Angel

Debug Protocol

Document number: ARM DUI 0052B
Issued: May 1998

Open Access

Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Change Log

Issue	Date	By	Change
A	March 1998	KTB/RI-C	First release
B	May 1988	RI-C	Amendments following initial review.

Key

Code and other program texts are set in a `monospaced font`.

Contents

1 Introduction

The Angel Debug Protocol, (ADP) was designed to provide a reliable connection between a debug target and a host debugger during a debugging session. The protocol had to provide sufficient and flexible access to the target from the host, and be resilient. In addition, two constraints were placed on the target end of the connection; the target could not be assumed to have a timer, and the software resident on the target was of limited size. Finally, it was beneficial if the high level operations which the protocol implemented were similar to those of the previous RDP protocol, to facilitate the transition.

The protocol does not address issues of routing, as the data link layer is assumed to be implemented as a point-to-point link.

ADP implements the basic operations in typical client server fashion, using a request–response style typical of remote procedure call systems. Both sides can act as client or server, different data streams (channels) being used to distinguish these roles. The underlying layers implement packetization of requests onto identified channels, recovery from simple packet loss and a simple data link protocol suitable for use over serial and parallel links. A data link layer suitable for use over TCP/IP networks is supported, using the UDP protocol and connection addresses.

2 The Protocol Suite

The basic unit of communication in Angel is a packet. Within each packet received or transmitted by Angel, there are at least three levels of protocol. These are: Data Provider (higher level), Channel Layer and Device Layer (lower level)

The data provider protocol is an application-layer protocol; the channel layer is the transport protocol and the device level is the data link layer. The protocol is deliberately assymetric; Figure 1, below, shows the host end view, while Figure 2 shows the target view.

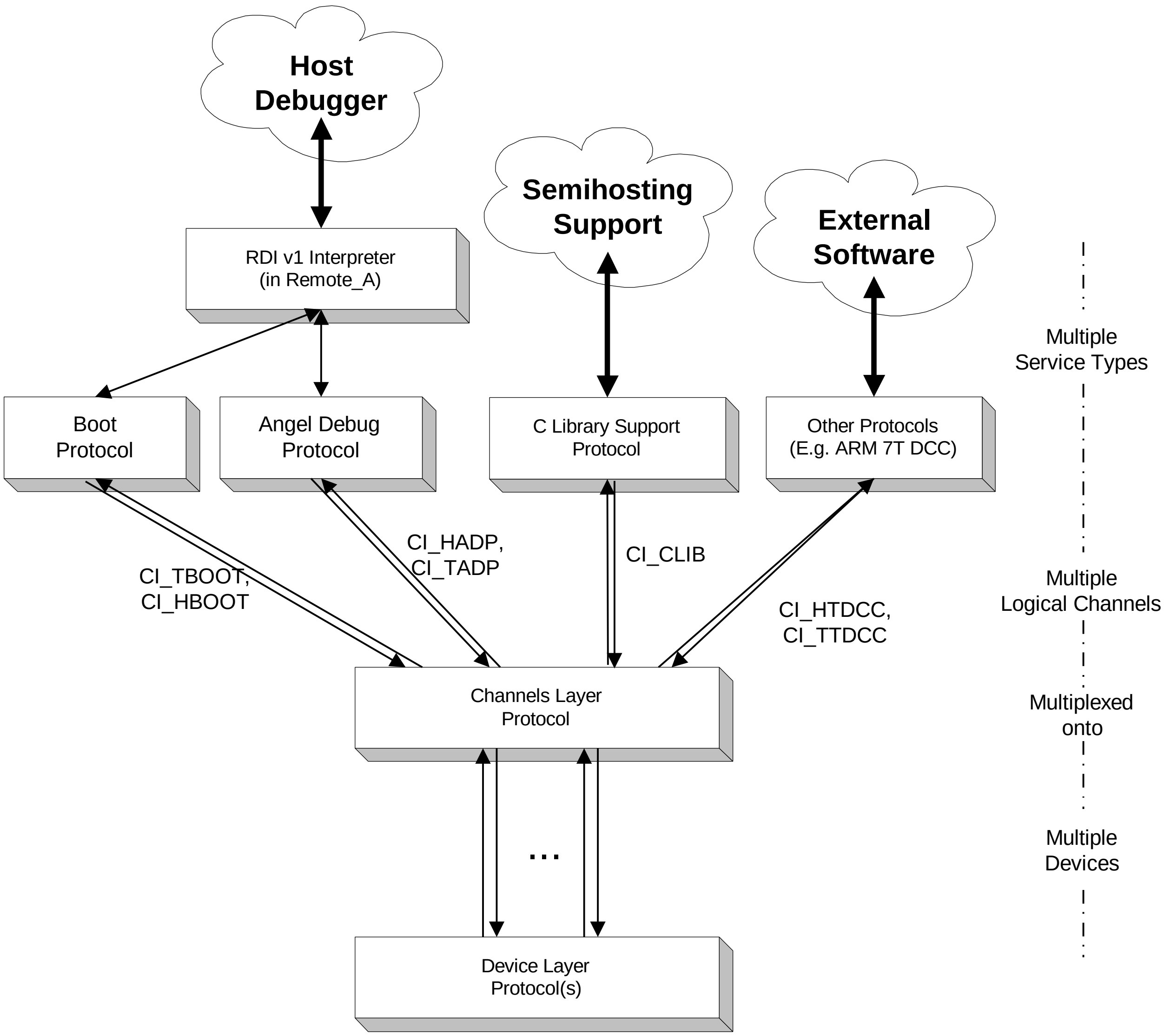

Figure 1: Overall protocol layering (host end)

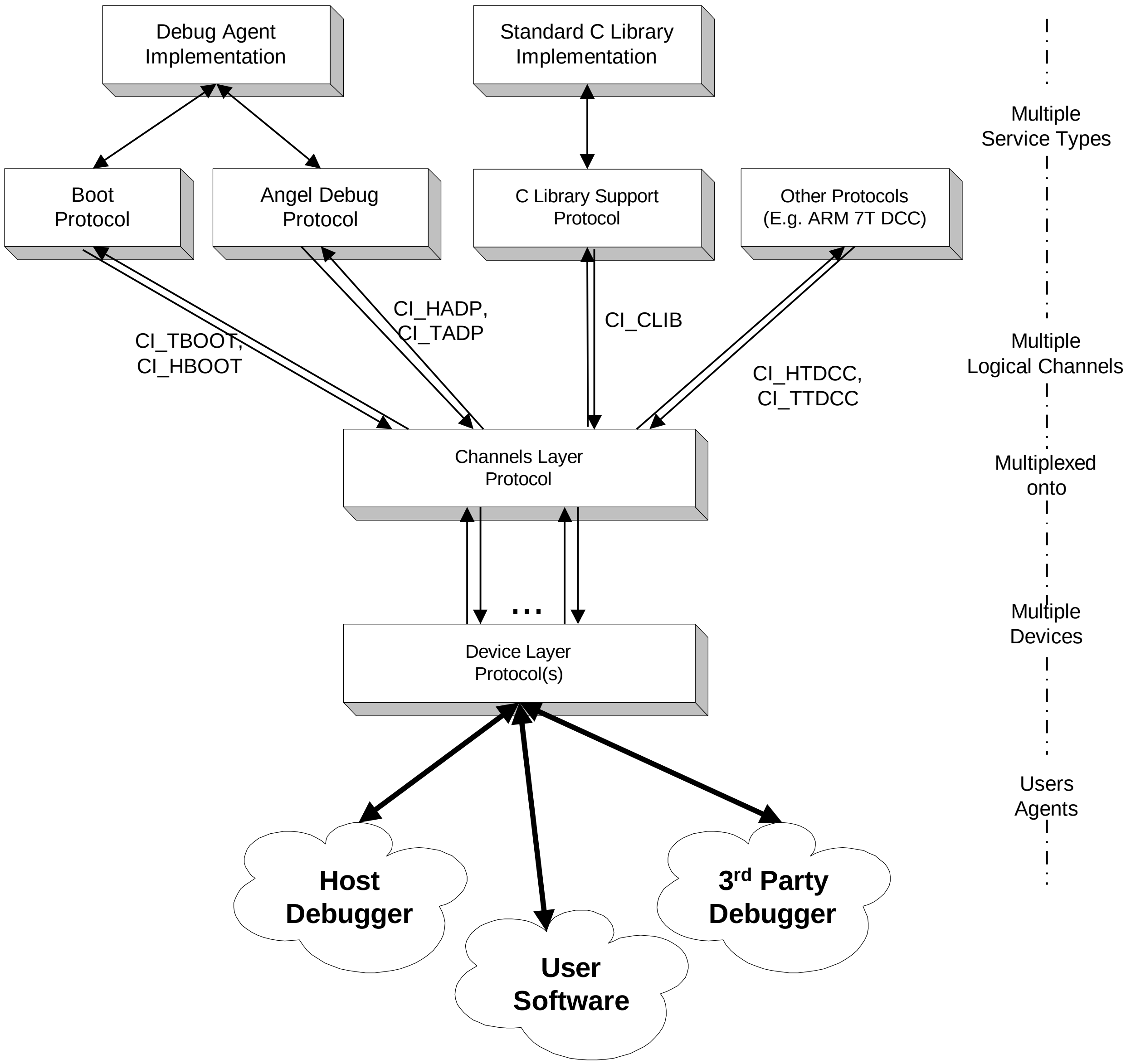

Figure 2: Overall protocol layering (target end)

2.1 Data Provider Level

The data provider level protocols are used by two different *services*, using different request and response formats conforming to a common base specification.

- Angel Debug Agents using ADP.
- Angel C semihosting support code using the C Library Support protocol.

The differentiation of which protocol is being used depends upon the channel number, in the same way that an FTP and a Telnet connection can be set up simultaneously over a TCP/IP link. It is important to note that other agents, including for example other debug agents, may use different data provider protocols over the same channel layer link.

For both ADP and the C Library support protocol, the packet formats are based on the same structure with slight differences in the reason code. This structure is:

- reason code
- information describing host debug world; private to host
- target OS information to identify process/thread world, etc. (target defined)
- data in a format defined by the message “reason” code.

Figures 3 and 4 below show these structures in detail:

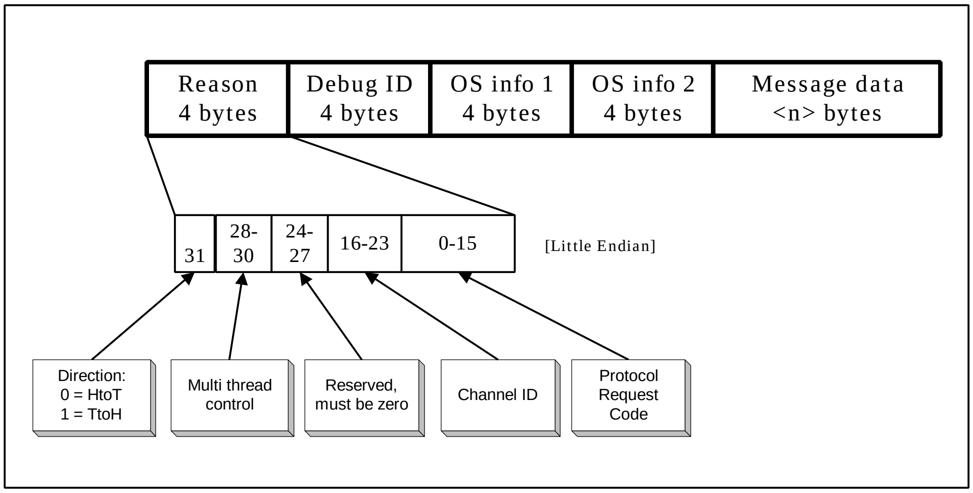

Figure 3: ADP protocol packet definition

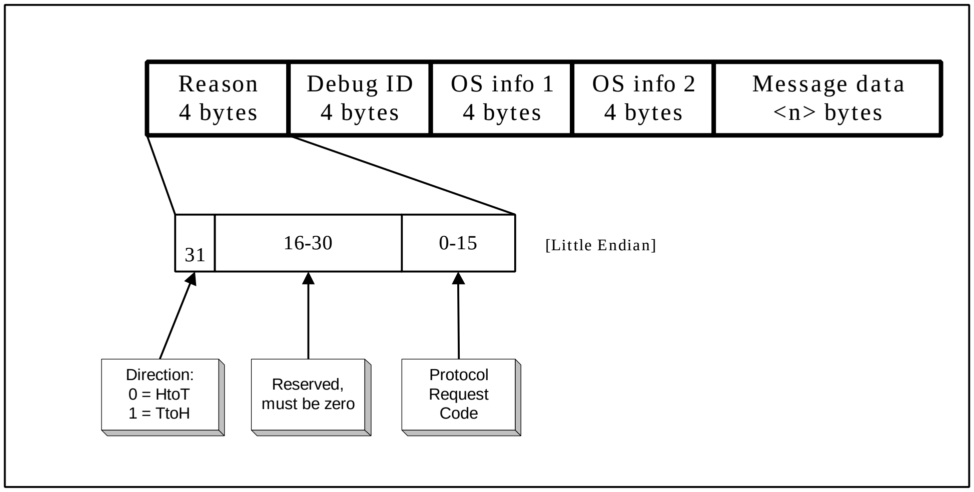

Figure 4: C support library protocol packet definition

2.1.1 Reason code

The reason code defines the operation being requested of the recipient, usually the target, and allows responses to be checked against requests; the response code in the reply to a message will differ from the request only in that the Direction bit will be flipped.

Bits 16-27 of this word contain the channel number of the channel over which this packet is travelling.

Note *This* also *applies to the subreason codes used in, eg. ADP_Info, and the ADP_Stopped reasons.*

The multi-thread bits are interpreted according to the following table:

Bit #	Name	Description
30	DisableFIQ	Disable FIQ whilst processing message.
29	DisableIRQ	Disable IRQ whilst processing message.
28	DisablePreemption	Disable O/S pre-emption whilst processing message.

These bits are used to control how the target system executes whilst processing messages. This allows for O/S specific host-based debug programs to interrogate system structures whilst ensuring that the access is atomic within the constraints imposed by the target O/S. They should be set to zero in messages sent from the target to the host.

2.1.2 Debug ID

The debug ID is a field provided for the use of the host software; the target guarantees that for a given request, the response to that request will have the same debug ID as the request.

For systems which have no need of this (such as single threaded debuggers) it should be set to 0xFFFFFFFF. Messages originated by the target (eg. the boot message) also set this field to 0xFFFFFFFF.

2.1.3 OSInfo 1, OSInfo 2

These fields can be used by multi-threaded *target* operating systems, etc, to identify the thread or context in which the call is being made. Host originated messages, and target originated messages on singly-threaded targets should set both of these values to 0xFFFFFFFF.

2.1.4 Flow control

Each service has been allocated a request channel and a response channel. A response packet must be received on the response channel for every request sent on the request channel. Thus flow control is implemented by program control, rather than explicit action of the channel layer below.

2.1.5 Byte ordering

In both ADP and C Library Support Protocol, data is transmitted little-endian. The byte format of other application data is defined by those applications; the transport protocol delivers the data in the order presented.

2.1.6 Startup and shutdown

Startup of the link is defined by the boot protocol, which is outlined in the Boot Agent section.

2.1.7 Reliability and error detection

The protocol assumes that the channel is error free, and makes little allowance for recovery if this is not in fact the case. About the only instance of such allowance is the ADP_LinkCheck packet which is sent in response to parameter negotiation, which is used to verify the connection has restarted correctly.

2.1.8 Predefined channel numbers

The following channels have been predefined for the Angel debug agent. Each service has a full duplex channel assigned to it. ADP uses two channels because with the ADP protocol each end acts sometimes as an RPC server (carries out requests) and sometimes as client (makes requests).

Name	Channel #	Description
CI_HADP	1	ADP, host originated
CI_TADP	2	ADP, target originated
CI_HBOOT	3	Boot, host originated
CI_TBOOT	4	Boot, target originated
CI_TLOG[1]	10	Target debug/logging

Angel debugging channels

In addition, there are a number of other channels, shown in the following table:

[1] The TLOG channel is only used if the Angel ROM debugging method is set to “logadp” which tells Angel to send ROM debugging messages over ADP to the host; normal builds of Angel use “panicblk”, in which these messages are stored in a small area of host memory.

Name	Channel #	Description
CI_PRIVATE	0	Channel protocol control messages
CI_CLIB	5	Semihosting C library support
CI_HUDBG	6	User debug support, host originated
CI_TUDBG	7	User debug support, target originated
CI_HTDCC	8	Thumb debug comms channel, host originated
CI_TTDCC	9	Thumb debug comms channel, target originated

Predefined angel channels

2.1.9 Operations

The reason codes are defined by the relevant protocol. There is no danger of a valid C Library Support packet being mistaken for an ADP packet as they are sent down different channels. The channels are specified in the channel level protocol.

The top bit of the reason code is used to indicate whether the message is a host to target message or a target to host message. Note that the same basic reason code is used for each direction, but depending on the direction, the packets may have differing data formats.

Details of the requests, responses and error numbers can be found in a separate document *Angel Debug Protocol Messages* (ARM DUI0053).

2.2 Channel Layer

The channel level protocol is responsible for delivering packets from one end of the communications medium to the other along numbered channels. *Channels* are conceptually equivalent to *ports* in the world of Unix sockets, although in the current implementation the mapping of ports to services is statically defined. An Angel channel is also bidirectional.

The interface above this layer is packet based; calls to the layer present a data packet for transmission, with the guarantee that it will be delivered, if at all, in the order presented and intact. Note that the channel layer only guarantees that packets arrive at all if it has specifically been asked to when the packet was passed to it by requesting the reliable service level.

The interface below this layer is to a communications medium, such as a serial line, which is capable of delivering packets to the destination without additional addressing information; the channel layer considers that all communications are point-to-point. The medium is also expected to deliver packet contents in the order transmitted, and although the protocol is capable of packet reordering it is very inefficient at doing so; the protocol basically expects packets to arrive in the order sent.

2.2.1 Data formats

The channel level protocol is used to distinguish which channel a packet is destined for. It was also envisaged as the protocol which contained the information for packet sequencing. The current protocol header is 4 bytes long. The bytes (in transmission order) are:

- Channel ID
- Host sequence number
- Target sequence number
- Flags byte

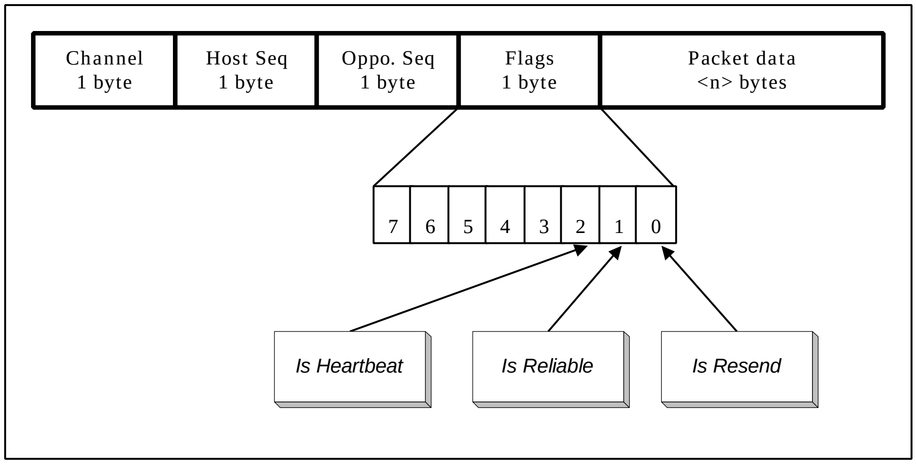

Figure 4: Channel packet data format

2.2.1.1 Channel ID

The Channel ID is checked for validity very early on in receipt of the packet. It must be less than a predefined limit (the sender and receiver must agree on a specific subset of channel ID numbers and their meanings).

2.2.1.2 Sequence numbers

The host and opposite sequence numbers are used to determine the relative state of "progress" between sender and receiver (eg. is the sender "ahead" of the receiver, so the receiver needs to request a resend). In more standard terminology:

Host Sequence Number == Packet Sequence Number == Transmitter's sequence

Opposite Sequence Number == Acknowledge sequence == Receiver's sequence

Sequence numbers are global and not limited by the protocol window size. That is, they start at zero, are incremented by one until the byte they are held in wraps around to zero again. The per-channel protocol however, is a simple Alternating Bit Protocol[1] (for each channel, there are only two valid sequence numbers, last and current, which can be represented by a single bit).

[1] *See* Bartlett, Scantlebury, Wilkinson *"A note on Reliable Full Duplex Transmission over Half Duplex Links"*, in Communications of the ACM, volume 12 (1969) no 5.

The sequence numbers are used as follows:

When a packet is to be sent:

1. It is marked with the current values of the transmit and next expected sequence numbers.
2. If and only if the packet is a reliable packet, the transmit sequence number is incremented by one and a reference to the packet stored in the resend packet list.
3. The packet is written to the output device.

When a packet is received:

- If the packet was considered 'bad' by the device layer, a resend packet may be sent, requesting retransmission of the packet the receiver was expecting next (2.11a Angel does *not* request a resend). The packet is thrown away and no further action is taken.
- If the packet's flags indicate this is a heartbeat packet, the packet sequence number from the packet is checked against the receiver's idea of this value:
  - If the packet's value is lower (mod 255) than expected, then the packet's transmitter lost the last packet sent to it for that channel. An acknowledge heartbeat is (in 2.11a, but *not* in 2.11 or 2.10) returned.
  - If the packet's value is higher than expected then the receiver has missed a packet on that channel and it should request a resend of the missing packet. The current packet is thrown away.
  - If the packet's value matches the expected value, a further check must be made (in 2.11a only):
  - If a data packet has already been received with this sequence number, an acknowledge heartbeat is returned; otherwise, a resend message is sent for the expected sequence number–the host sent a packet which has been lost.

  The current packet is then thrown away and no further action is taken.
- If the packet's flags indicate this is not a reliable packet, then the packet is delivered and no further action is taken.

- The packet sequence number from the packet is recorded (for heartbeat checks, 2.11a only) and checked against the receiver's idea of this value:
  - If the packet's value is lower than expected, then this packet is a duplicate of the last received packet on the same channel. The current packet is thrown away.
  - If the packet's value is higher than expected then the receiver has missed a packet and it requests a resend of the missing packet. The current packet is thrown away.
  - If the packet's value matches the expected value, the next expected sequence number is incremented, the channel's last transmitted packet (2.11a: and any previous packets) is removed from the resend buffer, as correct receipt of this packet has acknowledged it, and the packet is delivered.

Note *The current implementation simply maintains a list of currently-unacknowledged packets, rather than maintaining one per channel.*

2.2.1.3 Flags values

The Flags byte is used to distinguish various packets from each other. The three flag bits are mutually exclusive; a resend packet cannot be reliable, and neither is a heartbeat. Thus there are four packet types available:

- Resend
- Heartbeat
- Reliable data
- Unreliable data ("Datagram").

See the later descriptions of packet types for more information about these packets.

2.2.1.4 Flow control

Flow control is not implemented in this layer.

2.2.1.5 Byte ordering

Data is transmitted in little-endian format, although this is effectively irrelevant as the only fields are single bytes.

2.2.1.6 Startup and shutdown

On startup, each end initializes the values used for the host and target (opposite) sequence numbers to zero. The device driver is initialized and the state (of the target) set to BootAvailable.

2.2.1.7 Reliability and error detection

The protocol attempts to deliver packets marked with the `reliable` flag reliably, that is, it attempts to ensure that once the packet is presented to the channel layer it does actually reach the destination without error. There are two mechanisms which are used to implement this:

- CRC32 checking of the data
- checking of the packet sequence numbers against the expected range

If the `reliable` flag is not set, the channel layer simply delivers packets to the data link layer for transfer; the application is responsible for any error recovery actions, although CRC checking is still performed, as this is actually part of the data link layer.

The CRC checks allow the receipt of packets which have become corrupted to be detected. The algorithm used is the IEEE 802.3 32-bit CRC algorithm for byte data. In the case where a packet fails the CRC check, the packet is transmitted to the upper layers with a ‘bad frame’ type code.

When receiving bad frames whether detected by framing errors or by CRC, the channel protocol simply requests the currently expected packet be transmitted again.

The sequence numbers used are more complex. Each end of a link tries to keep track of two sequence numbers–the home number, which is simply a count of (reliable) packets sent, and the opposite number, which is what the local endpoint *thinks* the other end’s home sequence number is.

2.2.1.8 Packet length

Packets are formed in buffers which are either a minimal (256 byte) or long (often 7KB, but variable) length. For any particular request, the packet sent is the length of the data in it, not the length of the buffer.

When performing block data transfers, larger buffers, up to 16KB in length, are used to reduce the overheads incurred in packet transfer and subsequent processing. Note that use of large buffers is currently restricted (by convention) to the ADP_Write operation. This is likely to change in future.

Maximum packet lengths must be agreed between the sender and receiver. During link startup the host asks the target using ADP_Info calls the size of standard and long packets, and adjusts its own packet sizes to match.

When using ADP and the C library, many requests or responses are in fact significantly less than 256 bytes; the most common length is around 32 bytes, inclusive of the various headers.

2.2.1.9 Packet types: resend requests

A resend request packet contains no data; the opposite sequence number received in a resend packet defines the start of the resend sequence (as the last packet successfully received), and the end is defined as the “current” packet. All packets within this range will be resent in order, from earliest to latest. The receiving system interprets the packets as normal and sends back acknowledge packets as appropriate. A resend request is completed when the last packet has been sent.

A resend request is initiated as described above.

New packet sequence numbers are only allocated to packets which can be resent; the current sequence number is included in other packet types, but only in order that receipt of these packet types can cause a resend request (of a previous, resendable packet). A resend request packet should not cause a resend request.

The behaviour of the system is currently undefined if a resend request is received which cannot be satisfied because the packet is not available. Care should be taken to ensure this does not happen. The 2.11a Angel port currently detects the attempt and ignores the request.

2.2.1.10 Heartbeats

A heartbeat packet contains nothing, other than the sequence numbers, which is required by the protocol; its mere existence is what is needed. Its purpose is to ensure that the target end of the link is still alive, even if no other data transfer is occurring.

Heartbeats are initiated by the host system, and merely reflected by the target (including a *copy* of the time stamp, see below). Heartbeats do not count in the normal sequencing of packets (ie. the sending of a heartbeat packet, while it does include host and opposite sequence numbers, does not imply the incrementing of those numbers).

A recent addition (in 2.11) is the inclusion of a time stamp in centiseconds, which allows the host to determine the current round trip delay. The absolute value of the timestamp is not useful, and targets should not assume any interpretation of it. Currently, Remote_A uses the timestamp to calculate the round trip delay on the link.

2.2.1.11 Reliable data

A reliable data packet is remembered, until acknowledged by the receipt of a reliable data packet with the next higher sequence number. Packets sent in this manner are recorded on the resend list. A resend request examines the resend list to identify the packets which can be resent, and resends them in order. Packets are removed from the resend list when a packet is received from the target in response (ie. it has a sequence number one (or more) higher).

2.2.1.12 Unreliable data

An unreliable data packet is a traditional datagram; it is up to the application to determine whether the packet has been lost or corrupted, and what to do if it has been. Few higher level services use this level of service. The channel layer still checks packets have not been corrupted in transit. Note that while a bad packet indication may be received if a packet is corrupted, there are occasions when this will not happen, even if a partial packet is received.

2.3 Boot agent

The boot agent uses the channel layer reliable packet stream, and must establish communication between the host and the target. This involves determining the device used for data transport, getting the host and target into sync and agreeing on the parameters (such as maximum message size) which will be used for the session.

At the end of the session, the boot agent must return the system to a state where another session can be initiatied.

The boot agent only supports a few messages. All Angel systems with host communications *must* provide the boot agent, even if they do not have support for semihosting or debug agents.

2.3.1 Target board powered up before the host

After switching on the target and initialisation is completed the target will send an `ADP_Booted` message. The debugger has not been started yet so this message will not be received. In a serial world this makes it important that any buffers on the host side are flushed during initialisation of the debugger, and in an Ethernet world it makes it important that the target can cope with the message not being received.

Eventually the debugger will be started up and will send an `ADP_Reboot` or `ADP_Reset` request[1]. The target will respond to this with an `ADP_Reboot` or `ADP_Reset` acknowledge and will then reboot, finally sending an `ADP_Reboot` when it has done all it needs to do (very little in the case of `ADP_Reset`, but completely rebooting in the case of `ADP_Reboot`).

The target and host are now ready to start a debug session.

2.3.2 The target board powered up after the host

The debugger will send an `ADP_Reboot` or `ADP_Reset` request, but will receive no reply until the target is powered up.

When the target is powered up then it will send an `ADP_Booted` message to the debugger. The debugger should accept this message even though it has received no `ADP_Reboot` or `ADP_Reset` acknowledge message from the target. ARM host debuggers will then proceed to reset the target (with `ADP_Reset`), prompting another `ADP_Booted` message prior to initiating the debug session.

[1] Currently, Remote_A always sends ADP_Reset. Do not rely on this behaviour.

For serial links, the initial baud rate should be set to 9600, although it may be renegotiated to a higher (or lower) value very early on.

The optional renegotiation packets are shaded grey in the startup sequences 1 to 5, below.

Host	Direction	Target	Channel
	<–	Boot message	BOOT
		(wait)	
Parameter negotiate	–>		BOOT
	<–	Parameter negotiate acknowledge	BOOT
Link check	–>		BOOT
	<–	Link check acknowledge	BOOT
Reset	–>		BOOT
	<–	Reset acknowledge	BOOT
	<–	Boot message	BOOT
Boot acknowledge.	–>		BOOT
(Debug session)	–>		DEBUG

Sequence 1: Serial or serial/parallel startup sequence when target boots first

Host	Direction	Target	Channel
Parameter negotiate	–>		BOOT
	<–	Parameter negotiate acknowledge	BOOT
Link check	–>		BOOT
	<–	Link check acknowledge	BOOT
Reset	–>		BOOT
	<–	Boot message	BOOT
Boot acknowledge	–>		BOOT
(Debug session)	–>		DEBUG

Sequence 2: Serial or serial/parallel startup sequence when host boots first

Host	Direction	Target	Channel
Reset	–>		BOOT
	<–	Boot message	BOOT
Boot acknowledge	–>		BOOT
(Debug session)	–>		DEBUG

Sequence 3: Ethernet startup sequence when host boots first

Note that the channel for some of these transfers is the boot channel, rather than the debug channel. There session may be terminated with an End request, which results in the session moving back to the ‘can connect’ state, as shown below:

Host	Direction	Target	Channel
End	–>		DEBUG
	<–	End	DEBUG
Parameter negotiate	–>		BOOT
	<–	Parameter negotiate acknowledge	BOOT
Link check	–>		BOOT
	<–	Link check acknowledge	BOOT

Sequence 4: Serial or serial/parallel shutdown sequence

Host	Direction	Target	Channel
End	–>		DEBUG
	<–	End acknowledge	DEBUG

Sequence 5: Ethernet shutdown sequence

The target and host are now ready to start a debug session. If at any point during the bootup sequence ADP messages are sent down the CI_HADP channel then they should be responded to with a RDI_NotInitialised error. (This should never happen).

An `ADP_Booted` or `ADP_Reboot` message should be accepted at any point, since it is possible for a catastrophe to occur (such as disconnecting the host and target during a debug message) which requires that one or other end be reset. Note that if a different baud rate has been negotiated for a session from the default 9600, booted messages will not be received correctly. It is expected that this issue will be resolved in future versions of the protocol.

2.4 Device Level

The device level protocol will differ depending upon the device to be used. For example in an ethernet implementation, the device level protocol is UDP. On devices where the device driver communicates directly with the hardware other protocols can be used. The one used by the serial device driver is as follows and is also used on other byte-serial point to point connections.

One special requirement is placed upon the device level interface by the Channel level; that the device level knows how much data is being transferred in the data portion and can reliably inform the channel layer of this on receipt. This condition would not, for example, be met by raw Ethernet (IEEE 802.3 etc.) It never the case that a null packet (ie. a device level packet with no data in it) is transmitted, although this is not a requirement of the current implementations of the device protocol.

2.4.1 Byte serial devices

2.4.1.1 Data formats

The data packet is used to transmit channel level packets, adding framing, error detection and byte escaping to the channel's data block. The end of packet byte transmitted as the last byte in the packet terminates the current packet.

To avoid the data values in a packet being mistaken for these values, the driver escapes the data values (and some others) by transmitting an escape character (value 0x1B) followed by the data value ORed with the value 0x40. So a data byte with the value 0x11 will be transmitted as the byte pair 0x1B 0x51. Clearly data bytes of the value 0x1B will also need to be escaped, becoming 0x1B 0x5B.

Also there are two bytes than can be (but in current implementations, are not) used for software flow control, 0x11 (XON) and 0x13 (XOFF). These characters are also escaped. The implementation of the protocol should not assume that these values are unchanging; the current implementation defines them in the invocation of the device. Note also that it is possible that other characters may need to be escaped.

The current default control bytes are:

Name	Hex Value
Start of Packet	0x1C
End of Packet	0x1D
Escape	0x1B
Stop Sending (XOFF)	0x13
Start Sending (XON)	0x11

As well the start and end bytes the device driver prepends an 8-bit type and 16-bit length field and appends a 32-bit CRC value to the channel packet. So both packet types are of the form:

SOP	TYP	LEN	DATA	CRC	EOP
1 byte	1 byte	2 bytes	<len> bytes	4 bytes	1 byte

Figure 5: Byte Serial Device Data Format

All values in the packet between the start and end packet bytes are escaped if appropriate.

If an error occurs whilst processing a packet (such as receiving an unescaped start of packet byte) then the protocol aborts processing the previous packet, delivers the bad packet to the channel layer, and starts processing the new packet. If an overrun error is detected, the serial driver may (in 2.11a) indicate this and cause packet processing to search for a start of packet byte again, irrespective of the current state.

2.4.1.2 Byte ordering

Data is transmitted in little-endian format. This affects the CRC and length fields of the packet.

2.4.1.3 Packet length

Data packets are used to transfer data from the channel layer to or from the destination. The data being transferred varies from a few bytes to a few kilobytes, and although the length field allows for much larger packets no current implementation allows packets more than approximately 16KB. Most packets are from 32-64 bytes in length.

2.4.1.4 Reliability and error detection

The protocol makes no effort to improve the reliability of the connection. Framing errors, and data errors detected by the CRC are reported to the higher layers as bad packets.

2.4.1.5 Flow control

Byte level flow control is not implemented in the current drivers, although unescaped XON and XOFF characters should be ignored (rather than causing packet errors).

2.4.1.6 Startup and shutdown

Transmitter startup involves initiation of the physical link (eg. setting up the serial data registers, or opening the OS's serial comms port) and setting up the transmit callbacks.

Receiver startup involves a similar initiation of the physical link, and setting up receive callbacks.

The receiver state should be *waiting for start-of-packet* on the link. The transmitter state should be *waiting for a packet to send*.

2.4.2 Other devices

Other devices and drivers can be used to connect the channel layers together. Examples include processor bus interfaces such as PCI and Ethernet, via UDP/IP. Device drivers for these interfaces need to conform to the following requirements:

- Data is received in the same byte order it was transmitted within a packet;
- Data within a packet is either delivered correctly, not delivered at all, or a bad packet indication is given on delivery.
- Data block boundaries are maintained, and the length of the data block actually sent is available at the receiver;
- The maximum data block length is at least 256 bytes, as seen from the driver interface.
- In the normal sequence of events, data blocks are delivered in the order presented. Although the channel layer can reorder packets where necessary, because it is not efficient at doing so the driver should attempt to deliver packets in the correct order.

3

Protocol State Machines

This protocol description is written from the perspective of the target (ie. send means write to host (MASTER) and receive means read from host (MASTER).

```
ADP
{
    Receive Messages
    {
        -- CI_HADP

        ADP_TargetResetIndication_HtoT();
        ADP_Reboot_HtoT();
        ADP_Reset_HtoT();
        ADP_HostResetIndication_HtoT();
        ADP_ParamNegotiate_HtoT();
        ADP_LinkCheck_HtoT();
        ADP_Info_HtoT();
        ADP_Control_HtoT();
        ADP_Read_HtoT();
        ADP_Write_HtoT();
        ADP_CPUread_HtoT();
        ADP_CPUwrite_HtoT();
        ADP_CPread_HtoT();
        ADP_CPwrite_HtoT();
        ADP_SetBreak_HtoT();
        ADP_ClearBreak_HtoT();
        ADP_SetWatch_HtoT();
        ADP_ClearWatch_HtoT();
        ADP_Execute_HtoT();
        ADP_Step_HtoT();
        ADP_InterruptRequest_HtoT();
        ADP_HW_Emulation_HtoT();
        ADP_ICEbreakerHADP_HtoT();
        ADP_ICEman_HtoT();
        ADP_Profile_HtoT();
        ADP_InitialiseApplication_HtoT();
        ADP_End_HtoT();


        doReset();                -- perform a target reset
        doReboot();               -- perform a target reboot
    }
    Send Messages
    {
        -- CI_TADP

        ADP_Booted_TtoH();
        ADP_TargetResetIndication_TtoH();
        ADP_Reboot_TtoH();
        ADP_Reset_TtoH();
        ADP_HostResetIndication_TtoH();
        ADP_ParamNegotiate_TtoH();
```

```
        ADP_LinkCheck_TtoH();
        ADP_HADPUnrecognised_TtoH();
        ADP_Info_TtoH();
        ADP_Control_TtoH();
        ADP_Read_TtoH();
        ADP_Write_TtoH();
        ADP_CPUread_TtoH();
        ADP_CPUwrite_TtoH();
        ADP_CPread_TtoH();
        ADP_CPwrite_TtoH();
        ADP_SetBreak_TtoH();
        ADP_ClearBreak_TtoH();
        ADP_SetWatch_TtoH();
        ADP_ClearWatch_TtoH();
        ADP_Execute_TtoH();
        ADP_Step_TtoH();
        ADP_InterruptRequest_TtoH();
        ADP_HW_Emulation_TtoH();
        ADP_ICEbreakerHADP_TtoH();
        ADP_ICEman_TtoH();
        ADP_Profile_TtoH();
        ADP_InitialiseApplication_TtoH();
        ADP_End_TtoH();

        ADP_TADPUnrecognised();
        ADP_Stopped();

        -- CI_TTDCC

        ADP_TDCC_ToHost();
        ADP_TDCC_FromHost();
    }

    Protocol
    {
        States
        {
            BootStartup,
            BootAvailable,
            BootResetting,
            Connected,
        }
        Transitions
        {
            BootStartup:    -ADP_Booted_TtoH                -> BootAvailable;
            BootAvailable:  +ADP_Booted_HtoT                -> Connected;

            -- These messages exist, but are not used! They were intended to
```

```
-- allow each end of the link to say it had reset "spontaneously"
Connected:      +ADP_TargetResetIndication      ->
Connected:      +ADP_HostResetIndication        ->

-- a reboot request returns a reply, then a complete reinitialisation
Connected:      +ADP_Reboot_HtoT                -> RebootAck;
BootAvailable:  +ADP_Reboot_HtoT                -> RebootAck;
BootStartup:    +ADP_Reboot_HtoT                -> RebootAck;
BootResetting:  +ADP_Reboot_HtoT                -> RebootAck;
RebootAck:      -ADP_Reboot_TtoH                -> BootStartup;

Connected:      +ADP_Reset_HtoT                 -> ResetAck;
BootAvailable:  +ADP_Reset_HtoT                 -> ResetAck;
ResetAck:       -ADP_Reset_TtoH                 -> BootResetting;
BootResetting:  -doReset                        -> Connected;

-- this s just saying a reset, while in reset state, is ignored
BootResetting:  +ADP_Reset_HtoT                 -> ResetAck;


Connected:      +ADP_ParamNegotiate_HtoT        -> ParamNegAck;
ParamNegAck:    -ADP_ParamNegotiate_TtoH        -> ExpectLinkCheck;
ExpectLinkCheck:+ADP_LinkCheck_HtoT             -> LinkCheckAck;
Connected:      +ADP_LinkCheck_HtoT             -> LinkCheckAck
LinkCheckAck:   -ADP_LinkCheck_TtoH             -> Connected;

-- these two messages are sent if the sender doesn't recognise
-- a message
Connected:      +ADP_HADPUnrecognised_HtoT      -> Connected;
Connected:      -ADP_TADPUnrecognised_TtoH      -> Connected;

-- info subtype defines op; reply contains result of op
Connected:      +ADP_Info_HtoT                  -> InfoAck;
InfoAck:        -ADP_Info_TtoH                  -> Connected;

-- control subtype defines op; reply contains result of op
Connected:      +ADP_Control_HtoT               -> ControlAck;
ControlAck:     -ADP_Control_TtoH               -> Connected;

Connected:      +ADP_Read_HtoT                  -> ReadAck;
ReadAck:        -ADP_Read_TtoH                  -> Connected;

Connected:      +ADP_Write_HtoT                 -> WriteAck;
WriteAck:       -ADP_Write_TtoH                 -> Connected;

Connected:      +ADP_CPUread_HtoT               -> CpuReadAck;
CpuReadAck:     -ADP_CPUread_TtoH               -> Connected;
```

```
Connected:      +ADP_CPUwrite_HtoT             -> CpuWriteAck;
CpuWriteAck:    -ADP_CPUwrite_TtoH             -> Connected;

Connected:      +ADP_CPread_HtoT               -> CPReadAck;
CPReadAck:      -ADP_CPread_TtoH               -> Connected;

Connected:      +ADP_CPwrite                   -> CPWriteAck;
CPWriteAck:     -ADP_CPwrite_TtoH              -> Connected;

Connected:      +ADP_SetBreakHtoT              -> SetBreakAck;
SetBreakAck:    -ADP_SetBreak_TtoH             ->

Connected:      +ADP_ClearBreak                ->
Connected:      +ADP_SetWatch_HtoT             -> SetWatchAck;
SetWatchAck:    -ADP_SetWatch_TtoH             -> Connected;

Connected:      +ADP_ClearWatch_HtoT           -> ClearWatchAck;
ClearWatchAck:  -ADP_ClearWatch_TtoH           -> Connected;

Connected:      +ADP_Execute_HtoT              -> ExecuteAck;
ExecuteAck:     -ADP_Execute_TtoH              -> Executing;

Connected:      +ADP_Step_HtoT                 -> StepAck;
StepAck:        -ADP_Step_TtoH                 -> Executing;

Connected:      +ADP_InterruptRequest_HtoT     -> InterruptAck;
InterruptAck:   -ADP_InterruptRequest_TtoH     -> Connected;

-- subtype defines op; reply contains result of op
Connected:      +ADP_HW_Emulation_HtoT         -> HWEMAck;
HWEMAck:        -ADP_HW_Emulation_TtoH         -> Connected;

-- subtype defines op; reply contains result of op
Connected:      +ADP_ICEbreaker_HADP_HtoT      -> IceBrkAck;
IceBrkAck:      -ADP_ICEbreaker_HADP_TtoH      -> Connected;

-- subtype defines op; reply contains result of op
Connected:      +ADP_ICEman_HtoT               -> IceManAck;
IceManAck:      -ADP_ICEman_TtoH               -> Connected;

-- subtype defines op; reply contains result of op
Connected:      +ADP_Profile_HtoT              -> ProfileAck;
ProfileAck:     -ADP_Profile_TtoH              -> Connected;

Connected:      +ADP_InitialiseApplication_HtoT -> InitAppAck;
InitAppAck:     -ADP_InitialiseApplication_TtoH -> Connected;

Connected:      +ADP_End                       -> EndAck
```

```
            EndAck:         -ADP_End                        -> BootAvailable;

            Executing:      -ADP_Stopped                    -> Connected;
            Executing:      +ADP_InterruptRequest_HtoT      -> InterruptAck;
        }
    }
}
```

3.1 Channel Level Protocol

The channel level protocol is mostly (but not perfectly) symmetric due to the master-slave relationship in the protocol. The slave (target hardware, etc.) is not assumed to have a timer, and so merely bounces a heartbeat. There is no good reason why the boot packet has no mirror on the master. The master can reset the slave, but the slave cannot reset the master; again, this is not necessarily justified.

```
Channel
{
    Receive Messages
    {
        ReadHeartbeatPacket(Channel&, HomeSeq&, OppSeq&, Timestamp&);
        ReadResendPacket(Channel&, HomeSeq&, OppSeq&);
        ReadDataPacket(Channel&, HomeSeq&, OppSeq&, Data&);
        TransferFromHost(Data);                         -- get data packet from
host
        Timeout();                                      -- timeout detected on
link
        Heartbeat(Timestamp&);                          -- heartbeat event
        CheckSequenceNumbers();
        BadPacket();                                    -- get invalid packet
from line

        -- ADP PACKETs used by CHANNEL protocol! }
        ReadBootPacket(Channel&, HomeSeq&, OppSeq&, BootInfo&);
        ReadResetPacket(Channel&, HomeSeq&, OppSeq&, Resetinfo);
    }
    Send Messages
    {
        WritePacket(Channel, HomeSeq, OppSeq, Data);
        WriteBootPacket(Channel, HomeSeq, OppSeq, BootInfo);
        WriteResendPacket(Channel, HomeSeq, OppSeq);
        WriteHeartbeatPacket(Channel, HomeSeq, OppSeq, timestamp);
        TransferToHost(Data);                           -- transfer data packet
to user
    }

    Protocol
    {
        States
        {
            Start(init),
            Wait,                   -- waiting for some event
```

```
            SentPacket,              -- have sent a packet out
            SendHeartbeat,           -- write a heartbeat with current sequence
number
            GotResendPacket,         -- got a resend request packet
            GotHeartbeatPacket,      -- got a heartbeat packet
            GotReliablePacket,       -- got a packet flagged "reliable"
            HandleReliablePacket,    -- determine whether the packet is ok
            GotDatagram,             -- got a datagram -- an unchecked packet
            GotBadPacket,            -- received packet with CRC error
            ResendNextPacket         -- resend one or more previously sent packets
            Error                    -- "no recovery" state
        }
        Transitions
        {
            -- initialisation: boot packet is not, however, part of the
            -- Channel protocol!

#ifdef MASTER
            Start:                   -ReadBootPacket             -> Wait;
#else
            Start:                   +WriteBootPacket            -> Wait;
#endif
            -- receive incoming packet
            Wait:                    +ReadDataPacket             -> GotDataPacket;
            Wait:                    +ReadResendPacket           ->
GotResendPacket;
            Wait:                    +ReadHeartbeatPacket        ->
GotHeartbeatPacket;
            Wait:                    +BadPacket                  -> GotBadPacket;

#ifdef MASTER
            -- receiving a boot packet signals that the SLAVE has reset;
            -- we must do so too!
            -- THIS ACTION IS NOT PART OF ANY CURRENT PROTOCOL OR IMPLEMENTATION
            Wait:                    +ReadBootPacket             -> Reset;
            Reset:                   +(doChannelReset)           -> Wait;
#else

            -- Again, a reset packet is not part of the channel protocol!
            Wait:                    +ReadResetPacket            -> Reset;
            Reset:                   +WriteBootPacket            -> Wait;
#endif

            -- handle incoming packet, see also heartbeat below
            GotBadPacket:            -WriteResendPacket          -> Wait;
            GotDatagram:             -TransferToHost             -> Wait;

            GotReliablePacket:       -CheckSequenceNumbers       ->
```

```
HandleReliablePacket;
            HandleReliablePacket:   +TransferToHost             -> Wait;
            HandleReliablePacket:   +WriteResendPacket          -> Wait;

            -- resend zero or more packets as requested
            GotResendPacket:                                    -> Wait;
            GotResendPacket:        -WritePacket                ->
ResendNextPacket;
            ResendPacket:           -WritePacket                ->
ResendNextPacket;

            -- handle packet from local application
            Wait:                   +TransferFromHost           -> SentPacket;

            -- handle timeout and heartbeat
            Wait:                   +Timeout                    -> Error;  --
non-auto: Start

#ifdef MASTER
            Wait:                   +Heartbeat                  ->SendHeartbeat;
            SendHeartbeat           +WriteHeartbeatPacket       -> Wait;
            GotHeartbeatPacket:                                 -> Wait;
#else
            GotHeartbeatPacket:     -WriteHeartBeatPacket       -> Wait;
#endif
        }
    }
}
```

3.2 Serial Data Link Level Protocol

The data level protocol given below is intended to operate over character-orientated devices such as serial and parallel lines. Two protocols are given, one instance of each running in parallel on each end of the link to allow full duplex access to the link.

The protocol defines the startup state of the line as a set of default parameters which can be renegotiated by the application using the ADP_ParamNegotiate request.

```
ChannelReceive
{
    Receive Messages
    {
        RecieveIntr();          -- Input Characters Available
        GotSTX();               -- got Start-Of-Packet character
        GotETX();               -- got End-Of-Packet character
        GotChar();              -- got a character in field
        GotLast();              -- got last character in field
        hasData();              -- is receive packet len > 0
        noData();               -- is receive packet len == 0
        invalidLen();           -- is receive packet len invalid (too big?)
    }
    Send Messages
    {
        BadPacketToHost();      -- deliver bad packet
        PacketToHost();         -- deliver completed packet to host
    }

    Protocol
    {
        States
        {
            Start(Init),
            Wait,

            WantSTX,
            WantTYP,
            WantLEN,
            WantDAT,
            WantCRC,
            WantETX,
            CheckCRC,
            BadPacket_STX,
            BadPacket_TYP
```

```
        }
        Transitions
        {
            Start:                                          -> Wait;

            Wait:            +ReceiveIntr                   -> WantSTX;

            WantSTX:         +GotSTX                        -> WantTYP;
            WantSTX:         +GotETX                        -> WantSTX;
            WantSTX:         +GotChar                       -> WantSTX;
            WantSTX:         +GotLast                       -> WantSTX;

            BadPacket_STX    -BadPacketToHost               -> WantSTX;
            BadPacket_TYP    -BadPacketToHost               -> WantTYP;

            WantTYP:         +GotSTX                        -> BadPacket_TYP;
            WantTYP:         +GotETX                        -> BadPacket_STX;
            WantTYP:         +GotChar                       -> BadPacket_STX;
            WantTYP:         +GotLast                       -> WantLEN;

            WantLEN:         +GotSTX                        -> BadPacket_TYP;
            WantLEN:         +GotETX                        -> BadPacket_STX;
            WantLEN:         +GotChar                       -> WantLEN;
            WantLEN:         +GotLast                       -> CheckCRC;

            WantDAT:         +GotSTX                        -> BadPacket_TYP;
            WantDAT:         +GotETX                        -> BadPacket_STX;
            WantDAT:         +GotChar                       -> WantDAT;
            WantDAT:         +GotLast                       -> WantCRC;

            WantCRC:         +GotSTX                        -> BadPacket_TYP;
            WantCRC:         +GotETX                        -> BadPacket_STX;
            WantCRC:         +GotChar                       -> WantCRC;
            WantCRC:         +GotLast                       -> WantETX;

            WantETX:         +GotSTX                        -> BadPacket_TYP;
            WantETX:         +GotETX                        -> CheckCRC;
            WantETX:         +GotChar                       -> BadPacket_STX;
            WantETX:         +GotLast                       -> BadPacket_STX;

            CheckCRC:        -goodCRC                       -> DeliverPacket;
            CheckCRC:        -badCRC                        -> BadPacket_STX;

            DeliverPacket:   -PacketToHost                  -> Wait
        }
    }
}
```

```
ChannelTransmit
{
    Receive Messages
    {
        WriteChar(Char);        -- write a character to the line
        finished();             -- have we got to the end of the body
        isPlain();              -- is the "current character" a plain
        isSpecial();            -- is the "current character" a special
    }
    Send Messages
    {
        PacketFromHost();       -- get packet from host; calc CRC & length
    }

    Protocol
    {
        States
        {
            Start(Init),
            Wait,

            SendSTX,
            SendBody,
            SendEscaped,
            SendSpecialChar,
            SendPlainChar,
            SendETX,
        }
        Transitions
        {
            Start:                                  -> Wait;

            Wait:           +PacketFromHost         -> TransmitSTX;

            -- PacketFromHost gives us a "body" with the length and CRC
            -- filled in for us

            SendSTX:        -WriteChar(STX)         -> SendBody

            -- Send body must work through the data character by character
            -- escaping those characters which are special to the protocol

            SendBody:       -isPlain                -> SendPlainChar
            SendBody:       -isSpecial              -> SendSpecialChar
            SendBody:       -finished               -> SendETX

            SendPlainChar   -WriteChar(c)           -> SendBody
```

```
            SendSpecialChar: -WriteChar(Escape)      -> SendEscaped
            SendEscaped:    -WriteChar(c)            -> SendBody

            SendETX:        -WriteChar(ETX)          -> Wait
        }
    }
}
```

4 Glossary

This section provides a brief glossary of key terms used in this document.

Glossary

Terms are used in this document with the following meanings:

Term	Definition
ADP	The Angel Debug Protocol. In various documents, this refers either to the whole protocol or just to the high level debug messages. With the exception of the title, this document uses the latter definition.
Remote_A	The name of the host-end of the protocol suite. Implemented as a static library on Unix, and a dynamic link library on Windows, this converts debugger requests into packet requests and then interprets the responses.
UDP	The User Datagram Protocol, part of TCP/IP.
Host	The computer on which the debugger software is running, usually Win32 or Unix operating systems.
Target	The computer being debugged, typically an ARM or customer designed ARM processor development board.
Semihosting	The ARM C Library running on the target performs some of its operations internally, and some with the help of the host computer. This is known as semihosting.