Arduino Communication protocol, Ensured (ACE) Specification

Jakob Odersky

February 26, 2013

1 Introduction & Purpose

The purpose of the Arduino Communication protocol, Ensured (ACE) is to provide reliable and reactive communication between a micro controller and a computer (or two micro controllers). The main features of the protocol are reliability, in a sense that the communicating parties know if a sent message was received or not, and reactivity, i.e. the main program loop is not blocked whilst waiting for a message response.

Compared to the OSI model, ACE is a data-link layer protocol.

2 Functional Overview

The operation of ACE may be divided into two parts: data framing and automatic repeat request. Data framing is the action of "packaging" a sequence of data bytes so that it may be sent as a whole over a physical channel. A checksum sent in the frame ensures that if an error occurred during transmission it is very likely to be detected and the whole frame rejected.

However, if such an error occurs, the sender has no way of knowing it and the sent data will be lost. To remedy this kind of situation, ACE uses a form of automatic repeat request mechanism.

See the following sections on details of these two parts.

3 Framing

As to ensure reliability, data is always sent in frames containing a checksum. A frame is composed of: a header consisting of one start byte, followed by an arbitrary amount of data bytes[1], followed by a trailer consisting of one checksum byte and one stop byte. The checksum is calculated by taking the exclusive or of all data bytes. Furthermore, data and checksum bytes are escaped by a preceding escape byte. An overview of a data frame and the definition of special byte values are given in table 1.

An invalid frame should be ignored by the receiver and no action taken.

Structure	Header	Data	Trailer	
Detailed structure	start	data	checksum	stop
Length (bytes)	1	any (within limits of implementation)	1	1
Hexadecimal values	0x02		XOR of all data bytes	0x10

Escape byte value: 0x03.

Table 1: A data frame in ACE.

[1] limits are implementation specific

3.1 Example 1

As a first example, consider the message "hello" encoded in ASCII. Equivalently, this message may be represented as a sequence of bytes (in decimal representation):

```
104 101 108 108 111
```

The checksum of this message is 98, therefore the corresponding data frame is:

```
002 104 101 108 108 111 098 016
```

3.2 Example 2

As a second example, consider the byte sequence:

```
001 108 002 111 003 102
```

The values 002 and 003 are special bytes and therefore have to be escaped. Considering that the checksum is 101, the resulting frame is given by:

```
002 001 108 003 002 111 003 003 102 101 016
```

4 Automatic Repeat Request (ARQ)

To remedy the loss of invalid frames, ACE uses a kind of stop-and-wait ARQ. After sending a frame, the sender waits for an acknowledgement of the receiver before transmitting a next frame. If no acknowledgement is received in a timeout delay, the message is retransmitted. If after retransmitting the message several times no acknowledgement has been received, the message is considered to have been lost and an error is generated at the sender side. Furthermore, if the sender whilst waiting for an acknowledgement, receives an acknowledgement for a different frame other than he had sent or receives a new data frame, the sent frame is considered to be lost and an error is generated on the sender side. Only if the correct acknowledgement is received the message may be considered successfully sent and an action may be taken.

On the receiver side, if a frame is received, an acknowledgement to that frame is sent back and application specific action (to the message) is taken. If the same frame is received following the acknowledgement, it is considered that the sender did not receive the acknowledgement and the acknowledgement is retransmitted, this time without taking application specific action.

To differentiate frames and to enable the distinction between acknowledgements and data frames, each message is preceded with a sequence byte and a command byte (in that order) before being sent as a frame.

The sequence byte is used an identification number and is incremented for every new message (a new message is a message that has not been retransmitted). In case the message is an acknowledgement, the sequence number determines to what message the acknowledgement responds. In case of an overflow, the sequence number restarts at zero.

The command byte determines if the message is data or an acknowledgement. Its value is 0x05 in case of data and 0x06 in case of an acknowledgement.

For an example, see the C pseudo-code in appendix A.

5 Reactivity

The previous sections specified the "reliability" part of ACE. The second important part of the protocol is reactivity. Since the concept of reactivity is very broad and possible implementations vary greatly, the ACE specification does not give any concrete requirements. Implementations must however provide a way to notify applications that a message was successfully transferred or that an error occurred.

6 Conclusion

This document specifies the ACE protocol. The reference implementations in C and Scala are authoritative in case of any uncertainties or inconsistencies with this document.

A Stop-and-wait ARQ, example implementation

```
#define DATA 0x05
#define ACK 0x06

void send(uint size, uint8_t* message) {
  increment_seq_counter();
  send_frame({seq, DATA, message});
  awaiting_ack = true;
  start_timer();
}

void receive(uint size, uint8_t* data) {
  uint8_t seq = data[0];
  uint8_t cmd = data[1];
  uint8_t* message = &(data[2]);
  int16_t message_size = size - 2;

  if (!awaiting_ack) { //ready to receive
    if (cmd == DATA) { //the message is data
      if (last_received_seq != seq) { //the message was not already processed
        last_received_seq = seq;
        application_specific_action(); //process message
      }
      send_ack(seq); //send acknowledgement to the received frame
    } else {
      //ignore case in which an ack is received even though none is awaited
    }

  } else { //awaiting ack
    awaiting_ack = false; //got something so stop waiting for ack
    stop_timer(); //stop timeout

    if (cmd == ACK && seq == last_sent_seq) { // the correct ack was returned
      application_specific_action_send_success();
    } else { // wrong ack or data received
      error_bad_acknowledgement();
    }
  }
}

void timeout() {
  if (resends > MAX_RESENDS) {
    error_no_ackowledgement();
  } else {
    resends += 1;
    restart_timer();
    resend_message();
  }
}
```