sequence numbers are both reset to zero. This provides positive acknowledgment when the first message is transmitted from either party. The sequence number for a new message is always incremented, thus the first message transmitted will have a send sequence number of one. This message will be transmitted continuously until a corresponding **ackid** sequence number is received.

Note 1: that with up to 16 sequence identifiers. Multiple messages can be sent all at once without having to wait one at a time. This can make more efficient use of available communications.

2.1.3 ID

The message identifier is a two byte value. The first or most significant byte is referred to as the category or **primary type**. All messages must have a primary type. The second byte is the message type or **secondary type**. Generally, the primary type is assigned to a category of message types (such as “Configuration” messages), and the secondary type is used to define which parameter within this category is being set, queried, or reported.

Note that 0xA0, and 0xB0 category or primary types are reserved for transport framing schemes.

2.1.4 Example

The following message shows the Version 1 header bytes with an Idle message in hex format:

```
20 56 01 00
```

The message indicates a version 1 message with **seqid =** 5 and **ackid** = 6. Message category is 01, 00 meaning idle message.

2.2 Payload Encoding

In the version 1 DCI format, up to four different payload encoding schemes can be supported. Defined as follows:

Encoding ID	Description
0	No Encoding (Default). By default, encoding ID is 0 for an unencoded format. This is the actual binary image.
1	Binary serial encoding. This is a encoding that ensures the payload does not contain the start of sequence identifiers within a binary stream. This will ensure no possibility of framing errors. This encoding may only be needed for variable payloads and can be used as needed.
2	Not Defined
3	Not Defined

2.2.1 Binary Serial Encoding (id=1)

The start of sequence identifier is 0xA0, 0xA4 when encountered in the unencoded payload, the two byte characters are replaced by 0xA0, 0x00, 0xA4. If 0xA0, 0x00 is encountered in the unencoded payload it is expanded into 0xA0, 0x00, 0x00. The following C code demonstrates how to encode and decode a payload for binary serial encoding.

```
void EncodePayload( byte* pOut*, byte* pbuf, short nLen )
{
    for (byte* pend = pbuf+nLen; pbuf < pend; pbuf++, pOut++)
     {
        *pOut = *pbuf;
        if( *pbuf == 0xA0 &&( pbuf[1] == 0x04 || pbuf[1] == 0x00))
        {
            pOut++;
            *pOut = 0x00;
        }
    }
}

void DecodePayload( byte* pOut*, byte* pbuf, short nLen )
{
    for (byte* pend = pbuf+nLen; pbuf < pend; pbuf++, pOut++)
     {
        *pOut = *pbuf;
        if( *pbuf == 0xA0 &&( pbuf[1] == 0x00))
        {
            pbuf++;
        }
    }
}
```

3. Transport Conventions

When binary streams are used transport DCI messages, the following transport framing shall be used. This framing is designed to ensure data accuracy and minimize processor loading while making it relative easy to identify message boundaries. DCI provides for future encoding methods to support specific optimizations and security.

3.1 Serial Stream Binary Transport

Binary serial streams with little or no infrastructure to ensure the quality of the message require add additional bytes at the beginning and end of the message for framing and synchronization of the data. The following conventions provide a standard convention for framing DCI messages within the stream. Serial streams are typically used with low-level stream such as RS-232 or similar. The Start Sequence identifier (0xA0, 0xA4) is reserved identifier within the DCI protocol denoting the framing wrapper and is discussed below.

DCI protocol, an additional 8 bytes are added to the standard DCI message to support binary framing and synchronization. The format of transport framing is shown below.

0xA0, 0xA4	2 bytes	4 bytes	N Bytes	2 Bytes	0xB0, 0xB2
Start Seq	**Length (15 bits)**	**Hdr**	**Encoded Payload**	**Checksum (15 bits)**	**End Seq**

A four byte header is added before the standard DCI message, which contains a unique start sequence identifier and a two byte payload length value. Four bytes are added to the end of the message comprising a 2-byte checksum and 2-byte end sequence specifier.

Where:

Start Seq = Packet identification flags

Start Seq = Start of Sequence specifiers: 0xA0, 0xA4

Length = Length of the encoded payload. This is the length of the payload in bytes after encoding. **NOTE length must be encoded in big endian format.**

Hdr = Message Payload: N Bytes size is based on particular message definition.

Payload = Encoded Payload. Encoding specified in the flags of the header. See Section 2.2 for options on payload encoding methods. Recommend using encoding type 1.

Checksum = Header / Payload Checksum See Section 3.1.1 for details on calculating checksum.

End Seq = End of Sequence specifiers: 0xB, 0x0B2

3.1.1 Checksum

The checksum is transmitted in Big Endian order. This is a 15 bit checksum of the 4 byte Hdr and the **Encoded Payload** together. The following C code demonstrates how to calculate the checksum.

```
short CalcChecksum( byte* pHdr*, byte* pPayload, short nLen )
{
    Byte* pend;
    int chksum = 0;

   //Calculate Header Checksum.
   for (pend = pHdr+4; pHdr < pend; pHdr++)
    {
      chksum += (*pbuf);
   }

   //Calculate Encoded Payload Checksum
    for (byte* pend = pbuf+nLen; pbuf < pend; pbuf++)
    {   chksum += *pbuf; }

    return (short) (chksum & 0x7FFF);
}
```

3.2 TCP/IP Transports

The following conventions are to be used when sending and receiving DCI data via TCP/IP Sockets.

- **DCI messages sent/received via TCP Socket transport** – the message length shall be prefix to the message so the socket processing engine can properly frame the message. The length prefix shall be a 16 bit signed integer with a maximum length of 32,768 bytes.
- **DCI message sent/received via UDP Socket transport** – the message shall be sent in its entirety with no message length prefix. UDP messages are guaranteed to be atomic so each UDP message will contain exactly one DCI message. However, be advised using the UDP transport, delivery is not guaranteed.

4. Specification Conventions

The following messages represent data that is sent to or received from the device by a controlling host program using DCI protocol.

According to its usage, DCI messages are classified as "command", "response" and "query". A command message is mainly used by a computer or server running a host program to set configurations and parameters or send instructions to the device. A command message usually has a response message from the device. A response message is from the device or host in response to a query or command message. Some DCI messages can be either command or response. In this case, the same DCI message types are used in both device configuration setting and report response.

Whenever requesting configuration or status from the device, usually a "Query" message is sent to the device. This query command is made up of the same primary and secondary types as the associated response, except that the secondary type will have the most significant bit set. For example, sending the command 20, 85 will result in the device responding with the message 20, 05.

In the definitions below, all DCI messages and examples are shown in hex. Also, the index numbers given for the packet formats are byte (8-bit) indices. Besides, a C Type structure is given in each DCI message to facilitate application developers.

Unless noted otherwise, when any data is sent as more than 1 byte the most significant byte is always sent first.

5. Infrastructure Messages

To facilitate common coordination and control of SCP throughout the network, DCI defines standard infrastructure messages that are useable by both hosts and devices.

5.1 Command / Response Idle (01, 00)

Idle message sent periodically depending upon communication timing requirements. This is just to let host and devices know that the connection is still active.

The idle message is also sent when no other messages are available to provide sequence acknowledgement. The message has no response data. The message can be used either as a command or response.

5.1.1 Message ID

Category	Type
01	00

5.1.2 C/C++ Structure

```
struct {
   }
```

5.1.3 Message Format

Index	0	1
Field	**01**	**00**

Field	Description	Type	Value	Index
N/A				

5.1.4 Example

```
01 00
```

This message carries no information.

5.2 Identify (02, 00), Identify Query(02, 80)

This message provides basic device identification. For systems providing transport level identification, this message provides a means to associate the transport level identifier with a known device identifier.

The Identify Query (02, 80) message requests the identification (02,00) message.

5.2.1 Message ID

Category	Type
02	00

5.2.2 C/C++ Structure

```
struct {
   char[32] szId;
   char[32] szSN;
   char[32] szMdl;
   };
```

5.2.3 Message Format

Index	0	1	2-33	34-65	66-97
Field	**02**	**00**	**szId**	**szSN**	**szMdl**

Field	Description	Type	Value	Index
szId	ASCII device identifier	char		2-33
szSN	ASCII device serial number	char		34-65
szMdl	ASCII device model number	char		66-97

NOTE if fields are not present then the message is shortened by the static length of the field. The indexes should also adjust accordingly.

5.2.4 Example

Host> 0281 – Queries for the device identification information.

Device> 0200[ASCII device identifier][ASCII device serial number][ASCII device model]

5.3 Unrecognized Message (03, 01)

Message sent from the host or device when received message is unrecognized. Either the category or message type was not recognized. Data payload contains unrecognized information.

5.3.1 Message ID

Category	Type
03	01

5.3.2 C/C++ Structure

```
struct {
    byte UnrecognizedMsgCategory;
    byte UnrecognizedMsgType;
    }
```

5.3.3 Message Format

Index	0	1	2	3
Field	**03**	**01**	**ucat**	**utype**

Field	Description	Type	Value	Index
ucat	Unrecognized Message Category	byte		2
utype	Unrecognized Message Type	byte		3

5.3.4 Example

```
03 01 28 4A
```

This message notifies host or device that message (28, 4A) was not recognized.

5.4 Transport Failure (03, 02)

Message sent from the host or device when received transport of messages failed in some manner. This can happen when a message is lost or had a bad checksum and was not retransmitted.

5.4.1 Message ID

Category	Type
03	02

5.4.2 C/C++ Structure

```
struct {
    byte flags;  // Transport error failure status flags
    byte hdr[4]; // Failed message header.
    }
```

5.4.3 Message Format

Index	0	1	2	3-6
Field	**03**	**02**	**flags**	**hdr**

Field	Description	Type	Value	Index
flags	Transport Error Flags Bits / Value 0-3 Failure Type: 0 = Bad Checksum, 1 = Out of Order or missing messages. 2 = Invalid conversation 3 = Seq Overflow. 4-7 *reserved*	byte		2
hdr	DCI Message Header resulting in failure.	byte		3-6

5.4.4 Example

```
03 02 01 35 31 20 13
```

This message notifies host or device that message identified by 35 31 20 13 typed data was out of order or previous message was missing.

5.5 Debug Message (05, 01)

Device debug text messages containing debug information. Debug messages provide a means to understand internals of device processing. Debug messages can be turned on and off.

5.5.1 Message ID

Category	Type
05	01

5.5.2 C/C++ Structure

```
struct {
    byte   status;
    short  src;
    short  len;
    char[len] szMsg;
    }
```

5.5.3 Message Format

Index	0	1	2	3-4	5-6	7-N

Field	**05**	**01**	**status**	**src**	**len**	**szMsg**

Field	Description	Type	Value	Index
status	Debug status type id: 3 = Debug Level 0 2 = Debug Level 1 (informational) 1 = Debug Level 2 (Warning) 0 = Debug Level 3 (Error)	byte		2
src	Software controller source id. Identifies the part of the controller software generating the debug information.	short		3-4
len	Length of text message	short		5-6
szMsg	Null terminated ASCII text message.	char*		7-N

5.5.4 Response Example

Respond with debug message from source 09 with a length of 32.

Characters are ASCII of 32.

```
05 01 00 00 09 20 32 32 ….32 00
```

6. Standard Messages (20)

To support generalized data processing and control of a variety of devices, standard messages are defined, which allow a host to configure and request information about the device. These messages apply to a variety of different devices including the sensors, beacons, and reference stations.

These messages are not required to be supported by a particular device. Consult the documentation regarding the specific device for a list of supported DCI messages.

6.1 Reset (20, 01)

Reset specified device.

6.1.1 Message ID

Category	Type
20	01

6.1.2 C/C++ Structure

```
struct {
    };
```

6.1.3 Message Format

Index	0	1
Field	**20**	**01**

Field	Description	Type	Value	Index
N/A				

6.1.4 Command Example

Reset the specified device:

```
20 01
```

This message carries no information.

6.2 Version Information (20, 02) / Version Information Query (20,82)

6.2.1 Message ID

Category	Type
20	02

6.2.2 C/C++ Structure

```
struct {
    byte   major;
    byte   minor;
```

```
            byte   maint;
            uint16  rev;
        }
```

6.2.3 Message Format

Index	0	1	2	3	4	5-6
Field	**20**	**02**	**major**	**minor**	**maint**	**rev**

Field	Description	Type	Value	Index
major	Major version number	byte		2
minor	Minor version number	byte		3
maint	Maintenance version number	byte		4
rev	Build revision number	byte		5-6

6.2.4 Example

Issue a (20,82) Version Information Query message. The returned message for version 1.02.03 build 175 message payload is

```
20 02 01 02 03 00 AF
```

6.2.5 Querying Version information

Send a (20,82) message with no payload to get the version information.

6.3 Command Synchronize Time (20, 87)

Message sent from the host to command device to synchronize time with the specified host. This is a DCI protocol method to enable time synchronization of a device, using a statistical method over the network.

6.3.1 Message ID

Category	Type
04	80

6.3.2 C/C++ Structure

```
struct {
    byte   ctObservations;
    byte[4] byteServerAddres;
    short   idPort;
    byte    flags
    }
```

6.3.3 Message Format

Index	0	1	2	3-6	7-8	9
Field	**04**	**80**	**ctobs**	**addr**	**port**	**flags**

Field	Description	Type	Value	Index
Ctobs	Count of observations to make before calculating time synchronization. Number depends on how accurate and stable the network is. A good choice is between 3 and 5.	byte	5	2

Addr	mAddress of time server (host), interpretation depends upon flags, typically it is an IP address. If value is zero time server IP address is the default server IP address defined for the device.	byte[4]	3-6
Port	Port number of host to communicate on. Interpretation depends upon flags, typically it is an IP address.	short	7-8
Flags	Control flags specify how synchronization is performed. Bits / Value 0 — Synch Method: 0 = DCI synch, 1 = nettime (linux/windows) 1-2 — Address type: 0 : TCIP / Port Addressing 3 — Measure offset only, don't correct device time. 4-7 — *reserved*	byte	9

6.3.4 Example

Command device to time synch using DCI synch method with server 192.168.191.23 port 3200.

```
04 80 05 C0 A8 BF 17 0C 80 00
```

6.4 Response Time Info (20, 08)

Message sent from device or host in response to query time information. Responding machine returns the time it sent the message. For relatively low latency networks, this message can be used as part of a time transfer mechanism. Repeated queries and responses can provide a means to determine network latency and thus transfer time from one machine to another.

Typical goal for this time synch is 10s of milliseconds. Units of time are typically in UTC.

Message sent from the host to command device to synchronize time with the specified host. This is a DCI protocol method to enable time synchronization of a device, using a statistical method over the network.

6.4.1 Message ID

Category	Type
04	01

6.4.2 C/C++ Structure

```
struct {
    short  yr;
    byte   mo;
    byte   day;
    byte   hr;
    byte   min;
    byte   sec;
    short  fracSecs;
    short  uncert;
}
```

6.4.3 Message Format

Index	0	1	2-3	4	5	6	7	8	9-10	11-12
Field	**04**	**01**	**Yr**	**mo**	**day**	**hr**	**min**	**sec**	**fsecs**	**uncert**

Field	Description	Type	Value	Index
Yr	Four digit year.	short		2-3
mo	Month: 1 – 12.	byte		4
day	Day of month: 1 – 31	byte		5
Hr	Hour of day: 0 – 23	byte		6
min	Minute of hour: 0 – 59	byte		7
sec	Seconds of minute: 0 – 59	byte		8
fsecs	Fractional seconds in units of 0.1 milliseconds. Range is between 0 and 999.9 milliseconds.	short		9-10
uncert	Uncertainty of time report in 0.1 milliseconds. This is typically reported when certainty of time is known. It is a quality of measure.	short		11-12

6.4.4 Example

Report UTC June 22, 2007 18:36:04.3321

```
04 01 07 D7 06 16 12 24 04 0C F9
```

6.5 Response Time Synchronization (20, 09)

Report is returned to host requesting time synchronization with the results of the time synchronization (0x480).

6.5.1 Message ID

Category	Type
04	01

6.5.2 C/C++ Structure

```
struct {
    short  tTransitTime;
    long   uncertTransitTime;
    short  tTimeOffset;
    long   uncertTimeOffset;
    short  ctObs
}
```

6.5.3 Message Format

Index	0	1	2-5	6-7	8-11	12-13	14-15
Field	**04**	**01**	**ttransit**	**uncerttransit**	**toffset**	**uncertoffset**	**ctobs**

Field	Description	Type	Value	Index
ttransit	One way transit time in 0.1 milliseconds.	long		2-5
uncerttransit	Uncertainty of transit time in 0.1 milliseconds	short		6-7
toffset	Offset between host server and device in 0.1 milliseconds	long		8-11
uncertoffset	Uncertainty of host/device offset in 0.1	short		12-13

	milliseconds		
ctobs	Count of observations used in making these estimates	short	14-15

6.5.4 Example

```
04 02 <TODO>
```

6.6 Query Time Info (20, 88)

Request device or host respond with a (04, 01) time information message.

6.6.1 Message ID

Category	Type
04	81

6.6.2 C/C++ Structure

```
struct {
   }
```

6.6.3 Message Format

Index	0	1
Field	**04**	**81**

Field	Description	Type	Value	Index
N/A				

6.6.4 Example

```
04 81
```

This message carries no information.

6.7 String Properties(20, 05)

Message sends receives categorized Device configuration settings. This message uses a name/value mechanism to return named properties for the device. Proper interpretation is device specific. This is a generalized method to send and receive multiple properties.

When sending configuration settings, send only those properties that are to be modified.

6.7.1 Message ID

Category	Type
21	05

6.7.2 C/C++ Structure

```
struct {
   ushort flags;
   ushort len
   char[len] settings
   };
```

6.7.3 Message Format

Index	0	1	2-3	4-5	6-len-1
Field	**21**	**05**	flags	len	settings

6.7.4

Field	Description	Type	Value	Index
flags	Configuration settings flags. Device / device class specific flags control how settings are interpreted. No currently used.	ushort		2-3
len	Length of settings block	ushort		4-5
settings	ASCII array of settings organized into name value pairs. Structure is bar delimited. Fore example: <name1>|<value1>|<name2>|<value2> Length of settings is determined by the len value. All settings are communicated in ASCII string format.	byte		35

6.7.5 Command / Response Example

TODO

6.8 String Property Query (20, 85)

Request device respond with a (20, 05) device configuration settings message.

6.8.1 Message ID

Category	Type
20	85

6.8.2 C/C++ Structure

```
struct {
   };
```

6.8.3 Message Format

Index	0	1
Field	**20**	**85**

Field	Description	Type	Value	Index
N/A				

6.8.4 Query Example

For the specified device, query configuration:

```
21 85
```

This message carries no information.

6.9 EndianFormat (20,06) / EndianFormatQuery (20,86)

Message specifies Endian format, big or little, of payload data. For networked systems the default is big endian. Embedded devices may implement little endian format in order to reduce implementation size and overhead. Host devices may query the device for the endian format so that it can configure parsing routines appropriately. Loctronix C# and C++ libraries support either endian wire format for payload data.

6.9.1 Message ID

Category	Type
20	06

6.9.2 C/C++ Structure

```
struct {
    byte   flags;
    int16  testval;
    }
```

6.9.3 Message Format

Index	0	1	2	3-4
Field	**20**	**06**	**flags**	**testval**

Field	Description	Type	Value	Index
flags	Flags specify endian information <u>Bits</u> <u>Value</u> 0-1 Endian format: 0 = Unknown Endian Format, 1 = Payloads are BigEndian 2= Payloads are LittleEndian 3-7 *reserved*	byte		2
testval	Test value of 15 to determine endian format: 00 0F big endian or 0F 00 little endian format.	byte[4]	0Fh	3-6

6.9.4 Example

Query (20,86) with no payload to get endian format message. Message returns indicates little endian format with the test in little endian format as well.

```
04 06 01 0F 00
```

6.9.5 Querying Endian Information

Send a (20,86) message with no payload to get the endian information.

6.10 Typed Data (20,13)

Reports typed data records in a specific data format specified by the typeid field contained within the message. Provides one message for transferring data in multiple different formats.

6.10.1 Message ID

Category	Type
20	13

6.10.2 C/C++ Structure

```
struct {
```

```
        short  week;
        int32  secWhole;
        short  secFrac;
        uint32 timecode;
        uint16 typedid;
        uint16 flags;
        unit16 lenData;
        int16  Data[lenData];
    };
```

6.10.3 Message Format

Index	0	1	2-3	4-7	8-9	10-13	14-15	16-17	18-19	20- [20+lenData-1]
Field	**20**	**13**	week	secWhole	secFrac	timecode	typeid	flags	lenData	Data

Field	Description	Type	Value	Index
week	GPS week number	Short		2-3
secWhole	GPS seconds (whole seconds, in the week)	Long		4-7
secFrac	GPS fractional seconds in 0.1 ms units	Short		8-9
timecode	Unique counter value defining the data Record	uint32		10-13
typeid	Identifies the data class type. Use to determine what type of information is being looked at. A device can potentially provide multiple types of data records and this is used to distinguish between them.	uint16		14-15
flags	Flags definitions are specific to the type identifier. Bits Value 0-15 Reserved for typeid specification.	uint16		16-17
lenData	Length of data in bytes	uint16		18-19
Data	Serialized data in accordance to the specification of the typeid.	byte		20 +

6.11 Typed Data Query (20,93)

Request device respond with a (20, 13) typed data record with the specified typeid.

6.11.1 Message ID

Category	Type
20	93

6.11.2 C/C++ Structure

```
struct {
    uint16 typeid;
    uint16 flags;
    };
```

6.11.3 Message Format

Index	0	1	2-3	4-5
Field	**20**	**93**	**typeid**	**flags**

Field	Description	Type	Value	Index
typeid	TypeDataRecord type identifier being requested.	uint16		2-3
flags	Flags definitions are specific to the type identifier. Bits Value 0-15 Reserved for typeid specification.	uint16		4-5

6.11.4 Query Example

For the specified device, query the for typeid = 32 with no flags:

```
20 93 00 20 00 00
```

6.12 Log Typed Data (20, 14) – NEEDS UPDATE

Command / Response message for configuring session raw data logging to sensor data storage. Not all devices may support this capability. For devices that do support it, it enables the user to store raw data into local sensor storage which can be retrieved for post analysis.

Logging is not persisted between sessions and has a limit to maximum amount of memory logged to help prevent overrun of local device storage. If insufficient storage space exists, command will cause an error to be reported to the host prior to enabling logging.

6.12.1 Message ID

Category	Type
21	14

6.12.2 C/C++ Structure

```
struct {
    byte     flags
    char[64] szPrefix;
    long     lMaxBytes;
    long     lDuration;
    char[256] szDescr;
    char[64]  szSite;
    char[64]  szOperator;
};
```

6.12.3 Message Format

Index	0	1	2	3-66	67-70	71-74
Field	**21**	**14**	Flags	szPrefix	lMaxBytes	lDuration

Index	75-330	331-394	395- 458
Field	szDescr	szSite	szOperator

Field	Description	Type	Value	Index
Flags	Logging Configuration Flags Bits Value 0 Logging Enabled (1) / Disabled (0) 1 Auto Number File Enable 2 Overwrite Existing File Enable 3-7 Reserved	byte		2
szPrefix	File name prefix used in forming log files. This is a required field and must be specified. Function appends file number and .bds postfix file type.	Char[64]		3-66
lMaxBytes	Maximum number of bytes to log before suspending. A value of -1 will fill log until full. -1 is not recommended for normal use.	Long		67-70
lDuration	Maximum number of seconds to log raw data. A value of -1 means it will log indefinitely. Not the recommended method for standard usage.	Long		71-74
szDescr	Optional null terminated string providing an optional description of the archive log. This is embedded in the BDS header when the logging is enabled.	Char[256]		75-330
szSite	Optional null terminated string providing a means to specify the station the data was logged at.	Char[64]		331-394
szOperator	Optional null terminated string providing operator information	Char[64]		395-458

6.13 Log Typed Data Query (20, 94) – NEEDS UPDATE

Causes device to respond with a Response Log Raw Data Message(20,13);

6.13.1 Message ID

Category	Type
20	94

6.13.2 C/C++ Structure

```
struct {
    };
```

6.13.3 Message Format

Index	0	1
Field	**20**	**94**

Field	Description	Type	Value	Index
N/A				

6.13.4 Query Example

For the specified device, Query Raw Data Logging Status:

```
2094
```

This message carries no information.

7. WCA Messages (21)

All devices utilizing the WCA architecture for integrating can use these messages to support integration with framework. These messages are harmonized with the structures simplifying the tasks of configuration and reporting status information.

By convention all WCA components must have an identifier between 0 and 255 (FF hex). Identifiers between 128 and 255 (80 and FF hex) are reserved for fixed hardware components specific to a particular hardware implementation. By definition, component 0 (0 hex) is the default WCA hardware component corresponding to the programmable FPGA logic. Specialized implementations may specify other component identifiers as needed or warranted by the implementation.

7.1 Typed Properties (21,01)

Messages to send and receive typed property information for WCA component. This is a type specific version of string properties designed to aid in type specific property access.

Supported types are byte, uint16, int16, uint32, int32, float, and double.

See Query Typed Properties (21,81) for specification of querying specific property values.

7.1.1 Message ID

Category	Type
21	01

7.1.2 C/C++ Structure

```
//Defines a property value record.
struct propval_tag {
   byte idProp;
   byte typeProp;
   byte[8] propData;
};

enum PropTypesEnum {
PT_BYTE = 0,
PT_UINT16 = 1,
PT_INT16 = 2,
PT_UINT32 = 3,
PT_INT32 = 4,
PT_FLOAT = 5,
PT_DOUBLE  = 6
}
//Defines the typed property message payload with
//a variable number of property values contained.
struct {
   byte   idComponent;
   byte   ctProperties;
   struct propval_tag propVals[ctProperties];
}
```

7.1.3 Message Format

Index	0	1	2	3	4 – 10*ctProperties
Field	**21**	**01**	**idComponent**	**ctProperties**	**propdata**

Field	Description	Type	Value	Index
idComponent	WCA component numerical identifier.	byte		2
ctProperties	Number of properties specified in this message.	byte		3

propdata	Property data 10 bytes for each property specified. Where the fields of propval_tag are defined as:
	idProperty: 0 – 255 unique property id in the component.
	typeProp: type identifier for the property. Type identifiers are as follows:
	`PT_BYTE = 0,`
	`PT_UINT16 = 1,`
	`PT_INT16 = 2,`
	`PT_UINT32 = 3,`
	`PT_INT32 = 4,`
	`PT_FLOAT = 5,`
	`PT_DOUBLE = 6,`
	`PT_UINT64 = 7,`
	`PT_INT64 = 8`
	propData: eight (8) bytes of property data.
		byte		4-10*ctProperties

7.1.4 Command Example

TODO

7.1.5 Response Example

TODO

7.2 Typed Properties Query (21,81)

Message request Typed Data properties for the specified WCA component.

7.2.1 Message ID

Category	Type
21	81

7.2.2 C/C++ Structure

```
struct {
    byte   idComponent;
    byte   flags;
    byte   ctProps;          // Count properties requested.
    byte   propids[ctProps];
    byte   typeids[ctProps]; // Optionally defined
}
```

7.2.3 Message Format

Index	0	1	2	3	4	5 to 4+ctProps	5+ctProps to 2*ctProps+4
Field	**21**	**81**	**idComponent**	**flags**	**ctProps**	**propids**	**typeids**

Field	Description	Type	Value	Index
idComponent	WCA component numerical identifier.	byte		2
flags	Property query flags: Bits Value 0 types field / defined. Typically true for component properties that are implemented in non-fixed / reprogrammable components (e.g. FPGA).			3
ctProps	Number of properties specified in this message.	byte		4
propids	Array of property identifiers to return property information for.	byte		5 to 4+ctProps
typeids	Optional type identifiers for the property. Set the bit 0 flag when field is used. These type id's are needed for dynamic or reconfigurable types implemented in not fixed components. For simplicity they can always be implemented if bandwidth is not a consideration.	byte		5+ ctProps to 4 + 2*ctProps

7.2.4 Command Example

TODO

7.2.5 Response Example

TODO

7.3 Execute Action (21,02)

Message provides structure to command a WCA component to execute a particular action.

7.3.1 Message ID

Category	Type
21	02

7.3.2 C/C++ Structure

```
struct {
    byte   idComponent;
    byte   idAction;
    uint16 sizeData;
    byte   data[sizeData];
}
```

7.3.3 Message Format

Index	0	1	2	3	4	5 - sizeData

Field	**21**	**02**	**idComponent**	**idAction**	**sizeData**	**data**

Field	Description	Type	Value	Index
idComponent	WCA component numerical identifier.	byte		2
idAction	Action identifier.	byte		3
sizeData	Size of data for the action. Can be zero if not data provided.	uint16		sizeData
data	Action data.			5 – sizeData

7.4 Binary Image Transfer Info (21,03)

The message specifies binary image transfer information, which is sent prior to uploading or downloading binary image frames. It specifies information about the image, the transfer size and number of frames that will be transferred. The target will respond with a Status (21,06) indicating either initiating or unavailable.

7.4.1 Message ID

Category	Type
21	03

7.4.2 C/C++ Structure

```
struct {
    byte        idComponent;
    byte        flags;
    char[32]    szName;
    char[32]    szDescription;
    uint32      sizeImg;
    uint16      sizeFrame;
    uint32      ctFrames;
    byte        idTransfer;
}
```

7.4.3 Message Format

Index	0	1	2	3	4-35	36 – 67
Field	**21**	**03**	**idComponent**	**flags**	**szName**	**szDescr**

Index	68-71	72-73	74-77	78
Field	**sizeImg**	**sizeFrame**	**ctFrames**	**idTransfer**

Field	Description	Type	Value	Index
idComponent	Component numerical identifier.	byte		2
flags	Flags specify use and purpose of image. Values are: Bits Value 0-1 Lifecycle 0 = volatile (lost after reset), 1 = saved	byte		3

	2-7 *reserved*		
szName	Name of binary image. Null terminated string.	char[32]	4-35
szDescr	Description of binary image. Typically contains version information. Null terminated string.	char[32]	36-67
sizeImg	Size of image data.	uint32	68-71
sizeFrame	Size of image transfer message frame	uint16	72-73
ctFrames	Total count of images transfer frames that will be used in uploading or downloading image.	uint32	74-77
idTransfer	Transfer identifier, uniquely specifies the transfer set. Set to a unique index to distinguish between multiple simultaneous transfers. Set to zero if there is only one transfer at a time.	byte	78

7.5 Binary Image Transfer Frame (21,04)

This message is a data frame of a binary image transfer. A series of these frames are sent to transfer a binary image of data from the source to the target. A transfer id is provided to uniquely identify the transfer operation. The target will respond with a status (21,06) indicating the success of the frame transfer. Be sure to process the status message before proceeding to the next frame.

7.5.1 Message ID

Category	Type
21	04

7.5.1 C/C++ Structure

```
struct {
    byte        idComponent;
    byte        idTransfer;
    uint16      flags;
    uint32      idFrame;
    uint16      ctBytes;
    byte        data[sizeFrame];
}
```

7.5.2 Message Format

Index	0	1	2	3	4-5	4-7	8-9
Field	**21**	**03**	**idComponent**	**idTransfer**	**flags**	**idFrame**	**ctBytes**

Index	12 – (12 + sizeFrame)
Field	**data**

Field	Description	Type	Value	Index
idComponent	Component numerical identifier.	byte		2
idTransfer	Binary Image transfer operation	byte		3

	identifier specified in the Binary Image Info (21,03) message.		
flags	<reserved>	uint16	4-5
idFrame	Name of binary image. Null terminated string.	uint32	6-9
ctBytes	Number of data bytes that are valid in this frame. May be less than the frame size specified in the Binary Image Info (21,03) message if the last frame of data.	uint16	10-11
data	Size of image data.	byte[]	12 – (12 + sizeFrame)

7.6 Binary Image Transfer Status (21,06)

Reports Status of Binary Transfer operations, can be received on sending of info, queries, or frame data.

7.6.1 Message ID

Category	Type
21	06

7.6.2 C/C++ Structure

```
struct {
    byte         idComponent;
    byte         idTransfer;
    uint16       idFrame;
    uint32       ctTransferred;
    byte         idStatus;

};
```

7.6.3 Message Format

Index	0	1	2	3	4-5	6-9	10
Field	**21**	**83**	**idComponent**	**idTransfer**	**idFrame**	**ctTransferred**	**idStatus**

Field	Description	Type	Value	Index
idComponent	Component numerical identifier.	byte		2
idTransfer	Transfer identifier used to uniquely identify the image transfer operation.	byte		3
idFrame	Current transfer frame identifier	uint16		4-5
ctTransferred	Count of bytes transferred	uint32		6-9
idStatus	Status Flags defined as follows: ID Status Description 0 Initiating Transfer 1 Transfer Complete 2 Ready Next Frame 3 Frame Error 4 Write Error	byte		10

5	Read Error
6	Operation Not Available
7	Operation Cancelled

7.7 Binary Image Transfer Query (21,83)

Requested device responds with a Binary Image Transfer Information (21,03) and Binary Image Transfer Frames (21,04) if specified.

7.7.1 Message ID

Category	Type
21	83

7.7.2 C/C++ Structure

```
struct {
    byte        idComponent;
    byte        idTransfer;
    byte        flags;
    char[32]    szNname;
};
```

7.7.3 Message Format

Index	0	1	2	3	4	5-36
Field	**21**	**83**	**idComponent**	**idTransfer**	**flags**	**szName**

Field	Description	Type	Value	Index
idComponent	Component numerical identifier.	byte		2
idTransfer	Transfer identifier used to uniquely identify the image transfer operation.	byte		3
flags	Requests specific information Valid flags are: <u>Bits Value</u> 0-1 Transfer Action 0 = BIT Info and data. 1 = BIT Info only (no data) 2 = All BIT infos for component 2-7 *reserved*	byte		4
szName	Name of the binary image to transfer. This is a null terminated string. Maximum length is 32 characters. This field is ignored if Transfer Action = 2. Field may also be ignored if WCA component does not have multiple binary images. Set to zero if not used.	char		5-36

7.7.4 Query Example

The following query asks for Binary Image information only (no frame data) for idComponent = 12, idTransfer = 0.

```
21 83 12 00 00
```

7.8 Event Notification (21,05)

Asynchronous triggered event notification.

7.8.1 Message ID

Category	Type
21	05

7.8.2 C/C++ Structure

```
struct
{
    byte   idComponent;
    byte   idEvent;
    uint16 flags;
    uint32 info;
};
```

7.8.3 Message Format

Index	0	1	2	3	4-5	6-9
Field	**20**	**04**	**idComponent**	**idEvent**	**flags**	**info**

Field	Description	Type	Value	Index
idComponent	WCA Component identifier	byte		2
idEvent	Event identifier uniquely defines the event within the component. Components can have more than one event.	byte		3
flags	reserved	uint16		4-5
info	Event information. This data is specific to the event. Consult specific waveform and/or device WCA implementation documentation.	uint32		6-9