Angel
Debug Protocol Messages

Document number: ARM DUI 0053D
Issued: Sept 1999
Copyright © ARM Ltd 1997-1999
Key

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

1. Introduction  
   1.1 RPC Type Specifications  

2. ADP Remote Procedure Calls  
   2.1 Function Signatures and Packet Formation  
   2.2 Boot Channel Messages  
   2.3 Host ADP Channel Messages  
   2.4 Target ADP Channel Messages  
   2.5 Target Debug Comms Channel Messages  

3. C Support Library Remote Procedure Calls  
   3.1 Function Signatures and Packet Formation  
   3.2 Target to Host Messages  

4. Error Codes  
   4.1 ADP Error Codes  
   4.2 RDI Error Codes
1. Introduction

The Angel Debug Protocol (ADP) provides a reliable connection between a debug target and a host debugger during a debugging session. This document describes the messages used at the higher levels of the protocol, notably by the debug agent (the ADP messages) and the C Library. For details of the protocols, ordering constraints and other information, please refer to the document Angel Debug Protocol (ARM DUI 0053).

This document describes ADP version 1.0 and the notes the extensions included in version 1.1.
**Introduction**

### 1.1 RPC Type Specifications

Because the protocols define a procedure call interface, albeit with servers at both ends of the connection, the interface is described here in terms of conventional function calls. The relevant terminology is presented below:

<table>
<thead>
<tr>
<th>Syntax element</th>
<th>Simple description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>An integer type, 8 bits in size. Often used for character storage.</td>
<td>BYTE char:</td>
</tr>
<tr>
<td>INT16</td>
<td>A signed integer, 16 bits in size.</td>
<td>INT16 count:</td>
</tr>
<tr>
<td>INT32</td>
<td>A signed integer, 32 bits in size.</td>
<td>UINT32 address:</td>
</tr>
<tr>
<td>UINT16</td>
<td>An unsigned integer, 16 bits in size.</td>
<td>UINT16 count:</td>
</tr>
<tr>
<td>UINT32</td>
<td>A unsigned integer, 32 bits in size.</td>
<td>UINT32 address:</td>
</tr>
<tr>
<td>[&lt;size&gt;]TYPE</td>
<td>The type is an array of &lt;size&gt; elements; if &lt;size&gt; is missing, the size is variable.</td>
<td>[32]BYTE message:</td>
</tr>
<tr>
<td>RECORD … IS … :</td>
<td>A record definition.</td>
<td>(see below)</td>
</tr>
<tr>
<td>VAL … IS … :</td>
<td>A constant specification.</td>
<td>VAL Escape IS 27:</td>
</tr>
<tr>
<td>&lt;literal&gt;(&lt;type&gt;)</td>
<td>Consider &lt;literal&gt; to be of type &lt;type&gt;. This is a type decoration, explicitly defining the interpretation of a literal.</td>
<td>32(UINT32)</td>
</tr>
<tr>
<td>(&lt;result&gt;) = &lt;name&gt;(&lt;params&gt;)</td>
<td>Define a function called &lt;name&gt;, returning &lt;result&gt; when given a parameter list &lt;params&gt;.</td>
<td>(INT16 x) = sin(INT16 y)</td>
</tr>
</tbody>
</table>

Arrays of BYTES specifically representing strings will be zero terminated unless otherwise noted. The size includes the zero terminator byte.

Multiple return values are represented as a list in the same way as multiple parameters, with a bracketed list of types and names. Where types do not change between successive items, the type name can be omitted (that is, "UINT32 x, UINT32 y" is the same as "UINT32 x, y").
Named groups of types can be represented with record definitions. A record is described with the following syntax:

RECORD <name> IS (<list>):

For example, a point could be represented with the following definition:

RECORD Point IS (UINT32 x, UINT32 y):

Records are used much the same way as ordinary types:

RECORD Point x:

or an array of 32 points:

[32]RECORD Point point.segments:

To represent a variable length array, the following structure should be used:

<inttype> size, [size]<arraytype> name:

Where <inttype> is an integer (BYTE, INT16, UINT32) type containing the size of the array, and <arraytype> is the type of each element, eg. BYTE. It should be noted that the name of the size variable also appears in the array bound for the array.

Finally, it should be noted that use is made of the constant –1 as a (usually error) return value, especially in the Clib requests. This constant is always encoded as 0xFFFFFFFF.
This chapter describes the calls used to boot the target system (the TBOOT and HBOOT channels) and the calls used by the debugger to debug the target (the HADP and TADP channels).
2.1 ADP 1.1

The ADP has been extended in two ways for version 1.1:

- the packets types ADP_ReadExt, ADP_WriteExt have been added, extending the original calls with an access method parameter
- All memory and file I/O packet types (ADP_Read, ADP_ReadExt, ADP_Write, ADP_WriteExt, CL_Read, CL_Write, CL_ReadX, CL_WriteX) may use the long buffer system used only for ADP_Write in previous versions.

There are two ways for a debugger to identify the version of ADP in use:

- In the boot message, the ADP version number for 1.0 is 0x3 – for version 1.1 it must be 0x4.
- If the agent implements the ADP_ReadExt or ADP_WriteExt functions, it can be assumed to be conformant to at least version 1.1 of ADP: it may, of course, implement a higher (as yet unspecified) version.

All other details of ADP are unchanged.
2.2 Function Signatures and Packet Formation

As an example of the format used to describe a remote call, the following text identifies a function called CalcMinMax which accepts a 32-bit integer, a previous maximum and minimum value found, and returns two values, a new minimum and maximum. When packetized, a reference number 8 is used to represent the name:

Channel: CI_MISC
Reason Code: 8

Signature:

\[(\text{UINT32 } \text{newmin}, \text{UINT32 } \text{newmax}) = \text{CalcMinMax}(\text{UINT32 } \text{value}, \text{lastmin}, \text{lastmax})\]

Parameters:

- **value**: a new value to check
- **lastmin**: the previous minimum value found
- **lastmax**: the previous maximum value found
- **newmin**: the new minimum value, taking ‘value’ into account
- **newmax**: the new maximum value, taking ‘value’ into account

When converted into ADP form, this translates into two packets as shown in the figures below. The packets are, of course, also wrapped with the data link protocol used at the time.

Notes:

- There is no length information explicitly present; the function knows there are 3 parameters, and simply extracts them from the packet. Similarly, when returning the data, it just constructs a packet with two words in it and sends this off to the caller.
- While the request packet’s reason code has been set to 8, the response code is set to 0x80000008; this is because the TtoH bit has been set in the response. For a request originated by the target, this would be reversed.
### Request Packet:

<table>
<thead>
<tr>
<th>Reason 4 bytes</th>
<th>Debug ID 4 bytes</th>
<th>OS info 1 4 bytes</th>
<th>OS info 2 4 bytes</th>
<th>Value 4 bytes</th>
<th>Min 4 bytes</th>
<th>Max 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>4</td>
<td>2943</td>
</tr>
</tbody>
</table>

**Bytes, hex:**

8 0 0 0 ff ff ff ff ff ff ff ff ff ff 17 0 0 4 0 0 7f b 0 0

### Response Packet:

<table>
<thead>
<tr>
<th>Reason 4 bytes</th>
<th>Debug ID 4 bytes</th>
<th>OS info 1 4 bytes</th>
<th>OS info 2 4 bytes</th>
<th>New Min 4 bytes</th>
<th>New Max 4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000-0008</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>2943</td>
</tr>
</tbody>
</table>

**Bytes, hex:**

8 0 0 80 ff ff ff ff ff ff ff ff ff ff 2 0 0 0 7f b 0 0
2.3 Boot Channel Messages

2.3.1 ADP_Booted

Channel: CI_TBOOT
Reason Code: 0
Signature:

\[(\text{UINT32} \ status) = \text{ADP}_\text{Booted}(\text{VAL} \text{ UINT32} \text{ bufsize, largesize, angelversion, adpversion, archinfo, cpuinfo, hwstatus, bannersize, [bannersize][BYTE banner])}\]

Parameters:

- bufsize: Angel message default buffer size
- largesize: Angel message large buffer size (may be same as default)
- angelversion: Angel version number, including type (eg. boot ROM)
- adpversion: ADP version number
- archinfo: ARM architecture information
- cpuinfo: ARM CPU information, including target endianness.
- hwstatus: Target hardware status
- bannersize: Number of bytes in banner message, must be \(\leq 204\) bytes.
- banner: Startup banner message (single-threaded readable descriptive text, not null terminated)
- status: \(\text{Adp\_ok}\) for success, otherwise indicates an error

Description:

This message is sent by the target after the Angel system has been initialized. This message also contains information describing the Angel world. The information can then be used to check that the target debug agent and source debugger are compatible.
Angel version word
The Angel version word is formatted as shown below:

```
  24-31  16-23  0-15
  Unused  Angel Type  Angel version number
           1 Byte     1 Byte
```

The version number is stored in BCD, with the least significant digit stored in the least significant byte of the word. The Type is set according to the following table, depending on the nature of the Angel code which is running:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot Rom</td>
<td>0</td>
<td>Monitor ROM providing download capability.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Standard type.</td>
</tr>
<tr>
<td>Application ROM</td>
<td>1</td>
<td>ROM based application</td>
</tr>
<tr>
<td>Application Download</td>
<td>2</td>
<td>Downloaded Angel-based application</td>
</tr>
<tr>
<td>Unknown</td>
<td>3</td>
<td>Unknown type.</td>
</tr>
</tbody>
</table>

The unused bits at the top of the word should be ignored on read, and written as zero.

ADP version word
Currently only the least significant 8 bits are used in this word, which contains the version of ADP being used by this version of Angel.
ARM architecture word
This word defines the ARM architecture version of the target CPU. There are only two bits defined, which are both set when the given architecture feature is present:

![ARM architecture word diagram]

CPU architecture word
The following flags describe the feature set of the processor
Note that if the system can be switched between big-endian and little-endian modes of operation, both flags can be set at once.

![CPU architecture word diagram]
Target status word

The target status word contains flags which reflect current target hardware status:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Flag Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1 = cache on</td>
</tr>
<tr>
<td>30</td>
<td>0 = no cache or cache off</td>
</tr>
<tr>
<td>29</td>
<td>1 = big-endian</td>
</tr>
<tr>
<td>0-28</td>
<td>0 = little-endian</td>
</tr>
<tr>
<td></td>
<td>1 = MMU on</td>
</tr>
<tr>
<td></td>
<td>0 = no MMU or MMU off</td>
</tr>
<tr>
<td></td>
<td>Unused, set to zero</td>
</tr>
</tbody>
</table>

2.3.2 ADP_TargetResetIndication

Channel: CI_TBOOT

Reason Code: 1

Signature:

\[
\text{RECORD Parameter IS (UINT32 param, UINT32 value):}
\]

\[
() = \text{ADP_TargetResetIndication(UINT32 status,}
\]

\[
\text{UINT32 nparams, [nparams]RECORD Parameter params)}
\]

Parameters:

- status: Always 0 (makes body same as ADP_ParamNegotiate response)
- nparams: The number of parameters supplied
- params: The parameter list supplied

Description:

If parameter negotiation is enabled at the target, it configures itself to various likely parameter settings and sends this message at each configuration. The message describes the default settings, and after sending at each configuration the target sets itself to the defaults it has just broadcast, to await either an acknowledgement on TBOOT or a request or reset indication on HBOOT. If the host receives this message successfully, it resets to the indicated parameters and sends a reply. The reply has no associated body, just the normal ADP header.
2.3.3 **ADP_Reboot**

Channel: CI_HBOOT

Reason Code: 2

Signature:

\[(\text{UINT32 status}) = \text{ADP_Reboot}(\text{UINT32 hostfeatures})\]

Parameters:

- hostfeatures: Host supported feature word
- status: Boot acknowledgement code

Description:

This message is sent when the host wants the target system to be completely reset, back to the boot monitor Angel. This is how a host forces a cold reboot.

**Note**

An acknowledgement message must be sent by the target immediately; this must be sent before reboot occurs.

The parameter to this function is a bitset of host supported features (the same as for ADP_Reset below). This can be used by the target system to avoid using debug channel bandwidth raising messages that will be ignored by the host.

**Host feature word**

If the host sets neither of these in the word sent on a reset or reboot, the host does not care about the endianess of the target.
ADP Remote Procedure Calls

Boot acknowledgement codes
The following codes are defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Ack</td>
<td>0</td>
<td>Will comply, immediate booted message</td>
</tr>
<tr>
<td>Late Ack</td>
<td>1</td>
<td>Will comply, late startup</td>
</tr>
<tr>
<td>Error</td>
<td>2</td>
<td>Cannot comply</td>
</tr>
</tbody>
</table>

### 2.3.4 ADP_Reset

**Channel:** CI_HBOOT

**Reason Code:** 3

**Signature:**

\((\text{UINT32 } \text{status}) = \text{ADP\_Reset(\text{UINT32 } \text{hostfeatures})}\)

**Parameters:**
- **hostfeatures**: Host supported feature word
- **status**: Boot acknowledgement code

**Description:**
This message is a request from the host, which should eventually result in the "ADP\_Booted" message being sent by the target. An acknowledgement message will be sent immediately; this must be sent before the target can reset. This reset message is always treated as a warm boot, with the target preserving as much state as possible. However, the host will expect packet sequence numbers to be reset to zero, and consequently any unacknowledged packets should be deleted.

The parameter to this function is a bitset of host supported features. This can be used by the target system to avoid using debug channel bandwidth raising messages that will be ignored by the host.

The Host features word, and the Boot acknowledgement status code, are the same as for ADP\_Reboot.
2.3.5 ADP_HostResetIndication

Channel: CI_HBOOT
Reason Code: 4

Signature:

RECORD Parameter IS (UINT32 param, UINT32 value):

() = ADP_HostResetIndication(UINT32 status, UINT32 nparams, [nparams]RECORD Parameter params)

Parameters:
status Always 0 (makes body same as ADP_ParamNegotiate response)
nparams The number of parameters supplied
params The parameter list supplied

Description:
This is as for ADP_TargetResetIndication, but is sent by the host when it first starts up in case the target is listening at a non-default setting. Having sent at various configurations, the host then listens at the defaults it has just broadcast, to await either an acknowledgement on HBOOT or a reset indication on TBOOT.
For arguments and reply, see ADP_TargetResetIndication.
This message is not used in current implementations of Angel.

2.3.6 ADP_ParamNegotiate

Channel: CI_HBOOT
Reason Code: 5

Signature:

RECORD Parameter IS (UINT32 param, UINT32 value):

RECORD ParameterSet IS (UINT32 param, nvalues, [nvalues]UINT32 values):

(UINT32 status, nparams, [nparams]RECORD Parameter params) =

ADP_ParamNegotiate(UINT32 noptions, [noptions]RECORD ParameterSet options)

Parameters:
noptions The number of parameters supplied
options The parameter option list supplied
status Adp_ok if negotiate succeeded, error code otherwise
ADP Remote Procedure Calls

npnams The number of parameters returned
params The parameter option list returned

Description:
The host sends this messages to negotiate new parameters with the target. For each parameter the host specifies a range of possibilities, starting with the most favoured. All possible combinations of parameters must be valid.

If the target can operate at a combination of the offered parameters, it will reply with the parameters it is willing to use. After sending the reply, the target switch to this combination. On receiving the reply, the host will switch to the new combination and send a ADP_LinkCheck message (see below).

If the target cannot operate at any combination of the offered parameters, it will reply with an error status. The reply only contains the new parameters if the status was Adp_ok.

A list of parameter types currently defined:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP_PARAMS_START</td>
<td>0xC000</td>
<td>Base value for parameter names</td>
</tr>
<tr>
<td>AP_BAUD_RATE</td>
<td>0xC000</td>
<td>New baud rate when using serial connection. Must be one of: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400</td>
</tr>
</tbody>
</table>

All Parameter types should have associated semantics which can be represented within one word per parameter, or an associated enum for choices.

Note: There are maximum numbers of parameter types, and parameter options. These values were increased for Angel 1.04 (SDT2.11a):

<table>
<thead>
<tr>
<th>Angel 1.03 and before</th>
<th>Angel 1.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Types</td>
<td>1</td>
</tr>
<tr>
<td>Number of Options per Type</td>
<td>5</td>
</tr>
</tbody>
</table>

In addition, Angel 1.04 is more lenient if these limits are exceeded in a message sent to it; it ignores those items which exceed the limit. Previously, Angel would abort processing of the message, and thus refuse to boot.
2.3.7 ADP_LinkCheck

Channel: CI_HBOOT
Reason Code: 6

Signature:

() = ADP_LinkCheck()

Description:
This should be the first message that the host sends after a successful parameter negotiation. It is really just a ‘ping’. There are no associated parameters, but there must be a reply message.
2.4 Host ADP Channel Messages

2.4.1 ADP_HADPUnrecognised

Channel: CI_HADP
Reason Code: 0
Signature:
   (UINT32 status) = ADP_HADPUnrecognised()
Parameters:
status    Reason code which was not recognised

Description:
This message is unusual in that it is normally sent in reply to another message which
is not understood (the normal protocol states that a reply must have the same base
reason code as the original). There is a single reply parameter; the reason code that
was not understood.
This message can also be sent, and the reply is as if the message were
unrecognised.
2.4.2 ADP_Info

Channel: CI_HADP
Reason Code: 1

Description:
This is the ADP information message. It is used to interrogate the target debug agent, and provides information on the processor as well as the state of the debug world. This allows the host to configure itself to the capabilities of the target.

When the feature set is extended, this can be done in two supported ways:

- If an undivided reason code is to be added with no reason subcodes, such as ADP_Write, add a new ADP_Info sub-code which responds with a flag indicating whether that feature is supported by the target. If this has not been implemented then the reply will be with the ADP_HADPUnrecognised message.

- If a reason code which is subdivided into reason subcodes is added such as Adp_Info, reason subcode 0 should be set aside to indicate whether the functionality of that reason code is supported by the target. If it is not implemented, the reply will be the ADP_HADPUnrecognised message.

The first parameter to ADP_Info is a reason subcode, and subsequent parameters are defined by that subcode. The same reason subcode is always returned as the first UINT32 parameter in the reply message.

2.4.2.1 ADP_Info_NOP

Channel: CI_HADP
Reason Code: 1
Subreason Code: 0

Signature:

```c
VAL Info_NOP IS 0(UINT32):
(Info_NOP, UINT32 status) = ADP_Info(Info_NOP)
```

Parameters:
status Adp_ok for success, non-zero indicates an error

Description:
This message is used to check for ADP_Info being supported. If an error is returned, there is no handler for the ADP_Info message. Normal action is to return an OK status.
2.4.2.2 ADP_Info_Target

Channel: CI_HADP
Reason Code: 1
Subreason Code: 1

Signature:

\text{VAL Info\_Target IS 1(UINT32):}

\text{(Info\_Target, UINT32 status, UINT32 bitset, UINT32 model) =}
\text{ADP\_Info(Info\_Target)}

Parameters:

status
Adp\_ok to indicate OK, or non-zero to indicate some sort of error

bitset
Described in more detail below, and is mostly compatible with that used in Demon to avoid excessive changes to the ARM debugger code

model
The target hardware ID word, as returned by the ADP\_Booted message

Description:

This reason code is used to interrogate target system details.

Note

The minimum and maximum protocol levels provided under Demon are no longer supported. It was a design decision that debugging complexity should be shifted to the host if at all possible. This means that the host debugger should always try to configure itself to the features available in the target debug agent. This can be done by checking individual messages, rather than by a blanket version number dictating the feature set.
Bitset is a word with the following bits defined:

- **1** = Debug agent reloadable
- **0** = Not reloadable
- **1** = Can inquire buffer size
- **0** = Otherwise
- **1** = Thumb state supported
- **0** = Otherwise
- **1** = Profiling supported
- **0** = Otherwise
- **1** = Can inquire buffer size
- **0** = Otherwise
- **Target Speed (IPS):** \(10^{\text{bits0..3}}\)
- **Target Type:**
  - **1** = Hardware
  - **0** = Emulation

### 2.4.2.3 ADP_Info_Points

- **Channel:** CI_HADP
- **Reason Code:** 1
- **Subreason Code:** 2

**Signature:**

\[
\text{VAL Info\_Points IS 2(UINT32):}
\]

\[
(Info\_Points, UINT32 status, UINT32 breakinfo) = ADP\_Info(Info\_Points)
\]

**Parameters:**

- **status:** `Adp_ok` to indicate OK, or non-zero to indicate some sort of error
- **breakinfo:** Described in more detail below
**Description:**
Returns a 32-bit wide bitset of breakpoint and watchpoint features supported by the target debug agent. Breakinfo is a word with the following bits defined:

- **1 = info on break/watch**
- **0 = otherwise**

- **1 = trap on address bitmask**
- **0 = otherwise**

- **1 = trap on 32 bit reads**
- **0 = otherwise**

- **1 = trap on 32 bit writes**
- **0 = otherwise**

- **1 = trap on 8 bit reads**
- **0 = otherwise**

- **1 = trap on 8 bit writes**
- **0 = otherwise**

- **1 = trap on 16 bit reads**
- **0 = otherwise**

- **1 = trap on 16 bit writes**
- **0 = otherwise**

- **1 = thread specific breakpoints**
- **0 = otherwise**

- **1 = thread specific watchpts**
- **0 = otherwise**

- **1 = trap conditional exprs**
- **0 = otherwise**

- **1 = trap address equal**
- **0 = otherwise**

- **1 = trap address range**
- **0 = otherwise**

2.4.2.4 **ADP_Info_Step**

Channel: CI_HADP

Reason Code: 1

Subreason Code: 3

Signature:

\[
\text{VAL Info\_Step IS 3(UINT32) :}
\]

\[
(\text{Info\_Step , UINT32 status, UINT32 stepinfo}) = \text{ADP\_Info(Info\_Step)}
\]

Parameters:

- **status**
  Adp\_ok to indicate OK, or non-zero to indicate some sort of error
**ADP Remote Procedure Calls**

**Description:**

Returns a 32-bit wide bitmask of the single-stepping capabilities of the target debug agent.

```
  1 2 3 4 5 6 7 8 9
   1 2 3 4 5 6 7 8 9

  1 = Can single step
  0 = Otherwise

  1 = Can steps multiple instrs.
  0 = Otherwise

  Unused, set to zero

   [Little Endian]
```

**2.4.2.5 ADP_Info_MMU**

**Channel:** CI_HADP

**Reason Code:** 1

**Subreason Code:** 4

**Signature:**

```
VAL Info_MMU IS 4(UINT32):

  (Info_MMU, UINT32 status, UINT32 meminfo) = ADP_Info(Info_MMU)
```

**Parameters:**

- **status**
  - Adp_ok to indicate OK, or non-zero to indicate some sort of error
- **meminfo**
  - A 32-bit unique ID, or zero if there is no MMU support on the target

**Description:**

Returns information about the memory management system (if any). The only ID code currently defined is 0, indicating that no MMU is active. Please contact ARM for more information on this, as this is likely to change in the near future.
2.4.2.6 ADP_Info_SemiHosting

Channel: CI_HADP
Reason Code: 1
Subreason Code: 5

Signature:

\[
\text{VAL Info\_SemiHosting IS 5(UINT32):}
\]

\[
\text{(Info\_SemiHosting, UINT32 status, UINT32 meminfo) = ADP\_Info(Info\_SemiHosting)}
\]

Parameters:

status
Adp\_ok if semihosting info calls are available, non-zero otherwise

Description:
This message is used to check whether semihosting info calls are available on the target. These calls are described in Section 3 “C Library Remote Procedure Calls”.

2.4.2.7 ADP_Info_CoPro

Channel: CI_HADP
Reason Code: 1
Subreason Code: 6

Signature:

\[
\text{VAL Info\_CoPro IS 6(UINT32):}
\]

\[
\text{(Info\_CoPro, UINT32 status, UINT32 meminfo) = ADP\_Info(Info\_CoPro)}
\]

Parameters:

status
Adp\_ok if CoProcessor info calls are available, non-zero otherwise.

Description:
This message checks whether CoProcessor info calls are supported.
2.4.2.8 ADP_Info_Cycles

Channel: CI_HADP
Reason Code: 1
Subreason Code: 7

Signature:

VAL Info_Cycles IS 7(UINT32): (Info_Cycles, UINT32 status, UINT32 ninstr, scycles, ncycles, icycles, ccycles, fcycles) = ADP_Info(Info_Cycles)

Parameters:

status Adp_ok to indicate success, or non-zero if there is no target support for gathering cycle count information

ninstr The number of instructions executed

scycles The number of S-cycles executed: sequential

ncycles The number of N-cycles executed: nonsequential

icycles The number of I-cycles executed: internal

ccycles The number of C-cycles executed: coprocessor

fcycles The number of F-cycles executed: fast memory

Description:

Returns the number of instructions and cycles executed since the target was initialized. Please refer to the datasheets of the ARM core in use for more precise definitions of these terms.

2.4.2.9 ADP_Info_DescribeCoPro

Channel: CI_HADP
Reason Code: 1
Subreason Code: 8

Signature:

VAL Info_DescribeCoPro IS 8(UINT32): RECORD CoprocessorDesc IS (BYTE copro, rmin, rmax, nbytes, access, r0, r1, w0, w1): (Info_DescribeCoPro, UINT32 status) = ADP_Info(Info_DescribeCoPro, []RECORD CoprocessorDesc desc)
Parameters:

status  Adp_ok to indicate success or non-zero to indicate an error
cpno    The number of the coprocessor to be described
rmin    The bottom of a range of registers with the same description
rmax    The top of a range of registers with the same description
nbytes  The size of the register
access  Describes access to the register and is described in more detail below
r0, r1, w0, w1 See below

Description:

Note  The access and descriptor functions for coprocessors are now depreciated, and may be dropped or radically changed in future revisions of ADP. Any users of these functions should make their use known to the Angel and Debugger groups at ARM.

Describe the registers of a coprocessor. Use only if ADP_Info_CoPro returns Adp_ok. The CoprocessorDesc array is terminated by a single byte (value 0xff) in “copro”, without the rest of that record. The array must fit in a single ADP packet.

The interpretation of the bytes r0, r1, w0, w1 depends on bit 2 in the access byte, as shown in the diagram below. In the case bit 2 = 0, the byte pair r0, r1 us used when reading the coprocessor, and w0, w1 when writing to it. In the case bit 2 = 1, the values of w0, w1 are irrelevant, and r0, r1 are used for both reads and writes.
2.4.2.10 ADP_Info_RequestCoProDesc

Channel: CI_HADP
Reason Code: 1
Subreason Code: 9

Signature:

\[
\text{VAL Info\_ RequestCoProDesc IS 9(UINT32):}
\]
\[
\text{RECORD CoprocessorDesc IS (BYTE copro, rmin, rmax, nbytes, access):}
\]
\[
(\text{Info\_ RequestCoProDesc, UINT32 status, RECORD CoprocessorDesc desc}) =
\]
\[
\text{ADP\_ Info(Info\_ RequestCoProDesc, BYTE cpno)}
\]

Parameters:

status Adp_ok to indicate success or non-zero to indicate an error
cpno The number of the coprocessor to describe
rmin The bottom of a range of registers with the same description
rmax The top of a range of registers with the same description
nbytes The size of the register(s)
access Describes access to the register and is described in more detail in DescribeCoPro, above

Description:

Note The access and descriptor functions for coprocessors are now depreciated, and may be dropped or radically changed in future revisions of ADP. Any users of these functions should make their use known to the Angel and Debugger groups at ARM.

Requests a description of the registers of a coprocessor.

Use only if ADP_Info_CoPro returns Adp_ok.
2.4.2.11 ADP_Info_AngelBufferSize

Channel: CI_HADP
Reason Code: 1
Subreason Code: 10

Signature:

\[
\text{VAL } \text{Info}_\text{AngelBufferSize IS 10(UINT32):}
\]

\[
(\text{Info}_\text{AngelBufferSize}, \text{UINT32 status,UINT32 defaultsize, maxsize}) = \text{ADP}_\text{Info}(\text{Info}_\text{AngelBufferSize})
\]

Parameters:

- status: Adp_ok to indicate success or non-zero to indicate an error.
- defaultsize: The default Angel ADP buffer size in bytes; this must be at least 256 bytes.
- maxsize: The largest Angel ADP buffer size in bytes. This must be greater than or equal to defaultsize. The target will accept ADP messages of up to this length for the ADP_Write packet type.

Description:

This call returns the Angel data packet sizes. This is the amount that the target should transmit in a single operation. This information is also given in the ADP_Booted message.

Note: The value returned should be the size of the data which can be transported in a packet, not the size of the packet itself. This is needed to ensure that the transport protocol information can be wrapped around the data.

Note: Packets larger than defaultsize cannot be used for target-to-host messages with the ARM Host code in SDT2.11a or before. In addition, the same ARM targets can only handle one large packet at a time.
2.4.2.12  ADP_Info_ChangeableSHSWI

Channel: CI_HADP
Reason Code: 1
Subreason Code: 11

Signature:

VAL Info_ChangeableSHSWI IS 11(UINT32):
(Info_ChangeableSHSWI, UINT32 status) = ADP_Info(Info_ChangeableSHSWI)

Parameters:
status  Adp_ok to indicate that it is possible to change which SWI is used
        for semihosting, or non-zero otherwise

Description:
This message is used to check whether it is possible to change which SWI's are used
for semihosting.

2.4.2.13  ADP_Info_CanTargetExecute

Channel: CI_HADP
Reason Code: 1
Subreason Code: 12

Signature:

VAL Info_CanTargetExecute IS 12(UINT32):
(Info_CanTargetExecute, UINT32 status) = ADP_Info(Info_CanTargetExecute)

Parameters:
status  Adp_ok to indicate that the target is not executing application code
        but could do so if asked. Other values indicate why it cannot
        execute.

Description:
This message is used to see if the target is currently in an executable state. Typically
this is called after the debugger initializes. If a adp_ok is returned then the user is
allowed to 'go' immediately.
2.4.2.14 ADP_Info_AgentEndianess

Channel: CI_HADP
Reason Code: 1
Subreason Code: 13

Signature:

\[
\text{VAL Info}_\text{AgentEndianess IS 13(UINT32):} \\
(\text{Info}_\text{AgentEndianess, UINT32 status}) = \text{ADP}_\text{Info}(\text{Info}_\text{AgentEndianess})
\]

Parameters:

status See below

Description:

This message is used to determine the endianess of the debug agent (as opposed to the target). There could be a difference in the case that the debug agent is running on a distinct processor from the target (for example, with an EmbeddedICE unit). Status should be RDIError_LittleEndian or RDIError_BigEndian; any other value indicates the target does not support this request so the debugger will have to make a best guess.

2.4.2.15 ADP_Info_CanAckHeartbeat

Channel: CI_HADP
Reason Code: 1
Subreason Code: 14

Signature:

\[
\text{VAL Info}_\text{CanAckHeartbeat IS 14(UINT32):} \\
(\text{Info}_\text{CanAckHeartbeat, UINT32 status}) = \text{ADP}_\text{Info}(\text{Info}_\text{CanAckHeartbeat})
\]

Parameters:

status Adp_ok to indicate heartbeats are acknowledged, non-zero otherwise

Description:

This message checks whether heartbeat messages sent from the host are acknowledged (by sending a heartbeat message to the host). Heartbeat acknowledgements are part of the channel level protocol, but were not implemented in the versions of Angel prior to 1.03 (EmbeddedICE 2.03).
ADP Remote Procedure Calls

If the target supports heartbeats but they are not enabled by default, this call will enable them as well. Angel 1.04 and EmbeddedICE 2.07 support heartbeats by default; the call simply returns adp_ok.

2.4.3  ADP_Control

Channel: \text{CI\_HADP}
Reason Code: 2
Description:
This message allows for the state of the debug agent to be manipulated by the host. The following are subreason codes to ADP control; the first parameter is the subreason code which defines the format of subsequent parameters.

2.4.3.1  ADP\_Ctrl\_NOP

Channel: \text{CI\_HADP}
Reason Code: 2
Subreason Code: 0
Signature:
\begin{verbatim}
VAL Ctrl\_NOP IS 0(UINT32):
\end{verbatim}
\begin{verbatim}
(Ctrl\_NOP, UINT32 status) = ADP\_Control(Ctrl\_NOP)
\end{verbatim}
Parameters:
status Adp\_ok to indicate if ADP\_Control messages are supported, non-zero otherwise
Description:
This message is used to check that ADP\_Ctrl messages are supported.
2.4.3.2 ADP_Ctrl_VectorCatch

Channel: CI_HADP

Reason Code: 2
Subreason Code: 1

Signature:

\[
\text{VAL Ctrl\_VectorCatch IS 1(UINT32):}
\]

\[
(\text{Ctrl\_VectorCatch, UINT32 status}) = \text{ADP\_Control(Ctrl\_VectorCatch, UINT32 bitmap)}
\]

Parameters:

status Adp_ok if the target has completed the request and will report the requested exceptions.

bitmap A bit mask of exceptions to be reported, described in more detail below.

Description:

Specifies which hardware exceptions should be reported to the debugger; a set bit in 'bitmap' indicates that the exception should be reported to the debugger, a clear bit indicates that the corresponding exception vector should be taken.
2.4.3.3 ADP_Ctrl_PointStatus_Watch

Channel: CI_HADP
Reason Code: 2
Subreason Code: 2

Signature:

\[
\text{VAL Ctrl\_PointStatus\_Watch IS 2(UINT32):}
\]
\[
(Ctrl\_PointStatus\_Watch, UINT32 status, hwresource, type) =
\]
\[
\text{ADP\_Control(Ctrl\_PointStatus\_Watch, UINT32 handle)}
\]

Parameters:
handle A handle to a watchpoint
status Adp_ok to indicate success or non-zero to indicate an error
hwresource The hardware resource number
type The type of the resource
Description:
Returns the hardware resource number and the type of that resource when given a watchpoint handle. Should only be called if the value returned by ADP_Info_Points had bit 12, "info on break/watch" set.

2.4.3.4  ADP_Ctrl_PointStatus_Break

Channel: CI_HADP
Reason Code: 2
Subreason Code: 3

Signature:

\[
\text{VAL Ctrl\_PointStatus\_Break IS 3(UINT32):} \\
\quad (\text{Ctrl\_PointStatus\_Break, UINT32 status, hwresource, type}) = \\
\quad \text{ADP\_Control(Ctrl\_PointStatus\_Break, UINT32 handle)}
\]

Parameters:
handle A handle to a breakpoint
status Adp_ok to indicate success or non-zero to indicate an error
hwresource The hardware resource number
type The type of the resource

Description:
Returns the hardware resource number and the type of that resource when given a breakpoint handle. Should only be called if the value returned by ADP_Info_Points had bit 12, "info on break/watch" set.

2.4.3.5  ADP_Ctrl_SemiHosting_SetState

Channel: CI_HADP
Reason Code: 2
Subreason Code: 4

Signature:

\[
\text{VAL Ctrl\_SemiHosting\_SetState IS 4(UINT32):} \\
\quad (\text{Ctrl\_SemiHosting\_SetState, UINT32 status}) = \\
\quad \text{ADP\_Control(Ctrl\_SemiHosting\_SetState, UINT32 semihostingstate)}
\]
**ADP Remote Procedure Calls**

**Parameters:**
- semihostingstate: If zero, semihosting is enabled, otherwise it is disabled
- status: Adp_ok to indicate success or non-zero to indicate an error

**Description:**
Sets whether or not semihosting is enabled. If semihosting is disabled, the debug agent must treat semihosting calls (for example, to open a file) as undefined exceptions.

Disabling semihosting does not imply disabling all support for SWI calls from the application: in Angel, the ReportException and EnterSVC calls are still permitted (indeed, Angel requires EnterSVC for its own operation).

This should only be called if ADP_Info_SemiHosting returns a status of Adp_ok.

**2.4.3.6 ADP_Ctrl_SemiHosting_GetState**

**Channel:** CI_HADP

**Reason Code:** 2

**Subreason Code:** 5

**Signature:**

\[
\text{VAL Ctrl\_SemiHosting\_GetState IS 5(UINT32):} \\
\text{(Ctrl\_SemiHosting\_GetState, UINT32 status, UINT32 semihostingstate) =} \\
\text{ADP\_Control(Ctrl\_SemiHosting\_GetState)}
\]

**Parameters:**
- semihostingstate: If zero, semihosting is enabled, otherwise it is disabled
- status: Adp_ok to indicate success, or non-zero to indicate an error

**Description:**
Reads whether or not semihosting is enabled.

**Note**
This should only be called if ADP_Info_SemiHosting returns a status of Adp_ok.
2.4.3.7 ADP_Ctrl_SemiHosting_SetVector

Channel: CI_HADP
Reason Code: 2
Subreason Code: 6

Signature:

\[
\text{VAL Ctrl\_SemiHosting\_SetVector IS 6(UINT32):}
\]
\[
(Ctrl\_SemiHosting\_SetVector, UINT32 status) =
\]
\[
ADP\_Control(Ctrl\_SemiHosting\_SetVector, UINT32 semihostingvector)
\]

Parameters:

- semihostingvector: The value to which the semihosting vector is to be set.
- status: Adp\_ok to indicate success, or non-zero to indicate an error

Description:
Sets the semihosting vector. The normal vector to use would be the SWI vector, at address 8.

**Note** This should only be called if ADP\_Info\_SemiHosting returns a status of Adp\_ok.

2.4.3.8 ADP_Ctrl_SemiHosting_GetVector

Channel: CI_HADP
Reason Code: 2
Subreason Code: 7

Signature:

\[
\text{VAL Ctrl\_SemiHosting\_GetVector IS 7(UINT32):}
\]
\[
(Ctrl\_SemiHosting\_GetVector, UINT32 status, UINT32 semihostingvector) =
\]
\[
ADP\_Control(Ctrl\_SemiHosting\_GetVector)
\]

Parameters:

- semihostingvector: The current value of the semihosting vector
- status: Adp\_ok to indicate success, or non-zero to indicate an error

Description:
Gets the value of the semi-hosting vector.
2.4.3.9 ADP_Ctrl_Log

Channel: CI_HADP
Reason Code: 2
Subreason Code: 8
Signature:

\[
\text{VAL Ctrl_Log IS 8(UINT32):}
\]

\[
(Ctrl\_Log, \text{UINT32 status, UINT32 logbits}) = ADP\_Control(Ctrl\_Log)
\]

Parameters:
logbits A bitmap specifying the current level of protocol logging
status Adp_ok to indicate success, or non-zero to indicate an error

Description:
Returns the logging state for code running on the host.'logsetting' is a bitmap specifying the level of logging desired. The state is a combination of the following bits.

![Diagram of bit combinations](image)

Unused, set to zero
1 = Log Internal operations
0 = Otherwise

1 = Log ADP packets by byte
0 = Otherwise

1 = Log RDI Level operations
0 = Otherwise
2.4.4 ADP_Ctrl_SetLog

Channel: CI_HADP
Reason Code: 2
Subreason Code: 9
Signature:

\[
\text{VAL Ctrl_Log IS 9(UINT32):}
\]
\[
(Ctrl_Log, UINT32 status) = \text{ADP_Control(Ctrl_Log, UINT32 logbits)}
\]
Parameters:
logbits A bitmap specifying the current level of protocol logging. See ADP_Ctrl_Log above for details
status Adp_ok to indicate success or non-zero to indicate an error
Description:
Sets the logging state for code running on the host. See above for details of the bits which can be set.

Note The ARM debuggers do not issue the set log command until several packets into the boot sequence; if the default log state for the host is to log nothing, then no logging information will be added to the log for the open sequence and initial packet transfers.

2.4.4.1 ADP_Ctrl_SemiHosting_SetARMSWI

Channel: CI_HADP
Reason Code: 2
Subreason Code: 10
Signature:

\[
\text{VAL Ctrl_SemiHosting_SetARMSWI IS 10(UINT32):}
\]
\[
(Ctrl_SemiHosting_SetARMSWI, UINT32 status) = \text{ADP_Control(Ctrl_SemiHosting_SetARMSWI, UINT32 swinumber)}
\]
Parameters:
swinumber The new SWI number
status Adp_ok to indicate success, or non-zero to indicate an error
ADP Remote Procedure Calls

Description:
Sets the number of the ARM SWI used for semihosting. The debug agent will interpret ARM SWI's with the SWI number specified as semihosting SWI's.
The Angel libraries supplied with the toolkit, and the Angel ROM's installed on ARM Development Boards, use SWI number 0x123456 in ARM code and 0xAB in Thumb code.

Note  This should only be called if ADP_Info_ChangeableSHSWI does not return an error.

2.4.4.2 ADP_Ctrl_SemiHosting_GetARMSWI
Channel: CI_HADP
Reason Code: 2
Subreason Code: 11
Signature:

VAL Ctrl_SemiHosting_GetARMSWI IS 11(UINT32):
(Ctrl_SemiHosting_GetARMSWI, UINT32 status, UINT32 swinumber) =
ADP_Control(Ctrl_SemiHosting_GetARMSWI)

Parameters:
swinumber The current SWI number
status Adp_ok to indicate success or non-zero to indicate an error

Description:
Returns the number of the ARM SWI used for semihosting. This should only be called if ADP_Info_SemiHosting returns a status of Adp_ok.

2.4.4.3 ADP_Ctrl_SemiHosting_SetThumbSWI
Channel: CI_HADP
Reason Code: 2
Subreason Code: 12
Signature:

VAL Ctrl_SemiHosting_SetThumbSWI IS 12(UINT32):
(Ctrl_SemiHosting_SetThumbSWI, UINT32 status) =
ADP_Control(Ctrl_SemiHosting_SetThumbSWI, UINT32 swinumber)
Parameters:
swinumber The current SWI number.
status Adp_ok to indicate success, or non-zero to indicate an error

Description:
The debug agent will interpret Thumb SWI's with the SWI number specified as
semihosting SWI's.
The Angel libraries supplied with the toolkit, and the Angel ROM's installed on ARM
Development Boards, use SWI number 0x123456 in ARM code and 0xAB in Thumb
code.

Note This should only be called if ADP_Info_ChangeableSHSWI returns a status of
Adp_ok.

2.4.4.4 ADP_Ctrl_SemiHosting_GetThumbSWI

Channel: CI_HADP
Reason Code: 2
Subreason Code: 13

Signature:
VAL Ctrl_SemiHosting_GetThumbSWI IS 13(UINT32):
(Ctrl_SemiHosting_GetThumbSWI, UINT32 status, UINT32 swinumber) =
ADP_Control(Ctrl_SemiHosting_GetThumbSWI)

Parameters:
swinumber The current SWI number
status Adp_ok to indicate success or non-zero to indicate an error

Description:
Reads the number of the Thumb SWI used for semihosting.

Note This should only be called if ADP_Info_SemiHosting returns a status of Adp_ok.
2.4.4.5 ADP_Ctrl_Download_Supported

Channel: CI_HADP
Reason Code: 2
Subreason Code: 14

Signature:

\[
\text{VAL Ctrl\_Download\_Supported IS 14(UINT32):} \\
(Ctrl\_Download\_Supported, UINT32 status) = \\
\text{ADP\_Control(Ctrl\_Download\_Supported)}
\]

Parameters:
status \(\text{Adp\_ok}\) to indicate that download is possible or non-zero otherwise.

Description:
Asks whether a new Debug Agent can be downloaded. Use ADP_Ctrl_Download_Agent to initiate the actual download.

2.4.4.6 ADP_Ctrl_Download_Data

Channel: CI_HADP
Reason Code: 2
Subreason Code: 15

Signature:

\[
\text{VAL Ctrl\_Download\_Data IS 15(UINT32):} \\
(Ctrl\_Download\_Data, UINT32 status) = \\
\text{ADP\_Control(Ctrl\_Download\_Data, UINT32 nbytes, [nbytes]BYTE data)}
\]

Parameters:
status \(\text{Adp\_ok}\) to indicate that download is possible or non-zero otherwise
nbytes The number of bytes of configuration data
data The configuration data

Description:
Loads configuration data to the debug agent. The message follows ADP_Ctrl_Download_Agent.

Note
\(\text{nbytes must not cause the packet size to exceed the standard packet size (256 bytes).}\)
Should only be used if ADP_ICEM_AddConfig didn't return an error.
2.4.4.7 ADP_Ctrl_Download_Agent

Channel: CI_HADP  
Reason Code: 2  
Subreason Code: 16  

Signature:

\[
\text{VAL Ctrl_Download_Data IS 16(UINT32):} \\
\quad (\text{Ctrl_Download_Data, UINT32 status}) = \\
\quad \text{ADP_Control(Ctrl_Download_Agent, UINT32 loadadress, UINT32 size)}
\]

Parameters:

- loadaddress: The address where the new debug agent code should be loaded.
- size: The number of bytes of debug agent code to be loaded.
- status: Adp_ok to indicate success, non-zero otherwise.

Description:
Prepares debug agent to receive configuration data which it should interpret as a new version of the debug agent code.
The message follows ADP_Ctrl_Download_Supported. The data will be downloaded using one or more ADP_Ctrl_Download_Data messages. The new agent is started with ADP_Ctrl_Start_Agent.

2.4.4.8 ADP_Ctrl_Start_Agent

Channel: CI_HADP  
Reason Code: 2  
Subreason Code: 17  

Signature:

\[
\text{VAL Ctrl_Download_Data IS 17(UINT32):} \\
\quad (\text{Ctrl_Download_Data, UINT32 status}) = \\
\quad \text{ADP_Control(Ctrl_Download_Agent, UINT32 startaddress)}
\]

Parameters:

- startaddress: the address where the new debug agent should begin execution.
- status: Adp_ok to indicate success, non-zero otherwise.
ADP Remote Procedure Calls

Description:
Instruct the debug agent to begin execution of new agent, which has previously been
downloaded by ADP_Ctrl_Download_Agent and ADP_Ctrl_Download_Data.
The startaddress must satisfy the condition:
(loadaddress <= startaddress <= (loadaddress + size))
where loadaddress and size were specified in the ADP_Ctrl_Download_Agent
message.

Note: Immediately after sending the host should reinitialise it’s buffers and sequence
numbers and wait for the target to boot. There should be a timeout on this waiting
period of at least 2 seconds.

2.4.4.9  ADP_Ctrl_SetTopMem

Channel: CI_HADP
Reason Code: 2
Subreason Code: 18
Signature:
VAL Ctrl_SetTopMem IS 18(UINT32):
( Ctrl_SetTopMem, UINT32 status) =
ADP_Control(Ctrl_SetTopMem, UINT32 memtop)

Parameters:
memtop The address where the new debug agent should begin execution
status Adp_ok to indicate success, non-zero otherwise

Description:
Sets the top of target memory in systems where the debug agent is distinct from the
target, and thus doesn’t know the target memory layout. This enables the C Library to
allocate the stack and heap in the correct place on startup (via the SYS_HEAPINFO
system calls).
This request is supported by EmbeddedICE, and not by debug agents running on the
target which should return an error such as HADPUncrecognised.
2.4.5 ADP_Read

Channel: CI_HADP
Reason Code: 3

Signature:

\[(UINT32\ status, \ mbytes, \ [mbytes]BYTE\ data) = ADP_Read(INT32address, \ UINT32\ nbytes)\]

Parameters:
- address: The target address from which memory transfer should start
- nbytes: The number of bytes to transfer
- status: Adp_ok to indicate success, non-zero otherwise
- mbytes: Holds the number of requested bytes not read (i.e., zero indicates success, non-zero a partial success or failure)
- data: The data requested, less "mbytes" at the end

Description:
This is a request by the host for a transfer of memory contents from the target to the debugger. The debug agent is free to perform this transfer in whatever way it chooses. That is, it should not be assumed that the transfer will use target byte or half-word writes, even when reading single bytes or half words. However, it should stop reading data as soon as it can if a data abort occurs.

The call returns the number of bytes which the debug agent has definitely not read, rounded up so that the read data is definitely correct. It is possible that the agent may have actually read more data, but is not able to determine how much (for example, it was reading the data using LDM instructions).

Transfers of 1, 2 and 4 bytes should be written to the destination memory as a single byte, short and word quantity, architecture permitting.

2.4.6 ADP_Write

Channel: CI_HADP
Reason Code: 4

Signature:

\[(UINT32\ status, \ mbytes) = ADP_Write(INT32address, \ UINT32\ nbytes, \ [nbytes]BYTE\ data)\]
ADP Remote Procedure Calls

Parameters:
- **address**: The target address from which memory transfer should start
- **nbytes**: The number of bytes to transfer
- **status**: Adp_ok to indicate success, non-zero otherwise
- **mbytes**: Holds the number of requested bytes not written (that is, zero indicates success, non-zero a partial success or failure). Note that this field is only present if status is not adp_ok.
- **data**: The bytes of data to write.

Description:
This is a request by the host for a transfer of memory contents from the debugger to the target. The debug agent is free to perform this transfer in whatever way it chooses. That is, it should not be assumed that the transfer will use target byte or half-word writes, even when writing single bytes or half words. However, it should stop writing data as soon as it can if a data abort occurs.

The call returns the number of bytes which the debug agent has definitely not written, rounded up to the unit of data transfer. It is possible that the agent may have actually written more data, but is not able to determine how much (for example, it was writing the data using STM instructions).

Transfers of 1, 2 and 4 bytes should be written to the destination memory as a single byte, short and word quantity, architecture permitting.

2.4.7 ADP_CPUread

**Channel**: CI_HADP

**Reason Code**: 5

**Signature**:

\[(\text{UINT32 } \text{status}, [\text{BYTE } \text{data}]) = \text{ADP_CPUread}(\text{BYTE } \text{mode}, \text{UINT32 } \text{mask})\]

**Parameters**:
- **mode**: Defines the processor mode from which the transfer should be made. It is described in more detail below.
- **mask**: Indicates which registers should be transferred. Setting a bit to one will cause the designated register to be transferred. The details of mask are specified below.
- **status**: Adp_ok to indicate success, non-zero otherwise.
data  Holds the values of the registers on successful completion. The lowest numbered register is transferred first. Note that data must not cause the buffer size to exceed the maximum allowed buffer size.

Description:
This is a request to read values in the CPU for the application task when it was last running. These values must be saved as part of the execution interrupt sequence (caused either by a breakpoint, single step, unhandled exception, application exit or debugger interrupt request).

The mode number is the same as the mode number used by an ARM CPU core (in the PSR) with the exception that a value of 0xFF indicates the current mode. It is valid to make multiple CPUread requests, although for efficiency this should be kept to a minimum.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADP_CPUmode_Current</td>
<td>0xFF</td>
<td>Used to specify the current mode.</td>
</tr>
<tr>
<td>ADP_CPUread_26bitUser</td>
<td>0x0</td>
<td>26-bit User mode.</td>
</tr>
<tr>
<td>ADP_CPUread_26bitFIQ</td>
<td>0x1</td>
<td>26-bit FIQ mode.</td>
</tr>
<tr>
<td>ADP_CPUread_26bitIRQ</td>
<td>0x2</td>
<td>26-bit IRQ mode.</td>
</tr>
<tr>
<td>ADP_CPUread_26bitSVC</td>
<td>0x3</td>
<td>26-bit Supervisor mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitUser</td>
<td>0x10</td>
<td>32-bit User mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitFIQ</td>
<td>0x11</td>
<td>32-bit FIQ mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitIRQ</td>
<td>0x12</td>
<td>32-bit IRQ mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitSVC</td>
<td>0x13</td>
<td>32-bit Supervisor mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitAbort</td>
<td>0x17</td>
<td>32-bit Abort mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitUndef</td>
<td>0x1B</td>
<td>32-bit Undefined mode.</td>
</tr>
<tr>
<td>ADP_CPUread_32bitSystem</td>
<td>0x1F</td>
<td>32-bit System mode.</td>
</tr>
</tbody>
</table>

Note  32-bit System mode was added in Architecture 4 ARM CPU cores eg.ARM7TDMI. Some CPU cores (e.g. StrongARM) do not implement the 26 bit modes.
The bitmask is defined as follows:

```
18-31 18 17 16 15 0-14
```

Unused, set to zero

SPSR

CPSR

PC, without mode/flags in 26 bit state

PC, with mode/flags in 26 bit state

Register map:
bit0 is r0, bit14 is r14

2.4.8 ADP_CPUwrite

Channel: CI_HADP
Reason Code: 6
Signature:

\[(UINT32\ status) = ADP_CPUwrite(BYTE\ mode, UINT32\ mask, [BYTE\ data])\]

Parameters:

- **mode**
  - Defines the processor mode from which the transfer should be made. It is described in more detail in ADP_CPUread.

- **mask**
  - Indicates which registers should be transferred. Setting a bit to one will cause the designated register to be transferred. The details of mask are specified below.

- **status**
  - Adp_ok to indicate success, non-zero otherwise.

- **data**
  - Holds the values of the registers on successful completion. The lowest numbered register is transferred first. Note that data must not cause the buffer size to exceed the maximum allowed buffer size.
ADP Remote Procedure Calls

Description:
This is a request to write values to the CPU register block for the application task. When the debugger next requests the application execute (using ADP_Execute), these values will be loaded into the CPU registers.

If made, multiple CPUwrite requests are cumulative. It is not defined what state the application task registers are in when the target is initialised; the debugger should write all registers which the application is likely to care about.

2.4.9 ADP_CPread

Channel: CI_HADP
Reason Code: 7

Signature:

(UINT32 status, []BYTE data) = ADP_CPread(BYTE cpno, UINT32 mask)

Parameters:

- cpno: The number of the co-processor to transfer values from.
- mask: Specifies which registers to transfer and is co-processor specific.
- status: Adp_ok to indicate success, non-zero otherwise.
- data: holds the registers specified in 'mask' if successful, otherwise just rubbish. The lowest numbered register is transferred first.

Description:
This message requests a controller's internal state. The only coprocessor currently defined by ARM is the system coprocessor, in which the bits of 'mask' relate directly to the registers of the coprocessor, starting with the least significant bit being register 0.

The semantics of reading coprocessor register values are as for reading the CPU registers in ADP_CPUread; the request reads the registers as they were when the application stopped.

Note: The size of the 'data' array must not cause the buffer size to exceed the maximum allowed buffer size.

1 Angel 1.04 initialises the CPSR to USR mode. Future versions of Angel may initialise the CPSR to SVC mode instead. Interrupts are disabled.

2 If the debug agent knows that the coprocessor state cannot change as a result of its activity, it can optimise application context switching by reading from and writing to the coprocessor directly.
2.4.10 ADP_CPwrite

Channel: CI_HADP
Reason Code: 8

Signature:

(UINT32 status) = ADP_CPread(BYTE cpno, UINT32 mask, JBYTE data)

Parameters:
- cpno: The number of the coprocessor to transfer values to.
- mask: Specifies which registers to transfer and is coprocessor specific.
- status: Adp_ok to indicate success, non-zero otherwise.
- data: Holds the registers specified in ‘mask’ if successful. The lowest numbered register is transferred first.

Description:
This message requests a write to a coprocessors internal state. See the comments for ADP_Cpread for the coprocessor definition.

The semantics of writing coprocessor register values are as for writing the CPU registers in ADP_CPUwrite; the request writes (cumulatively) to a store of the registers which will be restored to the coprocessor when the application is (re)started.

Note: The size of the ‘data’ array must not cause the buffer size to exceed the maximum allowed buffer size.

2.4.11 ADP_SetBreak

Channel: CI_HADP
Reason Code: 9

Signature:

(UINT32 status, pointhandle, raddress, rbound) = ADP_SetBreak(UINT32 address, BYTE type, UINT32 bound)

Parameters:
- address: The address of the instruction to set the breakpoint on.
- type: Specifies the sort of breakpoint and is described in more detail below.
- bound: Only present in the message if the least significant 4 bits of ‘type’ are set to 5 or above.
status: Adp_ok to indicate success, non-zero otherwise.
pointhandle: Returns a handle to the breakpoint, it will be valid if bit 7 of 'type' is set.
raddress: Valid depending on 'type' (see below, always included).
rbound: Valid depending on 'type' (see below, always included).

Description:
Sets a breakpoint at the specified address. If the debug agent allows multiple application threads and the requests DebugID is set to a valid thread identifier, it should selectively place the breakpoint only when that thread is running.

The interpretation of “type” is as shown in the figure below. Bits 5, 6 and 7 are used as follows:

- Bit 5 set indicates that the breakpoint is on a 16-bit (Thumb) instruction rather than a 32-bit (ARM) instruction.
- Bit 6 set indicates that the breakpoint should be conditional (execution halts only if the breakpointed instruction is executed, not if it is conditionally skipped). If not set, execution halts whenever the breakpointed instruction is reached (whether executed or skipped).
- Bit 7 set requests a dry run: the breakpoint is not set, and the ‘raddress’, and if appropriate the ‘rbound’, that would be used, are returned (for comparison and range breakpoints the address and bound used need not be exactly as requested).

A returned adp_ok status indicates that resources are currently available to set the breakpoint, non-zero indicates an error, while the error code RDIError_NoMorePoints indicates that the required breakpoint resources are not currently available.

Note: It is possible that a NoMorePoints error could occur because watchpoints are using the required resources.
If a breakpoint is set on a location which already has a breakpoint, the first breakpoint will be removed before the new breakpoint is set.

2.4.12 ADP_ClearBreak

**Channel:** CI_HADP

**Reason Code:** 10

**Signature:**

\[(UINT32 \text{ status}) = \text{ADP\_ClearBreak}(UINT32 \text{ pointhandle})\]

**Parameters:**

- status: Adp\_ok to indicate success, non-zero otherwise.
- pointhandle: A handle to the breakpoint returned from a previous call to ADP\_SetBreak.

**Description:**

Clears a breakpoint, releasing whatever resources it was using. Once completed, 'pointhandle' becomes invalid.
2.4.13 ADP_SetWatch

Channel: CI_HADP
Reason Code: 11

Signature:

\[(UINT32 \text{ status}, UINT32 \text{ pointhandle}, raddress, rbound) = ADP\_SetWatch(UINT32 \text{ address}, BYTE \text{ type}, BYTE \text{ datatype}, UINT32 \text{ bound})\]

Parameters:
- address: The address at which to set the watchpoint.
- type: The type of watchpoint to set and is described in detail below.
- datatype: Defines the sort of data access to watch for and is described in more detail below.
- bound: Included depending on the value of type (see description of type below).
- status: Adp_ok to indicate success, non-zero otherwise.
- pointhandle: Valid depending on the value of type (see description of type below).
- raddress: Valid depending on the value of type (see description of type below).
- rbound: Valid depending on the value of type (see description of type below).

Description:

This call sets a watchpoint. If the debug agent allows multiple application threads and the requests DebugID is set to a valid thread identifier, it should place the watchpoint only when that thread is running.

The type byte is interpreted as shown in the figure below. Bits 6 and 7 are used as follows:

- Bit 6 set requests a dry run: the watchpoint is not set, and the 'address' and, if appropriate, the 'bound', that would be used are returned (for range and comparison watchpoints, the 'address' and 'bound' used need not be exactly as requested). A adp_ok status indicates that resources are currently available to set the watchpoint; RDIError_NoMorePoints indicates that the required watchpoint resources are not currently available.
- Bit 7 set requests that a handle should be returned for the watchpoint by which it will be identified subsequently. If bit 7 is set, a handle will be returned (‘pointhandle’), whether or not the request succeeds or fails (but, obviously, it will only be meaningful if the request succeeds).

Note Bits 6 and 7 must not be simultaneously set. Note also that the 'dry run' bit is in a different place in the byte from that used for breakpoints.
The 'type' argument is laid out as shown:

```
| 7 | 6 | 5 | 0-3 |
```

- 1 = Dry run only
- 0 = Set watch

Unused, set to zero

Value | Condition
--- | ---
0 | ea == addr
1 | ea > addr
2 | ea >= addr
3 | ea < addr
4 | ea <= addr
5 | addr <= ea <= bound
6 | ea < addr \(\land\) ea > bound
7 | (ea & bound) == addr

ea is the effective address of a data ref.

The 'datatype' argument defines the sort of data access to watch for:

```
| 6-7 | 5 | 4 | 3 | 2 | 1 | 0 |
```

1 = 32 bit Writes
0 = Ignore

1 = 16 bit Writes
0 = Ignore

1 = 8 bit Writes
0 = Ignore

1 = 32 bit Reads
0 = Ignore

1 = 16 bit Reads
0 = Ignore

1 = 8 bit Reads
0 = Ignore

Unused, set to zero
ADP Remote Procedure Calls

On successful completion an Adp_ok status byte is returned. If the request fails, Angel returns the value RDIError_NoMorePoints if there are no more watchpoint registers of the requested type, or a non-zero error code byte.

If a watchpoint is set on a location which already has a watchpoint, the first watchpoint will be removed before the new watchpoint is set.

2.4.14 ADP_ClearWatch

Channel: CI_HADP
Reason Code: 12
Signature:

\[(UINT32 \text{status}) = \text{ADP\_ClearWatch}(UINT32 \text{pointhandle})\]

Parameters:
status Adp_ok to indicate success, non-zero otherwise.
pointhandle A handle to the watchpoint returned from a previous call to ADP_SetWatch.

Description:
Clears a watchpoint.

2.4.15 ADP_Execute

Channel: CI_HADP
Reason Code: 13
Signature:

\[(UINT32 \text{status}) = \text{ADP\_Execute}()\]

Parameters:
status Adp_ok to indicate success, non-zero otherwise.

Description:
This message requests that the target starts executing from the stored CPU state (set with ADP_CPUwrite and possibly ADP_CPwrite). The target must respond immediately with the ADP_Execute reply message before initiating execution.
ADP Remote Procedure Calls

Execution will stop when allowed system events occur. This will be one of:

1. The application terminated normally and returned control to the debugger
2. The application caused an exception (e.g., a data abort) which it provided no handler for
3. The debugger requested the debug agent to interrupt the application
4. The application hit a predefined watchpoint or breakpoint

The host will be notified that the application has stopped executing via a ADP_Stopped message (described below), even in the case of an interrupt request from the debugger. The CPU registers and, if appropriate the coprocessor state, at the point the application stopped should be stored ready for examination by the debugger via subsequent calls to ADP_CPUread and ADP_CPUread.

Note: Execution may cause the processor to enter any legal CPU state, and may cause the processor to enter illegal states as well (such as the PC or SP pointing at aborting memory, the CPSR specifying an undefined mode). The effect of these situations should be considered carefully.

2.4.16 ADP_Step

Channel: CI_HADP
Reason Code: 14
Signature:

(UINT32 status) = ADP_Step(UINT32 ninstr)

Parameters:
status Adp_ok to indicate success, non-zero otherwise.
ninstr The number of instructions to execute.

Description:
Execute ‘ninstr’ instructions.
The number of instructions to execute should be interpreted starting at the address currently loaded into the CPU program counter. If ‘ninstr’ is zero, the target should execute instructions up to the next instruction that explicitly alters the Program Counter. i.e., a branch or ALU operation with the PC as the destination.
If the debug agent only supports stepping by one instruction at a time, it is permissible to return the HADP_Unrecognised reply when ‘ninstr’ is non-zero. In this case however the debug agent must return 0 in the ‘can step multiple instructions’ bit for an ADP_Info_Step.

Angel Debug Protocol Messages

Open Access
The ADP_STEP function will always return an acknowledging reply immediately. A subsequent ADP_STOPPED message will be delivered from the target to the host when the ADP_STEP operation has completed.

See ADP_Execute for comments on how the application can return control to the debug agent.

### 2.4.17 ADP_INTERRUPTREQUEST

**Channel:** CI_HADP  
**Reason Code:** 15  
**Signature:**  
\[(UINT32\ \text{status}) = \text{ADP_INTERRUPTREQUEST}()\]  
**Parameters:**  
status \hspace{1cm} \text{Adp}\_ok \text{ to indicate success, non-zero otherwise}  
**Description:**  
This requests the debug agent interrupt execution of the target. It must only be called between an ADP_Execute or ADP_STEP request and the corresponding ADP_STOPPED message from the target.

Once execution of the target program has halted, the reply message (type ADP_INTERRUPTREQUEST) to the interrupt must be sent. Some time after this, the target must send an ADP_STOPPED message which indicates that execution has completed.

**Note**  
It is possible that a debug agent might receive an ADP_INTERRUPTREQUEST after sending an ADP_STOPPED message resulting from a target breakpoint or exception. If this happens, the InterruptRequest should just return the error RDIErr_Error_NoError.

### 2.4.18 ADP_HW_EMULATION

**Channel:** CI_HADP  
**Reason Code:** 16  
**Description:**  
The first parameter to ADP_HW_EMULATION is a Reason Subcode, and subsequent parameters are defined by that subcode.
2.4.18.1 ADP_HW_Emul_Supported

Channel: CI_HADP
Reason Code: 16
Subreason Code: 0

Signature:

\[ \text{VAL HWEm\_Supported IS 1(UINT32):} \]
\[ (HWEm\_Supported, UINT32 \text{ status}) = \text{ADP\_HW\_Emulation}(HWEm\_Supported) \]

Parameters:
status \quad \text{Adp\_ok to indicate that the messages are available, non-zero otherwise.}

Description:
This request enquires whether the four messages MemoryAccess, MemoryMap, Set_CPUspeed and ReadClock are understood.

2.4.18.2 ADP_HW_Emul_MemoryAccess

Channel: CI_HADP
Reason Code: 16
Subreason Code: 1

Signature:

\[ \text{VAL HWEm\_MemoryAccess IS 0(UINT32):} \]
\[ (HWEm\_MemoryAccess, UINT32 \text{ status}, \]
\[ \text{UINT32 nreads, nwrites, sreads, swrites, nsecs, secs}) = \]
\[ \text{ADP\_HW\_Emulation}(HWEm\_MemoryAccess, UINT32 handle) \]

Parameters:
handle \quad \text{A handle to a memory block.}
status \quad \text{Adp\_ok to indicate success, non-zero otherwise.}
nreads \quad \text{The number of non-sequential reads.}
wwrites \quad \text{The number of non-sequential writes.}
sreads \quad \text{The number of sequential reads.}
swrites \quad \text{The number of sequential writes.}
nsecs \quad \text{Time in nano seconds.}
secs  Time in seconds.

Description:
Get memory access information for memory block with specified handle.

2.4.19 ADP_HW_Emul_MemoryMap

Channel: CI_HADP
Reason Code: 16
Subreason Code: 2

Signature:

VAL HWEm_MemoryMap IS 2(UINT32):
RECORDRegionInfo IS (UINT32 handle, start, limit, BYTE width, access,
UINT32 nread_ns, nwrite_ns, sread_ns, swrite_ns):

(HWEm_MemoryMap, UINT32 status) =
ADP_HW_Emulation(HWEm_MemoryMap,
UINT32 nregions, [nregions]RECORD RegionInfo regions )

Parameters:
nregions The number of memory regions in the memory map.
handle A handle to the region.
start The start of this region.
limit The end (limit) of this region.
width The memory width, described in detail below.
access Described in detail below.
nread_ns The access time for non-sequential read cycles in nano seconds.
nwrite_ns The access time for non-sequential write cycles in nano seconds.
sread_ns The access time for sequential read cycles in nano seconds.
swrite_ns The access time for sequential write cycles in nano seconds.
status Adp_ok to indicate success, non-zero otherwise.

Description:
Sets memory characteristics for the given area of memory.

Width should be one of:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryMap_Width8</td>
<td>0</td>
<td>8 bit memory width.</td>
</tr>
</tbody>
</table>
ADP Remote Procedure Calls

<table>
<thead>
<tr>
<th>MemoryMap_Width16</th>
<th>1</th>
<th>16 bit memory width.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryMap_Width32</td>
<td>2</td>
<td>32 bit memory width.</td>
</tr>
</tbody>
</table>

Access is some combination of:

![Memory Access Diagram]

2.4.19.1 ADP_HW_Emul_SetCPUSpeed

Channel: CI_HADP
Reason Code: 16
Subreason Code: 3

Signature:

\[
\text{VAL HWEm\_SetCPUSpeed IS 3(UINT32):}\]
\[
\text{(HWEm\_SetCPUSpeed, UINT32 status) =}\]
\[
\text{ADP\_HW\_Emulation(HWEm\_SetCPUSpeed, UINT32 speed)}\]

Parameters:

- speed: The speed of the CPU core in nano-seconds per clock cycle.
- status: Adp_ok to indicate success, non-zero otherwise.

Description:

Sets the speed of the CPU, for use when returning clock time.
2.4.19.2 ADP_HW_Emul_ReadClock

Channel: CI_HADP
Reason Code: 16
Subreason Code: 4

Signature:

VAL HWEm_ReadClock IS 4(UINT32):
    (HWEm_ReadClock, UINT32 status, UINT32 ns, s) =
    ADP_HW_Emulation(HWEm_ReadClock)

Parameters:
ns The time in nano-seconds.
s The time in seconds.
status Adp_ok to indicate success, non-zero otherwise.

Description:
Reads the time in seconds and nano-seconds since some arbitrary starting point.

2.4.20 ADP_ICEbreakerHADP

Channel: CI_HADP
Reason Code: 17

Description:
The first parameter to ADP_ICEbreaker is a Reason Subcode, and subsequent parameters are defined by that subcode.

2.4.20.1 ADP_ICEB_Exists

Channel: CI_HADP
Reason Code: 17
Subreason Code: 0

Signature:

VAL ICEB_Exists IS 0(UINT32):
    (ICEB_Exists, UINT32 status) = ADP_ICEbreakerHADP(ICEB_Exists)
Parameters:
status  Adp_ok to indicate there is an Embedded ICE unit (or equivalent), non-zero otherwise.

Description:
Determines if the target debug agent is using the ARM hardware EmbeddedICE debug unit or not, and thus whether the ADP_ICEB calls are implemented.

2.4.20.2 ADP_ICEB_GetLocks

Channel:  CI_HADP
Reason Code:  17
Subreason Code:  1
Signature:
VAL ICEB_GetLocks IS 1(UINT32):
(ICEB_GetLocks, UINT32 status, locked) =
ADP_ICEbreakerHADP(ICEB_GetLocks)

Parameters:
status  Adp_ok to indicate success, non-zero otherwise.
locked  A bitmap of the Embedded ICE registers locked against use by the debug agent (because they are being explicitly written by the user).

Description:
Returns which Embedded ICE registers are locked. Bit n of the bitmap ‘locked’ represents hardware breakpoint unit n, and if set the unit is locked.
This request may be used in conjunction with ADP_ICEB_SetLocks by a user or by the debug agent to reserve breakpoint units against use by the debugger.

Note  Should only be used if ADP_ICEB_Exists does not return an error.
2.4.20.3 ADP_ICEB_SetLocks

Channel: CI_HADP
Reason Code: 17
Subreason Code: 2

Signature:

\[
\text{VAL ICEB\_SetLocks IS 2(UINT32):}
\]
\[
(\text{ICEB\_SetLocks, UINT32 status}) = \text{ADP\_ICEbreakerHADP(ICEB\_SetLocks, UINT32 locked)}
\]

Parameters:

status \hspace{1em} Adp\_ok to indicate success, non-zero otherwise.
locked \hspace{1em} A bitmap of the Embedded ICE breakpoint registers

Description:

Sets which Embedded ICE registers are locked. Bit n of the bitmap 'locked' represents hardware breakpoint unit n, and if set the unit is to be locked against use by the debug agent.

Note Should only be used if ADP_ICEB_Exists does not return an error.

2.4.20.4 ADP_ICEB_CC_Exists

Channel: CI_HADP
Reason Code: 17
Subreason Code: 3

Signature:

\[
\text{VAL ICEB\_CC\_Exists IS 3(UINT32):}
\]
\[
(\text{ICEB\_CC\_Exists, UINT32 status}) = \text{ADP\_ICEbreakerHADP(ICEB\_CC\_Exists)}
\]

Parameters:

status \hspace{1em} Adp\_ok to indicate there is a Debug Comms Channel, non-zero otherwise.

Description:

Asks whether there is an EmbeddedICE Debug Comms Channel (DCC).

Note Should only be used if ADP_ICEB_Exists does not return an error.
2.4.20.5 ADP_ICEB_CC_Connect_ToHost

Channel: CI_HADP
Reason Code: 17
Subreason Code: 4

Signature:
```
VAL ICEB_CC_Connect_ToHost IS 4(UINT32):
  (ICEB_CC_Connect_ToHost, UINT32 status) =
    ADP_ICEbreakerHADP(ICEB_CC_Connect_ToHost, BYTE connect)
```

Parameters:
- status: Adp_ok to indicate there is a Comms Channel, non-zero otherwise.
- connect: 1 or 0; controls transmission of messages from EmbeddedIce to the host.

Description:
Connect Comms Channel in ToHost direction. If ‘connect’ is true, the debug agent must enable transmission of data read from the DCC into the debug agent on to the host over the TDCC channel. If it is false, it should disable such transmission.

Note: Should only be used if ADP_ICEB_CC_Exits does not return an error.

2.4.20.6 ADP_ICEB_CC_Connect_FromHost

Channel: CI_HADP
Reason Code: 17
Subreason Code: 5

Signature:
```
VAL ICEB_CC_Connect_FromHost IS 5(UINT32):
  (ICEB_CC_Connect_FromHost, UINT32 status) =
    ADP_ICEbreakerHADP(ICEB_CC_Connect_FromHost, BYTE connect)
```

Parameters:
- status: Adp_ok to indicate there is a Comms Channel, non-zero otherwise.
- connect: 1 or 0; controls transmission of messages from EmbeddedIce to the host.
Description:
Connect Comms Channel in FromHost direction. If ‘connect’ is true, the debug agent must enable forwarding of data from the host TDCC channel to the target over DCC. If it is false, it should disable such transmission.

Note  Should only be used if ADP_ICEB_CC_Exists does not return an error.

2.4.21 ADP_ICEman

Channel: CI_HADP
Reason Code: 18
Description:
The first parameter to ADP_ICEman is a reason subcode, and subsequent parameters are defined by that subcode.
The requests allow the manipulation of “configurations” held within the EmbeddedICE unit. A configuration is used to describe the ARM CPU which the unit will expect to see on the end of the JTAG link. In current ARM EmbeddedICE units, two types are defined: ARM7DI and ARM7TDMI.
The configuration block format is not published.

2.4.21.1 ADP_ICEM_AddConfig

Channel: CI_HADP
Reason Code: 18
Subreason Code: 0
Signature:
VAL ICEM_AddConfig IS 0(UINT32):

(ICEM_AddConfig, UINT32 status) =
ADP_ICEman(ICEM_AddConfig, UINT32 nbytes)

Parameters:
status  Adp_ok to indicate success, non-zero otherwise.
nbytes  The number of bytes in the complete configuration block.
ADP Remote Procedure Calls

Description:
Prepares target to receive configuration data block. The data must follow in blocks described by the ADP_Ctrl_DownloadData request; it is complete when the length of data read in the DownloadData blocks equals the ‘nbytes’ value provided in the AddConfig request.

2.4.21.2 ADP_ICEM_SelectConfig

Channel: CI_HADP
Reason Code: 18
Subreason Code: 1

Signature:

VAL ICEM_SelectConfig IS 1(UINT32):

(ICEM_SelectConfig, UINT32 status, UINT32 version_sel) =

ADP_ICEman(ICEM_SelectConfig, BYTE aspect, namelen, matchtype,

UINT32 version_req, [namelen]BYTE name)

Parameters:
aspect One of two values defined below.
namelen The number of bytes in 'name'.
matchtype Specifies how the selected version must match that specified, and takes one of the values defined below.
version_req The requested version of the named configuration.
name The name of the configuration.
status Adp_ok to indicate success, non-zero otherwise.
version_sel The version number of the configuration selected on success.

Description:
Selects one of the sets of configuration data blocks and reinitialises to use the new configuration. Aspect defines what the scope of the configuration data is, and is one of the following constants:\(^1\):

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure CPU</td>
<td>0</td>
</tr>
</tbody>
</table>

\(^1\) Current ARM EmbeddedICE units do not implement the Configure System request.
Configure System 1

Matchtype defines whether any version of the configuration is acceptable, or whether a particular revision is needed. Version numbers are compared numerically using unsigned comparison. Select one of:

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match Any</td>
<td>0</td>
</tr>
<tr>
<td>Match Exactly</td>
<td>1</td>
</tr>
<tr>
<td>Match No Earlier</td>
<td>2</td>
</tr>
</tbody>
</table>

The version actually selected is returned.

2.4.21.3 ADP_ICEM_ConfigCount

Channel: CI_HADP
Reason Code: 18
Subreason Code: 2

Signature:

\[
VAL \text{ ICEM\_ConfigCount IS } 2(\text{UINT32}):
\]

\[
(\text{ICEM\_ConfigCount, UINT32 status, UINT32 count}) = \text{ADP\_ICEman(ICEM\_ConfigCount)}
\]

Parameters:
- **count**: Returns the number of configurations if status is zero.
- **status**: Adp\_ok to indicate success, non-zero otherwise.

Description:
Return number of known ICEman configurations. The details can be retrieved with the ADP_ICEM_ConfigNth request.
2.4.21.4  ADP_ICEM_ConfigNth

Channel:  CI_HADP
Reason Code:  18
Subreason Code:  3

Signature:

VAL ICEM_ConfigNth IS 3(UINT32):

(ICEM_ConfigNth, UINT32 status, UINT32 version, BYTE namelen, [namelen]BYTE name) =

ADP_ICEman(ICEM_ConfigNth, UINT32 confignum)

Parameters:

confignum  The number of the configuration, starting at 1.
status  Adp_ok to indicate success, non-zero otherwise.
version  The configuration version number.
namelen  The number of bytes in 'name'.
name  The name of the configuration.

Description:

Gets the name of the nth processor configuration. See ICEM_ConfigCount to determine the number of available configurations and thus the range of confignum.

2.4.22  ADP_Profile

Channel:  CI_HADP
Reason Code:  19

Description:

The first parameter to ADP_Profile is a reason subcode, and subsequent parameters are defined by that subcode.

The Profiling codes are used to control the code profiling abilities of the debug agent. A possible sequence of events would be:

if (profile_supported())
  profile_writemap()
  profile_start()
  <<wait for event>>
  profile_stop()
2.4.22.1 ADP_Profile_Supported

Channel: CI_HADP
Reason Code: 19
Subreason Code: 0

Signature:
VAL Profile_Supported IS 0(UINT32):
(Profile_Supported, UINT32 status) = ADP_Profile(Profile_Supported)

Parameters:
status Adp_ok to indicate profiling is supported, non-zero otherwise.

Description:
Checks whether profiling is supported.

Note: Can also be determined using Info_Target.

2.4.22.2 ADP_Profile_Stop

Channel: CI_HADP
Reason Code: 19
Subreason Code: 1

Signature:
VAL Profile_Stop IS 1(UINT32):
(Profile_Stop, UINT32 status) = ADP_Profile(Profile_Stop)

Parameters:
status Adp_ok to indicate profiling has been stopped, non-zero otherwise.

Description:
Stops collecting application profile data, leaving the results in the count array intact.
2.4.22.3 ADP_Profile_Start

Channel: CI_HADP
Reason Code: 19
Subreason Code: 2

Signature:

\[
\text{VAL Profile\_Start IS 2(UINT32):}\\
\quad (\text{Profile\_Start, UINT32 status}) = \text{ADP\_Profile(Profile\_Start, UINT32 interval)}
\]

Parameters:

status \hspace{1cm} \text{Adp\_ok to indicate profiling is supported, non-zero otherwise.}
interval \hspace{1cm} \text{The period of PC sampling in microseconds.}

Description:

Starts profiling. The map of PC interval values must have been written to the debug agent first using the ADP_Profile_WriteMap request.

2.4.22.4 ADP_Profile_WriteMap

Channel: CI_HADP
Reason Code: 19
Subreason Code: 3

Signature:

\[
\text{VAL Profile\_WriteMap IS 3(UINT32):}\\
\quad (\text{Profile\_WriteMap, UINT32 status}) = \text{ADP\_Profile(Profile\_WriteMap, UINT32 len, size, offset, [UINT32 map\_data])}
\]

Parameters:

len \hspace{1cm} \text{The number of elements in the entire map array being downloaded.}
size \hspace{1cm} \text{The number of words being downloaded in this message, ie. the length of map\_data.}
offset The offset into the entire map array which this message starts from, in words.

map_data Consists of size words of map data.

status Adp_ok to indicate success, non-zero otherwise.

**Description:**
Downloads (part of) a PC interval map array, which describes the instruction address ranges for profiling. The whole profile map will be sent as a number of WriteMap messages.

Each map entry represents the PC value for the start of a code block to be profiled; the end address is assumed to be just before the next start address. There should therefore be an extra entry to mark the end of the last routine in a block.

**Note**
The map as a whole must be in ascending address order.

Once the address map entries map[x] to map[x+n] are written, the equivalent counts c[x] to c[x+n] are zeroed. It is thus possible to:

1. selectively zero counts in the map (ADP_Profile_ClearCounts will clear all count values)
2. selectively add to or change the profile map, on the fly if necessary.

During application execution, if at some time (map[x] <= PC < map[x+1]) where PC represents the address of an executed instruction at that time, the value c[x] is incremented.

The count array can be zeroed as a whole with ADP_Profile_ClearCounts and read by the host with ADP_Profile_ReadMap.
2.4.22.5 ADP_Profile_ReadMap

Channel: CI_HADP
Reason Code: 19
Subreason Code: 4

Signature:

\[
\begin{align*}
\text{VAL Profile\_ReadMap IS 4(UINT32):} \\
(\text{Profile\_ReadMap, UINT32 status, [size]UINT32 counts}) &= \\
\text{ADP\_Profile(Profile\_ReadMap, UINT32 offset, size)}
\end{align*}
\]

Parameters:
- offset: The offset in the profile data array to begin reading, in words.
- size: The number of words to read (\(\leq 54\)).
- status: Adp\_ok to indicate success, non-zero otherwise.
- counts: The size words of profile counts.

Description:
Uploads a set of profile counts to the host. The number of words transferred must not cause the message size to exceed 256 bytes, which limits the number of words to 54. The offsets within the returned array correspond to the current profile map, as loaded into the target with ADP\_Profile\_WriteMap.

2.4.22.6 ADP_Profile_ClearCounts

Channel: CI_HADP
Reason Code: 19
Subreason Code: 5

Signature:

\[
\begin{align*}
\text{VAL Profile\_ClearCounts IS 5(UINT32):} \\
(\text{Profile\_ClearCounts, UINT32 status}) &= \text{ADP\_Profile(Profile\_ClearCounts)}
\end{align*}
\]

Parameters:
- status: Adp\_ok to indicate success, non-zero otherwise.
Description:
Requests that profiling sample counts be set to zero. This would typically be used to start a new profiling session.

2.4.23 ADP_InitialiseApplication

Channel: CI_HADP
Reason Code: 20
Signature:

\[(UINT32\ status) = ADP_InitialiseApplication()\]

Parameters:
status Adp_ok to indicate success, non-zero otherwise.

Description:
Requests the debug agent or target operating system to set up the “application” (typically a task or thread) so that it can be executed.

2.4.24 ADP_End

Channel: CI_HADP
Reason Code: 21
Signature:

\[(UINT32\ status) = ADP_End()\]

Parameters:
status Adp_ok to indicate success, non-zero otherwise.

Description:
Sent by the host to tell the target debug agent this debugging session is finished. The debug agent should free any session-related resources and make ready to resume with a new session.

Note: After ADP_End has been processed, the ARM debuggers (Angel, EmbeddedICE and Remote_A) explicitly reset the communications link back to default values, but only where there are negotiated parameters to change. Consequently, connections using Serial or Serial/Parallel devices where the baud rate chosen is not 9600 baud finish with ADP_End, ParameterNegotiate and LinkCheck, while those using the Ethernet device always finish with just ADP_End.
2.4.25 ADP_ReadExt [ADP 1.1 Only]

Channel: CI_HADP
Reason Code: 22

Signature:

\[(\text{UINT32 } \text{status}, \text{ mbytes}, [\text{mbytes}]\text{BYTE } \text{data}) = \]

\[\text{ADP\_Read(}\text{INT32 } \text{address, UINT32 } \text{nbytes, UINT32 } \text{access})\]

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>The target address from which memory transfer should start</td>
</tr>
<tr>
<td>nbytes</td>
<td>The number of bytes to transfer</td>
</tr>
<tr>
<td>status</td>
<td>Adp_ok to indicate success, non-zero otherwise</td>
</tr>
<tr>
<td>mbytes</td>
<td>Holds the number of requested bytes not read (ie. zero indicates success, non-zero a partial success or failure)</td>
</tr>
<tr>
<td>data</td>
<td>The data requested, less &quot;mbytes&quot; at the end</td>
</tr>
<tr>
<td>access</td>
<td>The memory access method, from the following table:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbolic Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADP_Data</td>
<td>0</td>
<td>Access a range of data memory of arbitrary size: the debugger does not mind what granularity of access is used.</td>
</tr>
<tr>
<td>ADP_Data8</td>
<td>1</td>
<td>Access data 1 byte at a time, using LDRB or equivalent.</td>
</tr>
<tr>
<td>ADP_Data16</td>
<td>2</td>
<td>Access data 2 bytes at a time, using LDRH or equivalent.</td>
</tr>
<tr>
<td>ADP_Data32</td>
<td>3</td>
<td>Access data 4 bytes at a time, using LDR or equivalent.</td>
</tr>
<tr>
<td>ADP_Data64</td>
<td>4</td>
<td>Access data 8 bytes at a time.</td>
</tr>
<tr>
<td>ADP_Code</td>
<td>8</td>
<td>Access a range of code memory of arbitrary size: the debugger does not mind what granularity of access is used.</td>
</tr>
<tr>
<td>ADP_Code8</td>
<td>9</td>
<td>Access code 1 byte at a time, using LDRB or equivalent.</td>
</tr>
<tr>
<td>ADP_Code16</td>
<td>10</td>
<td>Access code 2 bytes at a time, using LDRH or equivalent.</td>
</tr>
<tr>
<td>ADP_Code32</td>
<td>11</td>
<td>Access code 4 bytes at a time, using LDR or equivalent.</td>
</tr>
</tbody>
</table>
Description:
This is a request by the host for a transfer of memory contents from the target to the debugger.
See ADP_Read for general comments.
This message extends ADP_Read by specifying the method by which the agent is to perform the transfer.
If the data cannot be transferred in the method specified, the error code RDIError_UnimplementedSize must be returned. If access is not one of the values listed in the table above, the error code RDIError_UnimplementedType must be returned.
It may assumed that if ADP_ReadExt is implemented, then ADP_WriteExt is also, and vice versa.
If a target does not implement this call (for example, because it is an ADP1.0 target) then it must return RDIError_UnimplementedMessage [this is the standard return status for any unrecognised call]. Hence, a debugger which wishes to write memory may call ADP_ReadExt with the details, and if it gets back the error RDIError_UnimplementedMessage it should default to using ADP_Read with the same parameters. Once such an error is returned, it can be assumed to last for the complete session.

2.4.26 ADP_WriteExt [ADP 1.1 Only]

Channel: CI_HADP
Reason Code: 4
Signature:
\[(UINT32\ status, mbytes) = ADP_Write(INT32\ address, UINT32\ nbytes, [nbytes]BYTE\ data)\]
Parameters:
address The target address from which memory transfer should start
nbytes The number of bytes to transfer
status Adp_ok to indicate success, non-zero otherwise
mbytes Holds the number of requested bytes not written (that is, zero indicates success, non-zero a partial success or failure). Note that this field is only present if status is not adp_ok.
data The bytes of data to write.
access The memory access method, from the table given in ADP_ReadExt.
Description:
This is a request by the host for a transfer of memory contents from the debugger to the target.

See ADP_Write for general comments. This message extends ADP_Write by specifying the method by which the agent is to perform the transfer.

If the data cannot be transferred in the method specified, the error code RDIError_UnimplementedSize must be returned. If access is not one of the values listed in the table above, the error code RDIError_UnimplementedType must be returned.

It may be assumed that if ADP_WriteExt is implemented, then ADP_ReadExt is also, and vice versa.

If a target does not implement this call (for example, because it is an ADP1.0 target) then it must return RDIError_UnimplementedMessage [this is the standard return status for any unrecognised call]. Hence, a debugger which wishes to write memory may call ADP_WriteExt with the details, and if it gets back the error RDIError_UnimplementedMessage it should default to using ADP_Write with the same parameters. Once such an error is returned, it can be assumed to last for the complete session.
2.5 Target ADP Channel Messages

2.5.1 ADP_TADPUncognised

Channel: CI_TADP
Reason Code: 0
Signature:

\[(UINT32 \text{ code}) = \text{ADP_TADPUncognised()}\]

Parameters:

code Reason code which was not recognised.

Description:
This message is unusual in that it is normally sent in reply to another message which is not understood. This is an exception to the normal protocol that a reply must have the same base reason code as the original. There is a single reply parameter which is the reason code which was not understood.

As well as being a reply, this message can also be sent and will return as if this message were unrecognised.

2.5.2 ADP_Stopped

Channel: CI_TADP
Reason Code: 1
Signature:

\[(UINT32 \text{ status}) = \text{ADP_Stopped} (UINT32 \text{ reason}, \ldots)\]

Parameters:

status Adp_ok to indicate success, non-zero otherwise.

Description:
This message is sent to the host when the application stops.

Execution may stop when the end of the application has been reached, or as the result of an exception. It can also be the return from an ADP_Step process, when the requested number of instructions have been executed, or a breakpoint or watchpoint has been hit etc.
The first set of stopped subreason codes are for the ARM hardware vectors. These events will be raised if the ADP_Control_Vector_Catch has been set to enable this (and the exception can be caught).

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stopped_BranchThroughZero</td>
<td>0</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_UndefinedInstr</td>
<td>1</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_SoftwareInterrupt</td>
<td>2</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_PrefetchAbort</td>
<td>3</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_DataAbort</td>
<td>4</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_AddressException</td>
<td>5</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_IRQ</td>
<td>6</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_FIQ</td>
<td>7</td>
<td>Vector Exceptions</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_BreakPoint</td>
<td>32</td>
<td>Breakpoint was reached</td>
<td>UINT32 Breakpoint handle</td>
</tr>
<tr>
<td>Stopped_Watchpoint</td>
<td>33</td>
<td>Watchpoint was reached</td>
<td>UINT32 Watchpoint handle</td>
</tr>
<tr>
<td>Stopped_StepComplete</td>
<td>34</td>
<td>End of ADP_Step</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_RunTimeErrorUnknown</td>
<td>35</td>
<td>Non-specific fatal runtime support error</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_InternalError</td>
<td>36</td>
<td>Angel internal error</td>
<td>UINT32 Error code</td>
</tr>
<tr>
<td>Stopped_UserInterruption</td>
<td>37</td>
<td>Host requested interruption</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_ApplicationExit</td>
<td>38</td>
<td>Application has exited via exit(), a code of zero indicates successful termination</td>
<td>UINT32 exit() code</td>
</tr>
<tr>
<td>Stopped_StackOverflow</td>
<td>39</td>
<td>Software stack overflow</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_DivisionByZero</td>
<td>40</td>
<td>Division by zero</td>
<td>None</td>
</tr>
<tr>
<td>Stopped_OSSpecific</td>
<td>41</td>
<td>OS requested that execution stops.</td>
<td>None</td>
</tr>
</tbody>
</table>
2.6 Target Debug Comms Channel Messages

2.6.1 ADP_TDCC_ToHost

Channel: CI_TTDCC
Reason Code: 0
Signature:

\[(UINT32 \text{ status}) = \text{ADP_TDCC\_ToHost}(UINT32 \text{ nbytes}, [\text{nbytes}/4]UINT32 \text{ data})\]

Parameters:
- nbytes: Number of bytes to be transferred from the target to the host via the Debug Comms channel.
- data: Data to be transferred from the target to the host via the Debug Comms channel.
- status: Adp\_ok to indicate success, non-zero otherwise.

Description:
Transfer data by reading the processor Debug Communications Channel and returning those word(s) to the host. The data being will be in target endianness.

Note: Current implementations only support single word transfers (nbytes = 4).

2.6.2 ADP_TDCC_FromHost

Channel: CI_TTDCC
Reason Code: 1
Signature:

\[(UINT32 \text{ status}, UINT32 \text{ nbytes}, [\text{nbytes}/4]UINT32 \text{ data}) = \text{ADP_TDCC\_FromHost()}\]

Parameters:
- nbytes: Number of bytes to be transferred from the target to the host via the Debug Comms channel.
- data: Data to be transferred from the target to the host via the Debug Comms channel.
- status: Adp\_ok to indicate success, non-zero otherwise.
ADP Remote Procedure Calls

Description:
Transfer data using the processor Debug Communications Channel from the host to the target. The data being sent must be in the correct endianness for the target processor.

Note  Current implementations only support single word transfers (nbytes = 4).
This section describes the C support library remote procedure calls. These calls mirror the semihosting SWI calls for the semihosting functions which the target debug agent is unable to perform.
3.1 Function Signatures and Packet Formation

These are as described for ADP RPC. It should be noted that C library packets have never had the channel number encoded in the reason word.

3.2 Target to Host Messages

3.2.1 CL_Unrecognised

Channel: CI_CLIB
Reason Code: 0

Signature: (UINT32 code) = CL_Unrecognised()

Parameters: code The reason code which was not understood.

Description: This message is used as a response to a packet whose message was not understood. The return parameter, code, is the reason code which was not understood.

3.2.2 CL_WriteC

Channel: CI_CLIB
Reason Code: 1

Signature: (UINT32 status) = CL_WriteC(BYTE char)

Parameters: char The character to write.
status Adp_ok if character write successful.

Description: This message writes a single character to the debugger’s console window, or similar, on the host. The character is encoded in ASCII. Some systems assume that CL_WriteC and CL_Write0 write to the same device or file.
3.2.3 CL_Write0

Channel: CI_CLIB
Reason Code: 2
Signature:

\[(UINT32\ status) = CL\_Write0(UINT32\ nbytes, [nbytes+1]BYTE\ string)\]

Parameters:

- **string**: The string of characters to write.
- **nbytes**: The number of bytes in ‘string’ excluding the null terminator.
- **status**: adp_ok if write successful.

Description:
This message writes a null terminated string to the debugger’s console window, or similar, on the host. The number of bytes in the supplied string is nbytes+1, as nbytes is the number of bytes to write. The text is encoded in ASCII.

Some systems assume that CL_WriteC and CL_Write0 write to the same device or file.

3.2.4 CL_ReadC

Channel: CI_CLIB
Reason Code: 4
Signature:

\[(UINT32\ status, BYTE\ char) = CL\_ReadC()\]

Parameters:

- **status**: Adp_ok if read successful.
- **char**: The character which was read.

Description:
This message reads a single character from the host’s keyboard as an ASCII encoded value.
3.2.5 CL_System

Channel: CI_CLIB
Reason Code: 5
Signature:

\[(\text{UINT32 code, BYTE exitcode}) = \text{CL_System}(\text{UINT32 nbytes, [nbytes+1]BYTE command})\]

Parameters:
- nbytes: The length of the command string, excluding the null termination byte.
- command: The command string to pass to the host's command interpreter. Null terminated.
- exitcode: The exit code returned by the host OS from running 'command'.
- char: The character which was read.

Description:
This message invokes the host's command line interpreter in a manner similar to the ANSI function `system()`. The command string is null terminated.

3.2.6 CL_GetCmdLine

Channel: CI_CLIB
Reason Code: 16
Signature:

\[(\text{UINT32 status, UINT32 nbytes, [nbytes+1]BYTE args}) = \text{CL_GetCmdLine()}\]

Parameters:
- status: Adp_ok if read successful
- nbytes: The length of the command argument string, excluding the null termination byte.
- args: The command line arguments used to invoke the program. Null terminated.

Description:
This message reads the null terminated string used to invoke the program. No interpretation is performed on this string. The command string is null terminated.
3.2.7 CL_Clock

Channel: CI_CLIB
Reason Code: 97
Signature:

\[(\text{UINT32} \text{ status}, \text{UINT32} \text{ ticks}) = \text{CL_Clock()}\]

Parameters:
status Adp_ok if read successful
ticks The number of clock ticks, in centiseconds.

Description
This request returns the number of centiseconds since the support code’s process was started on the host. Only the difference between calls has any meaning, and it should be remembered that over slower communications links, a significant variance can be expected in the results due to packet transmission delays.

3.2.8 CL_Time

Channel: CI_CLIB
Reason Code: 99
Signature:

\[(\text{UINT32} \text{ status}, \text{UINT32} \text{ time}) = \text{CL_Time()}\]

Parameters:
status Adp_ok if read successful.
time The time in seconds since 00:00 on the 1 Jan 1970.

Description:
This request returns the time of day, in seconds since the start of January 1\textsuperscript{st} 1970; this is the same as the standard Unix \texttt{time()} function. It should be remembered that over slower communications links, a variance can be expected in the results due to packet transmission delays.

Whether the value can exceed \(2^{31}\) seconds depends on the implementation of the host C library \texttt{time()} function; if it cannot, the value will wrap around during the year 2038.
3.2.9  CL_Remove

Channel: CI_CLIB
Reason Code: 100

Signature:

\[(UINT32 \text{ status}) = \text{CL\_Remove}(UINT32 \text{ nbytes}, [\text{nbytes}+1]\text{BYTE filename})\]

Parameters:

- status: Adp\_ok if read successful.
- filename: The name of the file to delete. Null terminated.
- nbytes: The length of filename, excluding the null termination.

Description:

This request deletes the file with the given name from the host’s file system.

The null terminated file name is not interpreted in any way, but is limited in length by the ADP packet size.

3.2.10  CL_Rename

Channel: CI_CLIB
Reason Code: 101

Signature:

\[(UINT32 \text{ status}) = \text{CL\_Rename}(UINT32 \text{ nbytes1}, [\text{nbytes1}+1]\text{BYTE filename1},
\text{UINT32} \text{ nbytes2}, [\text{nbytes2}+1]\text{BYTE filename2})\]

Parameters:

- status: Adp\_ok if read successful
- filename1: The old name of the file. Null terminated.
- nbytes1: The length of filename1, excluding the null termination.
- nbytes2: The length of filename2, excluding the null termination.

Description:

This requests the host to rename filename1 to have the name filename2. Both filenames must be contained within a single ADP packet.
3.2.11 CL_Open

Channel: CI_CLIB
Reason Code: 102

Signature:

\[(\text{UINT32 handle}) = \]
\[\text{CL_Open(\text{UINT32 nbytes}, [nbytes+1]BYTE filename, \text{UINT32 mode})}\]

Parameters:

- handle: The file handle which will be used in subsequent file operations, or zero if the open failed.
- mode: The open mode; see below
- nbytes: The length of filename, excluding the null termination
- filename: The name of the file to open. Null terminated.

Description:

This message attempts to open the named file in the fashion requested by `mode'.

The name "tt" is used to reference the standard input or standard output streams on the host (e.g. in "C", the value of "stdin" or "stdout". Which of the streams is opened is dependent on the value of "mode". Implementations for which there is no standard input or output should either treat the name as a file or return a failure code.

Note

The ARM C library will try to open "tt" three times to generate it's stdin, stdout and stderr streams on application startup.

The open mode is one of the values shown below. Other values are undefined:

<table>
<thead>
<tr>
<th>Mode:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI fopen():</td>
<td>r</td>
<td>rb</td>
<td>r</td>
<td>r+</td>
<td>r+b</td>
<td>w</td>
<td>w+</td>
<td>w+b</td>
<td>a</td>
<td>ab</td>
<td>a+</td>
<td>a+b</td>
</tr>
</tbody>
</table>

The file handle is an opaque integer which is used by the host to identify the file in subsequent read, write etc. operations.
3.2.12 CL_Close

Channel: CI_CLIB
Reason Code: 104
Signature:

\[(UINT32\ status) = CL\_Close(UINT32\ handle)\]

Parameters:

- status: Adp_ok if close successful.
- handle: The file handle to close.

Description:
This message closes a file opened with CL_Open above. The 'handle' should be non-zero. The close operation can fail; the result should always be checked.

3.2.13 CL_Write

Channel: CI_CLIB
Reason Code: 105
Signature:

\[(UINT32\ nwritten) = CL\_Write(UINT32\ handle, UINT32\ nbtotal, nbytes, [nbytes]BYTE\ data)\]

Parameters:

- status: Adp_ok if close successful
- nbtotal: The total number of bytes to write.
- nbytes: The number of bytes in this packet.
- data: The data to write to the file.
- handle: An open, writable file handle.
- nwritten: The number of bytes not written to the file.

Description:
This message writes a block of data to the file identified by handle. If the number of bytes to write exceeds the maximum packet size, a number of CL_WriteX packets will follow this packet. If nbtotal is less than the space left in the packet, then nbtotal == nbytes and nwritten is the number of bytes not written to the file (i.e. nwritten == 0 is a successful write). Otherwise, the return code from CL_Write can be ignored.
C Support Library Remote Procedure Calls

The data should be written as a single block of ‘nbtotal’ bytes wherever possible even when the data is transmitted using CL_WriteX packets.

Note: Both CL_Write and CL_WriteX packets expect acknowledgements from the host for every packet sent.

3.2.14 CL_WriteX

Channel: CI_CLIB
Reason Code: 106
Signature:

\[(\text{UINT32 status}) = \text{CL}\_WriteX(\text{UINT32 nbytes, [nbytes]BYTE data})\]

Parameters:

- nbytes: The number of bytes in this packet.
- data: The data to write to the file.
- nwritten: The number of bytes not written to the file.
- status: Adp_ok if close successful.

Description:

This message continues a write request started with a CL_Write; it is an invalid operation to make this call at any other time than after a CL_Write or CL_WriteX.

If this is the last packet of the write (ie. the accumulated value of nbytes from previous CL_Write and CL_WriteX calls, is equal to the value of nbtotal in the original CL_Write) then the return value nwritten is the number of bytes not written to the file. Otherwise, the value can be ignored.

3.2.15 CL_Read

Channel: CI_CLIB
Reason Code: 107
Signature:

\[(\text{UINT32 status, nread, nbmore, [nread]BYTE data}) = \text{CL}\_Read(\text{UINT32 handle, UINT32 nbytes})\]
C Support Library Remote Procedure Calls

Parameters:
status Adp_ok if close successful.
nread The number of bytes read in this packet.
 nbmore The number of bytes which are following in CL_ReadX packets.
 handle An open, readable file handle.
data The data read in this packet.

Description:
This message reads nbytes bytes of data from the file handle provided. If the number of bytes to read is less than the space left in the packet, then the reply packet will contain the bytes read, nread will be this number and nbmore will be zero. Otherwise, nread will be the number of bytes returnable in this reply packet, nbmore will be the number of bytes left to return, and the replies to following CL_ReadX packets will contain the rest of the data.
The reader should never send a CL_Read request unless there is data waiting to be sent, as indicated by a previous CL_Read or CL_ReadX reply.

3.2.16 CL_ReadX

Channel: CI_CLIB
Reason Code: 108
Signature:
(UINT32 status, nread, nbmore, [nread]BYTE data) = CL_ReadX()

Parameters:
nread the number of bytes read in this packet.
nbmore the number of bytes which are following in CL_ReadX packets.
data the data read in this packet
status adp_ok if close successful

Description:
This message continues a read request started with a CL_Read; it is an invalid operation to make this call at any other time than after a CL_Read or CL_ReadX.
If this is the last packet of the read (ie. the value of nbmore from the last CL_Read or CL_ReadX call is zero) then the return value nwritten is the number of bytes not written to the file. Otherwise, the value can be ignored.
3.2.17  CL_Seek

Channel:       CL_CLIB
Reason Code:   109
Signature:
   (UINT32 status) = CL_Seek(UINT32 handle, UINT32 position)
Parameters:
status          Adp_ok if close successful.
handle          An open, seekable file handle.
position        The offset in bytes from the start of the file represented by handle.
Description:
This message positions the file's read/write pointer to 'position' bytes from the start of
the file. If the file is shorter than this, or if the position when considered as a signed
number is negative, the behavior is host operating system dependant.

3.2.18  CL_FLen

Channel:       CL_CLIB
Reason Code:   110
Signature:
   (UINT32 status, len) = CL_FLen(UINT32 handle)
Parameters:
status          Adp_ok if close successful.
len             The length of the file.
handle          An open file handle.
Description:
This message returns the length of the file specified by 'handle'. If the file cannot be
considered to have a length (eg. a terminal or pipe) then the returned value is -1
(0xFFFFFFFF). If a file larger than 2^31 is examined, the length value is undefined.
3.2.19 CL_IsTTY

Channel: CI_CLIB
Reason Code: 111
Signature:

\[(UINT32 \text{ status}) = \text{CL_IsTTY}(UINT32 \text{ handle})\]

Parameters:
status \hspace{1em} \text{Adp\_ok if close successful.}
handle \hspace{1em} \text{An open file handle.}

Description:
This message returns \text{adp\_ok} if the file handle given represents an interactive device, such as a console or terminal, and non-zero otherwise.

3.2.20 CL_TmpNam

Channel: CI_CLIB
Reason Code: 112
Signature:

\[(UINT32 \text{ status}, UINT32 \text{ nbytes}, [nbytes+1]\text{BYTE filename}) = \text{CL_TmpNam}(UINT32 \text{ maxlen}, UINT32 \text{ targetid})\]

Parameters:
status \hspace{1em} \text{Adp\_ok if close successful.}

Description:
This message returns a temporary host file name. The maximum length of a filename is passed to the host. The TargetID is some identifier from the target for this particular temporary filename. This value could be used directly in the generation of the filename.

If the host cannot create a suitable name, or the generated name is too long, then an error code is returned in status.
4. Error Codes
4.1 ADP Error Codes

ADP Error codes are used to describe errors which relate to the communication channel between target and host. Errors relating to the debug agent, or the operation of the application program (for example, reporting an exception) are reported using the ARM RDI error codes, which are reproduced in the next section.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adp_ok</td>
<td>0</td>
<td>Success; no error</td>
</tr>
<tr>
<td>adp_failed</td>
<td>256</td>
<td>Generic error</td>
</tr>
<tr>
<td>adp_malloc_failure</td>
<td>257</td>
<td>Failed to allocate dynamic memory (host side)</td>
</tr>
<tr>
<td>adp_illegal_args</td>
<td>258</td>
<td>Invalid parameter value or combination</td>
</tr>
<tr>
<td>adp_device_not_found</td>
<td>259</td>
<td>The device name specified was not found in the list of known devices.</td>
</tr>
<tr>
<td>adp_device_open_failed</td>
<td>260</td>
<td>Could not open comms device required</td>
</tr>
<tr>
<td>adp_device_already_open</td>
<td>261</td>
<td>Comms device already open, cannot share</td>
</tr>
<tr>
<td>adp_device_not_open</td>
<td>262</td>
<td>Operation requested which requires comms with target, but device not opened</td>
</tr>
<tr>
<td>adp_bad_channel_id</td>
<td>263</td>
<td>Channel ID provided outside the valid range</td>
</tr>
<tr>
<td>adp_callback_already_registered</td>
<td>264</td>
<td>Channel already has a registered callback; it must be removed</td>
</tr>
<tr>
<td>adp_write_busy</td>
<td>265</td>
<td>(another) write in progress; cannot write data at this time</td>
</tr>
<tr>
<td>adp_bad_packet</td>
<td>266</td>
<td>Bad (e.g. CRC error) packet received</td>
</tr>
<tr>
<td>adp_seq_high</td>
<td>267</td>
<td>Packet sequence number invalid – too high</td>
</tr>
<tr>
<td>adp_seq_low</td>
<td>268</td>
<td>Packet sequence number invalid – too low</td>
</tr>
<tr>
<td>adp_timeout_on_open</td>
<td>269</td>
<td>Couldn’t initiate target comms; target failed to respond</td>
</tr>
<tr>
<td>adp_abandon_boot_wait</td>
<td>270</td>
<td>On initialisation, the target did not respond to the host. This is an internal code which is translated to</td>
</tr>
<tr>
<td>adp_late_startup</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>adp_new_agent_starting</td>
<td>272</td>
<td></td>
</tr>
</tbody>
</table>
## Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adp_resend_failed</td>
<td>273</td>
</tr>
<tr>
<td>adp_timeout</td>
<td>274 general timeout</td>
</tr>
<tr>
<td>adp_bootmessage</td>
<td>275 Boot message at inappropriate time</td>
</tr>
</tbody>
</table>
4.2 RDI Error Codes

The RDI Error codes defined below are used by the RDI interface on the host, and are described in more detail in the RDI Protocol Specifications. They are also used by the target debug agent to report errors over ADP, saving the host debug agent from having to perform a translation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDIError_NoError</td>
<td>0</td>
<td>Success; no error. Alternatively, the ‘no’ answer of a Yes/No RDI Info question.</td>
</tr>
<tr>
<td>RDIError_Reset</td>
<td>1</td>
<td>Processor has hit the Reset vector</td>
</tr>
<tr>
<td>RDIError_UndefinedInstruction</td>
<td>2</td>
<td>Processor has hit the Undefined instruction vector</td>
</tr>
<tr>
<td>RDIError_SoftwareInterrupt</td>
<td>3</td>
<td>Processor has hit the Software Interrupt vector</td>
</tr>
<tr>
<td>RDIError_PrefetchAbort</td>
<td>4</td>
<td>Processor has hit the Prefetch Abort vector</td>
</tr>
<tr>
<td>RDIError_DataAbort</td>
<td>5</td>
<td>Processor has hit the Data Abort vector</td>
</tr>
<tr>
<td>RDIError_AddressException</td>
<td>6</td>
<td>Processor has hit the Address Exception vector (should only happen on 26 bit systems)</td>
</tr>
<tr>
<td>RDIError_IRQ</td>
<td>7</td>
<td>Processor has hit the IRQ vector (may be masked by Angel use of this vector)</td>
</tr>
<tr>
<td>RDIError_FIQ</td>
<td>8</td>
<td>Processor has hit the Fast Interrupt vector (may be masked by Angel use of this vector)</td>
</tr>
<tr>
<td>RDIError_Error</td>
<td>9</td>
<td>An unclassified error has occurred. Alternatively, the ‘no’ answer of a Yes/No RDI Info question.</td>
</tr>
<tr>
<td>RDIError_BranchThrough0</td>
<td>10</td>
<td>[For Angel]: A prefetch abort occurred.</td>
</tr>
<tr>
<td>RDIError_NotInitialised</td>
<td>128</td>
<td>Debug session initiated before Agent initialisation complete.</td>
</tr>
<tr>
<td>RDIError_UnableToInitialise</td>
<td>129</td>
<td>[For Angel]: Not used. (e.g. Armulator: no processor specification available)</td>
</tr>
<tr>
<td>RDIError_WrongByteSex</td>
<td>130</td>
<td>Debugger asked for initialisation with one byte sex, but the target was another.</td>
</tr>
<tr>
<td>RDIError_UnableToTerminate</td>
<td>131</td>
<td>Not currently used.</td>
</tr>
<tr>
<td>RDIError_BadInstruction</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>RDIError_IllegalInstruction</td>
<td>133</td>
<td></td>
</tr>
</tbody>
</table>
## Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDIError_BadCPUStateSetting 134</td>
<td>Attempt to use a coprocessor which has not been defined, or which is not present in the hardware.</td>
</tr>
<tr>
<td>RDIError_UnknownCoPro 135</td>
<td>Attempt to use an undefined coprocessor register.</td>
</tr>
<tr>
<td>RDIError_UnknownCoProState 136</td>
<td>Attempt to use an undefined coprocessor register.</td>
</tr>
<tr>
<td>RDIError_BadCoProState 137</td>
<td>Coprocessor register not writable for read op, or readable for write op.</td>
</tr>
<tr>
<td>RDIError_BadPointType 138</td>
<td>Not used in Angel.</td>
</tr>
<tr>
<td>RDIError_UnimplementedType 139</td>
<td>Break or Watch-point type (condition or address codes) not implemented.</td>
</tr>
<tr>
<td>RDIError_BadPointSize 140</td>
<td>Size of break/watch point not implemented.</td>
</tr>
<tr>
<td>RDIError_UnimplementedSize 141</td>
<td></td>
</tr>
<tr>
<td>RDIError_NoMorePoints 142</td>
<td>Limit on the number of watch/break points has been reached.</td>
</tr>
<tr>
<td>RDIError_BreakpointReached 143</td>
<td>Program stopped as a result of reaching a breakpoint.</td>
</tr>
<tr>
<td>RDIError_WatchpointAccessed 144</td>
<td>Program stopped as a result of reaching a watchpoint.</td>
</tr>
<tr>
<td>RDIError_NoSuchPoint 145</td>
<td>Break or Watch-point reference, e.g. in ClearBreak, does not refer to a current watch or breakpoint.</td>
</tr>
<tr>
<td>RDIError_ProgramFinishedInStep 146</td>
<td>Step over the exit SWI (ADP_Stopped_ApplicationExit) attempted.</td>
</tr>
<tr>
<td>RDIError_UserInterrupt 147</td>
<td>User requested an application program interruption, which has now occurred.</td>
</tr>
<tr>
<td>RDIError_CantSetPoint 148</td>
<td>The debug agent was unable to set a break or watchpoint.</td>
</tr>
<tr>
<td>RDIError_IncompatibleRDILevels 149</td>
<td>A specific RDI definition was requested which the target does not support.</td>
</tr>
<tr>
<td>RDIError_CantLoadConfig 150</td>
<td>Response to ADP_Ctrl_DownloadSupported: debug agent cannot load new processor configurations.</td>
</tr>
<tr>
<td>RDIError_BadConfigData 151</td>
<td>The processor config data selected by ADP_ICEM_SelectConfig is in error.</td>
</tr>
<tr>
<td>RDIError_NoSuchConfig 152</td>
<td>ADP_ICEM_SelectConfig called with a pattern which matches no currently loaded configuration.</td>
</tr>
</tbody>
</table>
## Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDIError_BufferFull 153</td>
<td>No more memory available for requested operation. This may indicate a system shortage, but many items are allocated from individual memory pools.</td>
</tr>
<tr>
<td>RDIError_OutOfStore 154</td>
<td>ADP_Ctrl_DownloadData called without a prior ADP_Ctrl_LoadAgent or ADP_ICEM_AddConfig.</td>
</tr>
<tr>
<td>RDIError_NotInDownload 155</td>
<td>Load file not in correct executable image format.</td>
</tr>
<tr>
<td>RDIError_PointInUse 156</td>
<td>ADP_Ctrl_LoadAgent or ADP_ICEM_AddConfig.</td>
</tr>
<tr>
<td>RDIError_BadImageFormat 157</td>
<td>Load file not in correct executable image format.</td>
</tr>
<tr>
<td>RDIError_TargetRunning 158</td>
<td>Could not stop target to perform requested operation.</td>
</tr>
<tr>
<td>RDIError_DeviceWouldNotOpen 159</td>
<td>Requested target communications device found, but failed to open.</td>
</tr>
<tr>
<td>RDIError_NoSuchHandle 160</td>
<td></td>
</tr>
<tr>
<td>RDIError_ConflictingPoint 161</td>
<td></td>
</tr>
<tr>
<td>RDIError_LinkTimeout 200</td>
<td>A timeout error occurred on the communications link to the target; the target is not responding.</td>
</tr>
<tr>
<td>RDIError_OpenTimeout 201</td>
<td>A timeout error occurred when attempting to contact the target for the first time; is there a target there?</td>
</tr>
<tr>
<td>RDIError_LinkDataError 202</td>
<td>A data error (CRC, format etc) occurred for a data packet on the link, and automatic recovery was disabled or not implemented.</td>
</tr>
<tr>
<td>RDIError_Interrupted 203</td>
<td></td>
</tr>
<tr>
<td>RDIError_LittleEndian 240</td>
<td>[Not an error]: Initialisation of target succeeded, and the target is now operating little-endian.</td>
</tr>
<tr>
<td>RDIError_BigEndian 241</td>
<td>[Not an error]: Initialisation of target succeeded, and the target is now operating big-endian.</td>
</tr>
<tr>
<td>RDIError_SoftInitialiseError 242</td>
<td>Recoverable error during initialisation of a (host) debug agent.</td>
</tr>
<tr>
<td>RDIError_InsufficientPrivilege 253</td>
<td>Operation not permitted: In Angel, memory read or write to Angel code or data space.</td>
</tr>
<tr>
<td>RDIError_UnimplementedMessage 254</td>
<td>Request not implemented by target.</td>
</tr>
<tr>
<td>RDIError_UndefindedMessage 255</td>
<td>Request not understood by target.</td>
</tr>
</tbody>
</table>