Pecc communication

Tore Skjellnes
torsk@elkraft.ntnu.no

January 7, 2000

Abstract

This document describes communication with Peccros, a tiny operating system used by microcontroller applications for communication.

First, the general abstract layers are explained. Second, each layer is described separately. Finally, examples of communication and implementation details are given.

1 Introduction

The communication with Peccros can be divided into three abstraction layers. The first layer is the low-level communication layer. It describes how the data is transmitted through a transmission media. The second layer is the protocoll layer. It is responsible for encoding arbitrary packets of data into a form that can be transmitted safely and reliably by the low-level layer. The third layer is the command layer. It describes the shape of the actual command packets which the two communication systems exchange. The layer stucture is designed such that the individual layers can be replaced fairly independently.

2 The low-level layer

The low-level layer is responsible for the actual data transfer.

2.1 Standard RS232

This layer implements the asynchronus serial communcation standard RS232. The data is sent at 9600 baud using 8 data bits, 1 stop bit and no parity. No handshaking is used.

3 The protocoll layer

The protocoll layer is responsible for encoding an arbitrary packet into a form that can be transmitted by the low-level layer.

3.1 The Pecc protocoll, version 5.0

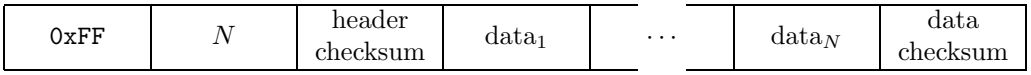
Some of the features of this protocoll includes:

- It is a byte oriented, self synchronizing protocoll.
- It has a theoretical maximum number of data bytes in a packet of 254 bytes.
- It has a maximum number of bytes in a encoded package of 512 bytes.
- It uses two simple additive checksums to ensure data integrity, one for the package header, and one for the data.

- It uses a lone byte value `0xFF` as a special value to synchronize packages. A byte value of `0xFF` used in any other context is encoded as two consecutive bytes with value `0xFF`.

3.1.1 The protocoll format

An encoded packet uses the following format:



3.1.2 Data byte encoding

If any byte value, except for the header byte of a packet, is `0xFF`, it is encoded as two consecutive bytes with value `0xFF`.

3.1.3 Checksumss

The checksums are simple additive checksums. If all the decoded bytes and the checksum are added, the lower 8 bits of the sum should be zero. Any other results indicates an error. To calculate a checksum, all the bytes can be added, then negated, and then anded with `0xFF`.

3.1.4 Header

The first two bytes, `0xFF` and N, always begins a new package. Here N is the number of *decoded* data bytes in the package in the range `0x01` to `0xFE`. This means that a package always start with a "lone" byte value of `0xFF`.

The header checksum is the checksum of these two bytes. Note that if N is 2, the header checksum is `0xFF`, and is encoded as two consecutive bytes of `0xFF`.

As N is constricted to the range `0x01` to `0xFE`, the byte sequence `0xFF 0x00` is not part of any valid package. Thus it can be used as a error indication (e.g. frame error) at the receiving end.

3.1.5 Data

The header is followed by the encoded packet data and the data checksum. The maximum number of bytes are $254 \cdot 2 + 1 = 509$. This worst case is 254 `0xFF`s, followed by the checksum = `0xFE`, or 253 `0xFF`s and a `0xFE`, followed by the checksum = `0xFF`. Thus the maximum package size is $509 + 3 = 512$.

However, to conserve buffer space on the microcontroller, the sender and receiver might agree upon a smaller maximum package size.

4 The command layer

The command layer defines the actual command packets which the two communication systems exchange.

4.1 Command packets

A standard command packet is made up by a byte identifying the command, possibly followed by a byte with a sub command, followed by the command data bytes. A summary of command bytes is shown in table 1. The receiver responds by sending an acknowledge packet. This acknowledge contains an error code, denominated in this document by *EC*. A summary of all error codes is shown in table 2.

New: The virtual address command packets have changed number and format.

Command	Description
`0x01`	Ping
`0x02`	Start
`0x03`	Start possible
`0x04`	Stop
`0x05`	Reset
`0x06`	Run function
`0x07`	Put variable
`0x08`	Get variable
`0x0C`	Initialize
`0x10`	Virtual address command packet:
`0x10 0x01`	Run virtual function
`0x10 0x02`	Put virtual variable
`0x10 0x03`	Get virtual varibale

Table 1: A summary of all commands

EC	Description
`0x00`	No error
`0xFF`	General error
`0xFE`	Timeout
`0xFD`	Break
`0xFC`	Not found
`0xFB`	Null pointer
`0xFA`	Wrong magic number
`0xF9`	Wrong data length
`0xF8`	Unknown command id
`0xF7`	Program is running
`0xF6`	Program is not running
`0xF5`	Unknown virtual address
`0xF4`	Permission denied

Table 2: A summary of all error codes

4.1.1 Ping

Command:	0x01	0x00
Answer:	0x01	EC

The ping packet is used to ask the receiver if it is "alive."

4.1.2 Start

Command:	0x02	0x00
Acknowledge:	0x02	EC

The start packet is used to start the user application.

4.1.3 Start possible

Command:	0x03	0x00
Acknowledge:	0x03	EC

The start possible packet is used to check if starting the user application is possible.

4.1.4 Stop

Command:	0x04	0x00
Acknowledge:	0x04	EC

The stop packet is used to stop the user application.

4.1.5 Reset

Command:	0x05	0x00
Acknowledge:	0x05	EC

The reset packet is used to reset the processor. The acknowledge packet is only sent in the event of an error, as the reset is immediate.

4.1.6 Run function

Command:	0x06	$addr_{0..7}$	$addr_{8..15}$	$addr_{16..23}$	$addr_{24..31}$
Acknowledge:	0x06	EC			

The run function packet is used to run an arbitrary function. *addr* gives the address of the function. The function is run immediately, and the acknowledge packet is sent after it has returned.

4.1.7 Put variable

Command:	0x07	id	N_d	$addr_{0..7}$	$addr_{8..15}$
	$addr_{16..23}$	$addr_{24..31}$	$data_1$	$\cdots$	$data_{N_d}$
Acknowledge:	0x07	id	EC		

The put variable function packet is used to put arbitrary data in any memory location. id is a id used to identify this package, and is returned in the acknowledge package. N_d is the number of data bytes. *addr* gives the address of the memory location.

4.1.8 Get variable

Command:

0x08	id	N_d	$addr_{0..7}$	$addr_{8..15}$
$addr_{16..23}$	$addr_{24..31}$			

Acknowledge:

0x08	id	$EC \neq 0$

Acknowledge:

0x08	id	0x00	N_d
$data_1$	$\cdots$	$data_{N_d}$	

The get variable function packet is used to get arbitrary data from any memory location. *id* is a id used to identify this package, and is returned in the acknowledge package. N_d is the number of data bytes. *addr* gives the address of the memory location.

4.1.9 Initialize

Command:

0x0C	0x00

Acknowledge:

0x0C	EC

The initialize packet is used to initialize the user application after downloading.

4.1.10 Run virtual function

Updated

Command:

0x10	0x01	$vaddr_{0..7}$	$vaddr_{8..15}$

Acknowledge:

0x10	EC	0x01

The run virtual function packet is used to run a function identified by *vaddr*. The function is run immediately, and the acknowledge packet is sent after it has returned.

4.1.11 Put virtual variable

Updated

Command:

0x10	0x02	id	N_d	$vaddr_{0..7}$
$vaddr_{8..15}$	$data_1$	$\cdots$	$data_{N_d}$	

Acknowledge:

0x10	EC	0x02	id

The put virtual variable function packet is used to put arbitrary data in a memory location identified by *vaddr*. *id* is a id used to identify this package, and is returned in the acknowledge package. N_d is the number of data bytes.

4.1.12 Get virtual variable

Updated

Command:

0x10	0x03	id	N_d	$vaddr_{0..7}$	$vaddr_{8..15}$

Acknowledge:

0x10	$EC \neq 0$	0x03	id

Acknowledge:

0x10	0x00	0x03	id
N_d	$data_1$	$\cdots$	$data_{N_d}$

The get virtual variable function packet is used to get arbitrary data in a memory location identified by *vaddr*. *id* is a id used to identify this package, and is returned in the acknowledge package. N_d is the number of data bytes.

5 Examples

5.1 Example packages

Some examples of encoded packages are shown below:

Ping:

0xFF	0x02	0xFF	0xFF
0x01	0x00	0xFF	0xFF

Ping ack.:

0xFF	0x02	0xFF	0xFF
0x01	0x00	0xFF	0xFF

Start:

0xFF	0x02	0xFF	0xFF
0x02	0x00	0xFE	

Start ack.:

0xFF	0x02	0xFF	0xFF
0x02	0x00	0xFE	

5.2 Example decoder

The following code is an example package decoder. The function decodes a single data byte and updates the decoder state saved in the `PECCP50` structure accordingly. If the input value `data` is outside the byte region of `0x00` to `0xFF`, it represents a faulty character (e.g. frame error). The decoder is reset to the synchronization state on errors. If the decoding is successful, the length of the packet is returned (≤ 254). The structure `PECCP50` must be initialized to state `PECCP50_SYNCH` before it is used for the first time. New

The function returns:

$N > 0$ if the decoding was successful. N is the length of the data packet.

$= 0$ if there was not enough data avaiable (yet).

$EC < 0$ on error, in particular

- -1 lost synchronization. The last character read was not part of a valid package. Expect a lot of these after other errors.
- -2 data error. A `FF 00` or *error* was received.
- -3 header checksum error.
- -4 body checksum error.

```
typedef struct PECCP50_ PECCP50;

struct PECCP50_
{
    enum {
        PECCP50_SYNCH,
        PECCP50_LEN,
        PECCP50_LSUM,
        PECCP50_DATA,
        PECCP50_ESUM,
        PECCP50_VALID
    } state;
    int pos;
    int len;
    Bool lastff;
    Byte8 *iter;
    Byte8 buf[254];
```

```
    Byte8 sum;
};

int
PECCP50Decode(PECCP50 *p, int data)
{
    if (data < 0 || data > 0xFF) {
        /* Resynch on <error> */
        p->state = PECCP50_SYNCH;
        return -2;
    }

    switch (p->state) {
    case PECCP50_VALID:
        p->state = PECCP50_SYNCH;
        /* fallthrough */
    case PECCP50_SYNCH:
        if (data == 0xFF)
            p->state = PECCP50_LEN;
        else
            return -1;
        break;

    case PECCP50_LEN:
        if (data > 0 && data < 0xFF) {
            /* Valid length */
            p->len = data;
            p->lastff = FALSE;
            p->state = PECCP50_LSUM;
        } else if (data == 0xFF) {
            /* FF FF - last FF might be a synch, continue in
               LEN state */
            return -1;
        } else {
            /* FF 00 - resynch */
            p->state = PECCP50_SYNCH;
            return -2;
        }
        break;

    case PECCP50_LSUM:
    case PECCP50_DATA:
    case PECCP50_ESUM:
        if (p->lastff) {
            if (data > 0 && data < 0xFF) {
                /* A synch (!) */
                p->len = data;
                p->state = PECCP50_LSUM;
                return -1;
            } else if (data == 0xFF) { /* FF FF -> FF */
                p->lastff = FALSE;
            } else {/* FF 00 - resynch */
                p->state = PECCP50_SYNCH;
                return -2;
            }
        } else if (data == 0xFF)
            p->lastff = TRUE;
```

```
        if (!p->lastff) {
            switch (p->state) {
            case PECCP50_LSUM:
                if (((0xFF + p->len + data) & 0xFF) != 0) {
                    /* Checksum error -> resynch */
                    p->state = PECCP50_SYNCH;
                    return -3;
                } else {
                    p->iter = &p->buf[0];
                    p->pos = 0;
                    p->sum = 0;
                    p->state = PECCP50_DATA;
                }
                break;

            case PECCP50_DATA:
                *p->iter++ = data;
                p->sum += data;

                if (++p->pos == p->len)
                    p->state = PECCP50_ESUM;
                break;

            case PECCP50_ESUM:
                if (((p->sum + data) & 0xFF) == 0) {
                    /* Packet done ! */
                    p->state = PECCP50_VALID;
                    return p->len; /* ...and there were much rejoicing */
                }

                /* Checksum error -> resynch */
                p->state = PECCP50_SYNCH;
                return -4;

            default:
                p->state = PECCP50_SYNCH;
            }
        }
        break;

    default:
        p->state = PECCP50_SYNCH;
    }

    return 0;
}
```

A Control of SMART-motor

The SMART-motor controller card uses the PECCROS kernel for communication. This section describes commands available for users of this card.

A.1 Ping

The ping packet can be used as a communcation check. It can also be used repeatedly to regain synchronization.

A.2 Initialization

As the program is resident on the card, the initialization of the program is done automatically at reset. Thus, no initialize packet is needed.

A.3 Start and stop

The start and stop packets are used for starting and stopping the controller. If two start packets are sent in a row, or stop is sent without the controller being started, an error will be returned.

A.4 Setting speed reference and reading actual speed

Two virtual variables are used for controlling the controller. The speed reference has virtual number 0. It can be set with the put virtual variable packet. This is an example that sets the speed reference to `0x0105`.

Error fixed. Reference is sent in little endian format.

Command (raw):

0xFF	0x08	0xF9	0x10	0x02
0x3F	0x02	0x00	0x00	0x05
0x01	0xA7			

Acknowledge (raw):

0xFF	0x04	0xFD	0x10	0x00
0x02	0x3F	0xAF		

This example uses the get virtual variable packet to read the speed reference (=`0x1005`):

Updated

Command (raw):

0xFF	0x06	0xFB	0x10	0x03
0x55	0x02	0x00	0x00	0x96

Acknowledge (raw):

0xFF	0x06	0xFB	0x10	0x00
0x03	0x55	0x02	0x05	0x01
0x90				

The actual speed measurement is a read only variable with virtual number 1. This example uses the get virtual variable packet to read the speed (=`0x00FF`):

Updated

Command (raw):

0xFF	0x06	0xFB	0x10	0x03
0x56	0x02	0x01	0x00	0x94

Acknowledge (raw):

0xFF	0x07	0xFA	0x10	0x00
0x03	0x56	0x02	0xFF	0xFF
0x00	0x96			