ICE Debug Board FPGA Interface protocol

Ben Kempke, Pat Pannuto {bpkempke,ppannuto}@umich.edu

Revision 0.2.5 — January 14, 2013

Introduction

The ICE board is a dual-purpose development board, enabling both high-speed programming (via DMA) of M3 chips and interfacing with live M3 sessions and external peripherals.

This means that libraries must be able to gracefully handle asychronous, unexpected events. In particular, some semantics resemebling transactions would be desirable, as well as an effort to maintain event ordering over the serial communications link. The consequence is that this document should be viewed less as a protocol specification and more of a living document as we explore the best semantics for this domain.

At a high-level, every message is a composition of an *event id*, a *message type* and an optional *message*. Every message sent *to* the ICE board **MUST** be replied to by either an ACK or NAK response. Every message sent *from* the ICE board **MUST NOT** be acknowledged. The rationale for this asymmetry is to simplify the ICE hardware design. With this mechanism, the hardware is not obligated to preserve any communication state beyond the immediate running context. The event ids provide a total ordering of the event history, insofar as is possible.

Contents

1 Event Ids

1.1 Motivation

Some decisions in microcontrollers necessarily race. As contrived example, if two GPIO pins are defined as interrupts but are electrically connected in the external circuit, and then line is pulled high, which interrupt fires first? While the decision is arbitrary, there is motivation to define an ordering of events in the system. The events observed by ICE can be replayed in the M-ultaor for debugging, but the is only possible if they can be accurately re-created.

In practice, this diverges to two ideas:

1.1.1 Concurrent Events

Two events could be labeled with the same event id, indicating that they occurred too close together in time for the ICE to distinguish them. This would require divergence in the simulator for debugging runs; do-able, but not trivial.

1.1.2 I/O Pass-Through

Currently, the FPGA and the M3 share GPIO pins (Nx2 header):

```
          /-----\
FPGA ---- | o o | ----- M3
          \-----/
          GPIO N
```

Instead perhaps we can explicitly pass all I/O through the FPGA:

```
/---\
| o | ---- FPGA -- {TP} -- M3
\---/
GPIO N
```

This has the disadvantage of doubling the number of FPGA I/O pins consumed for each of the M3 I/O pins. It does enable the FPGA to strictly define an order of events. We can possibly explore this with the current board by mapping GPIOs 16-23 as FPGA-input only, jumpering over GPIOs 8-15 and having the FPGA drive those, leaving GPIOs 0-7 as the first one.

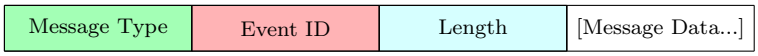
1.2 Implementation

Event IDs are an unsigned single byte number. They define a total ordering the events actually occurred in the system. In particular, if a command message is being sent to set GPIO 0 (an output) high at the same time that GPIO 1 (an input) goes high, the ordering will be `Event N: GPIO 1 --> High` then `Event N+1: GPIO 0 --> High`. With 1.1.2's pass through, perhaps the GPIO 1 transition could be delayed during command message reception, but this is for further exploration.

For message format consistency, event ids are included in both directions of communication, however the field may be safely ignored by the FPGA. The FPGA itself must by definition have some order of I/O events it processed, which are encapsulated by these event ids. The event id of a control message is assigned by the FPGA whenever it actually processes the event and is indicated to the controller via the ACK message.

2 Messages / Base Protocol

Messages are composed of a one-byte message type identifier followed by a one-byte event identifier followed by a one-byte unsigned length indicator followed by an optional message component. Visually:



Effort should be made to keep type identifiers within the ASCII range where reasonable, mapped to appropriate letters.

There are two types of transactions defined: *synchronous* and *asynchronous*. A synchronous message is one initiated by the controlling PC and must be responded to by a {N}ACK from the ICE board. An asychronous message is generated by the ICE board in response to a hardware event. Only one synchronous message is permitted to be live at any given time. The ICE board may send any number of asychronous messages before responding to the synchronous message.

The protocol in use is undefined until a version request is ACKed. The reception of a *NAK with reason* with a preferred version is **NOT** sufficient to establish a version, the controller **MUST** explicilty send another **v**ersion request and receive an ACK to establish the protocol version in use.

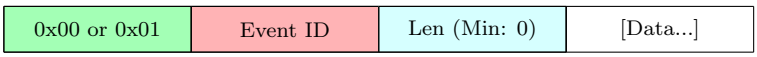
Only 'V' and 'v' messages are valid until a version has been negotiated. Any asynchronous hardware events may either be queued or discarded.

The base protocol defines the following immutable message types and their properties:

`0x00` ACK

`0x01` NAK

Synchronous Response



- **0:** ACK. Indicates success.
- **1:** NAK. Generic error code indicating failure.

– Unless otherwise specified...

- The remaining bytes shall be an ASCII-encoded string composed only of standard printable characters. The string shall not be NUL-terminated. The contents of this string is not well-defined and is expected to be something human-readable and useful.

– If specified...

- The response is permitted to be implementation defined.

`0x56 ('V')` – **Query Versions**

Synchronous Request

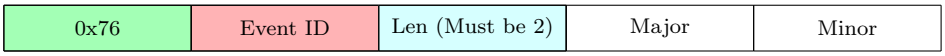
– This message queries the protocol version(s) this ICE implementation understands.

– If multiple versions are supported, they should be listed in order of version preference.

– Response:

`0x76 ('v')` – **Request to use version**

Synchronous Request



- The message shall be composed of exactly two bytes.
- Each byte shall be an unsigned number.
- The first byte shall be considered a major version number.
- The second byte shall be considered a minor version number.
- Response:

  - If the selected version is acceptable, an ACK shall be generated.

  - Otherwise a NAK shall be generated.
  - The NAK may optionally include a preferred version or list of versions in the same format as the 'V' response.
- This message is not queriable.

`0x58 ('X')` – **eXtension**

`0x78 ('x')` – **extension**

- These characters are reserved for future e**X**tentions.
- No further specification is defined here.

3 Protocols

The following protocols are currently well-defined:

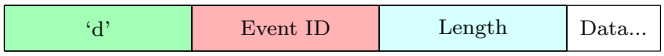
Contents

3.1 Version 0.1

Contents

3.1.1 0x64 'd' – Discrete interface I2C message

Synchronous Request, Asynchronous Message



The bytes in this message compose an I2C transaction.

The length field of this message necessarily limits the maximum message size to 255 bytes (addr + 254 data). Longer messages should be fragmented.

The sentinel value 255 for length indicates a *fragmented* message.

- A message fragment **MUST** be exactly 255 bytes long.
- A fragment message **MUST** be followed by another fragment, or terminated by a regular **d**iscrete message.
  - The ICE board is permitted to interleave other, non-I2C related messages (e.g. GPIO events).
- A series of message fragments **MUST** always be terminated by a discrete with an explicit length. An I2C transaction of length 0 is permitted, e.g.:
  - An I2C transaction of length 510 (1 byte addr + 509 bytes of data) would be **three** messages. The first of length 255 (addr + bytes 0-253), the second of length 255 (bytes 254-508), and the third of length 0 (there is no more data, but the fragment series must be terminated).
- Fragments are treated as one logical message, but individual I2C bus transactions, by an ICE board. In practice this means:
  - Each fragment message must be individually ACK'd by ICE.
    → A NAK'd fragment message ends an I2C message.
    → The NAK offset is relative to the current fragment, not the whole I2C transaction.
  - Only the first fragment includes the I2C address.

- A stop bit should **NOT** be generated after a fragment, instead the I2C clock should be stretched until the next fragment has arrived.

ICE will response with an ACK once every byte from an individual 'd' message has been ACK'd on the I2C bus.

If a byte is NAK'd on the I2C bus, ICE will respond with a NAK message of length 1 indicating the index of the first NAK'd byte (e.g. if the address is NAK'd, it will return 0).

NAK (0x01)	Event ID	Len (Must be 1)	Index of Byte NAK'd

3.1.2 `0x49 'I'` – Query I2C Configuration

Synchronous Request

0x49	Event ID	Len (Must be 1)	Parameter

These messages complement the set I2C messages.

The 'Parameter' field is the parameter specifier to query.

An ACK response should mimic the corresponding set message.

The following would query/response the address mask:

0x49	Event ID	0x01	0x61	
ACK (0x00)	Event ID	0x02	Ones Mask	Zeros Mask

3.1.3 `0x69 'i'` – Set I2C Configuration

Synchronous Request

The first byte of the message shall define which parameter is to be configured.

`0x69 0x63 'ic'` – Set I2C Clock Speed

'i'	Event ID	Len (Must be 2)	'c'	Clock Speed

- **Default:** `0x32` (50, 100 kHz)
- This shall be followed by a single byte valued N, where N * 2 kHz yeilds the desired clock speed. Values of N greater than 200 (400 kHz) exceed the I2C spec and may be rejected.

`0x69 0x61 'ia'` – Set ICE I2C Address

'i'	Event ID	Len (Must be 3)	'a'	Ones Mask	Zeros Mask

- **Default:** `0xff 0xff` (disabled)
- This shall be followed by two bytes, first the *ones mask* and then the *zeroes mask* as outlined below. The command sets the address mask that ICE board should pretend to be a device for. Coneceptually the mask is of the form `10xx010x`, where x's signify don't care. This is conveyed as a *ones mask* and a *zeroes mask*, where each mask defines the bits that must be a one or zero respectively. For the given example, the ones mask would be `10000100` and the zeroes mask `01001010`, generating a transaction of `0x61 0x84 0x4a`.
- To disable address-faking, set any bit as both required-one and required-zero. This impossible situation is a legal setting that will never match.
  - *Note:* While it is permissable to set the last bit must-be-zero (writeable-only) or must-be-one (readable-only), it is almost certainly an error to do so.

`0x69` Responses

- NAKs for this message shall be composed of an error code, followed by an optional explanitory string.

NAK (0x01)	Event ID	Len (Min: 1)	EINVAL (0x16)	["Out of Range"]

  - `EINVAL (22,0x16):` Invalid argument.

NAK (0x01)	Event ID	Len (Min: 1)	ENODEV (0x13)

  - `ENODEV (19,0x13):` The implementation does not support changing or querying this parameter. Unless otherwise specified, it **MUST** be hardcoded to the default.

3.1.4 `0x66 'f'` – FLOW (GOC) interface message

Synchronous Request

FLOW messages are formatted the exact same as 'd'iscrete messages.

3.1.5 `0x4f 'O'` – Query optical (FLOW (GOC)) Configuration

Synchronous Request

'O' (0x4f)	Event ID	Len (Must be 1)	Parameter

These messages complement the set I2C messages.

The 'Parameter' field is the parameter specifier to query.

An ACK response should mimic the corresponding set message.

3.1.6 `0x6f 'o'` – Set optical (FLOW (GOC)) Configuration

Synchronous Request

The first byte of the message shall define which parameter is to be configured.

`0x6f 0x63 'oc'`: Clock Speed (Divider)

'o'	Event ID	Len (Must be 4)	'c'	Clock Divider

- **Default:** `0x30D400` (2 MHz / 0x30D400 = .625 Hz)
- This shall be followed by a three byte value N (MSB-first), where 2 MHz / N yields the desired clock speed.

`0x6f` Responses

- NAKs for this message shall be composed of an error code, followed by an optional explanitory string.

NAK (0x01)	Event ID	Len (Min: 1)	EINVAL (0x16)	["Out of Range"]

  - `EINVAL (22,0x16):` Invalid argument.

3.1.7 0x47 'G' – Query GPIO State / Configuration

Synchronous Request

'G' (0x47)	Event ID	Len (Must be 2)	Parameter	GPIO IDX

These messages complement the set GPIO ('g') messages.

The 'Parameter' field is the parameter specifier to query.

An ACK response should mimic the corresponding set message.

3.1.8 0x67 'g' – Set / Configure GPIO

Synchronous Request, Asynchronous Message

The first byte of this message shall be a specificer, indicating what type of GPIO action is requested.

0x67 0x6c 'gl' – GPIO Level

- The first byte of the message shall be an integer indicating the GPIO index to set. The second byte shall be valued 0 or 1, depending on the desired GPIO state.

'o' (0x67)	Event ID	Len (Must be 3)	'l' (0x6c)	GPIO IDX	GPIO Val

0x67 0x64 'gd' – GPIO direction

- The first byte of the message shall be an integer indicating the GPIO index to set the direction of. The second byte shall be valued:
  - 0: Input
  - 1: Output
  - 2: TriState (DEFAULT)

'o' (0x67)	Event ID	Len (Must be 3))	'd' (0x64)	GPIO IDX	GPIO Direction

0x67 Responses

- ENODEV (19,0x13): The requested GPIO does not exist.

NAK (0x01)	Event ID	Len (Min: 1)	ENODEV (0x13)	["No such GPIO"]

- EINVAL (22,0x16): The requested GPIO exists, but cannot be configured this way at this time.

NAK (0x01)	Event ID	Len (Min: 1)	EINVAL (0x16)	["GPIO is input"]

3.1.9 0x50 'P' – Query Power State

Synchronous Request

'P' (0x50)	Event ID	Len (Must be 1)	Parameter	PWR IDX

These messages complement the Set Power ('p') messages.

The 'Parameter' field is the parameter specifier to query.

An ACK response should mimic the corresponding set message.

3.1.10 `0x70 'p'` – Set Power State

Synchronous Request, Asynchronous Message

Set Power State messages allow direct control of set-point voltage and on/off states for various power domains on the ICE board. The first byte of this message shall be a specificer, indicating which parameter is requested. The second byte of the message shall be the power domain identifier. Currently implemented power domain identifiers are:

- 0: M3 0.6V (0.675V Default)
- 1: M3 1.2V (1.2V Default)
- 2: M3 VBatt (3.8V Default)

`0x70 0x76 'pv'` – Voltage State

- The first byte of the message shall be a single byte indicating the power domain identifier to set. The second byte (*v_set*) shall specify the voltage according to the equation:

$$V_{out} = (0.537 + 0.0185 * v\_set) * V_{default}$$

Valid values for *v_set* range from `0` to `31`

'p' (0x70)	Event ID	Len (Must be 3)	'v' (0x76)	PWR IDX	*v_set*

`0x70 0x6f 'po'` – On/Off State

- The first byte of the message shall be a single byte indicating the power domain identifier to set. The second byte shall be valued `0` or `1`, depending on the desired On/Off state.

'p' (0x70)	Event ID	Len (Must be 3)	'o' (0x6f)	PWR IDX	On/Off

4 Document Revision History

Revision 0.2.5 – Jan 14, 2013

Add ‘p’ and ‘P’ messages

Revision 0.2.4 – Dec 20, 2012

Add ‘g’ and ‘G’ messages

Revision 0.2.3 – Dec 18, 2012

Add ‘f’, ‘o’, and ‘O’ messages

Revision 0.2.2 – Dec 15, 2012

‘d’ fragments are 255 bytes long so that a final fragment of maximum length can be distinguished

Make explicit that each ‘d’ fragment message is ACK/NAK’d

When a ‘d’ fragment is NAK’d, that ends the transaction

Revision 0.2.1 – Nov 29, 2012

0.1’s ‘d’ → ‘e’

Add fragmented ‘d’ messages

Revision 0.2 – Nov 27, 2012

Largely redefine the protocol (ish)

Revision 0.1.2 – Nov 26, 2012

Restrict Version to ‘V’ only

Restrict eXtension to ‘X’ only

Revision 0.1.1 – Nov 16, 2012

Add visualized packets

Revision 0.1 – Nov 14, 2012

Initial revision