



SCANNDY

SComP communication

manufactured by:



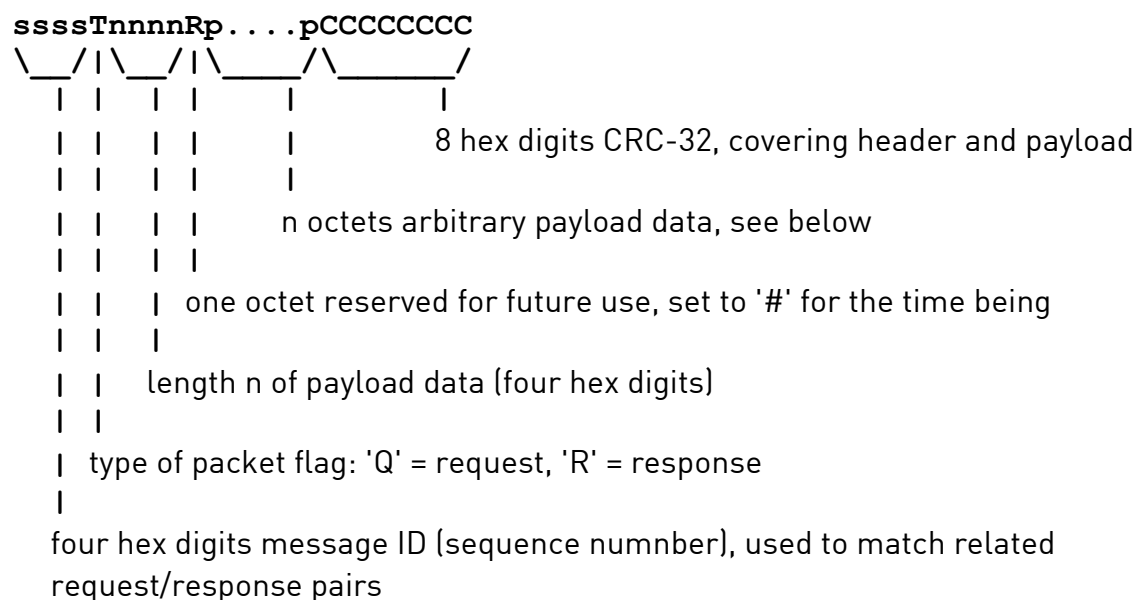
SComP – Scannyd Communications Protocol

SComP is intended to allow data transfer to and from, as well as remote control of, Panmobil devices by exchanging messages between two endpoints. SComP uses message ids to allow for matching request/response pairs and assures data integrity through use of CRC-32 checksums. Its main use is to provide reliable bidirectional communication over unreliable serial communication lines. The SComP metadata is human readable and is composed of only ASCII characters, to simplify debugging and transport over binary intransparent communication lines. In the design, simplicity was given higher precedence than efficiency.

SComP Packet Structure

Each SComP data packet is composed of three main parts:

- the fixed size (10 octets) packet header
- variable length (up to 65535 octets) payload data
- the fixed size (8 octets) packet trailer



Message IDs

The request generator should generate a new message ID for each request. The recommended implementation is a counter that is incremented for each generated request, but this approach is not mandatory. The receiver shall not make any assumptions about the algorithm generating the request message IDs. The sole purpose of the request message ID is to re-use it in the header of the corresponding response packet.

CRC-32

The CRC-32 implementation shall use an initial value of 0xFFFFFFFF and a polynomial of 0xEDB88320. The result shall be finalized by XORing it with 0xFFFFFFFF.

Example:

-> 0042 Q 000C # beep:200,200 742C823D

<- 0042 R 0007 # ok 538068BA

(The spaces were added to improve readability, they are not part of the protocol!)

CRC-32 Reference Implementation

```
#include "crc32.h"

static uint32_t crc_tab[256];
static int crc_tab_isinit = 0;
static uint32_t crc_poly = 0xEDB88320L;

static void crc32inittab( void )
{
    uint32_t crc;
    int i, j;

    for ( i = 0; i < 256; i++ )
    {
        crc = i;
        for ( j = 8; j > 0; j-- )
        {
            if ( crc & 1 )
                crc = ( crc >> 1 ) ^ crc_poly;
            else
                crc >>= 1;
        }
        crc_tab[i] = crc;
    }
    crc_tab_isinit = 1;
}

void crc32_init( crc32_t *pcrc )
{
    if ( !crc_tab_isinit )
        crc32inittab();
    *pcrc = 0xFFFFFFFFL;
}

void crc32_final( crc32_t *pcrc )
{
    *pcrc ^= 0xFFFFFFFFL;
}

void crc32_update( crc32_t *pcrc, uint8_t *data, size_t size )
{
    while ( size-- )
        *pcrc = ( *pcrc >> 8 ) ^ crc_tab[ ( *pcrc ^ *data++ ) & 0xFF ];
}

crc32_t crc32_calc( uint8_t *data, size_t size )
{
    crc32_t crc;
    crc32_init( &crc );
    crc32_update( &crc, data, size );
    crc32_final( &crc );
    return crc;
}

/* EOF */
```

SComP – Payload Structure

Though generally capable of carrying arbitrary binary payload data, SComP is mainly intended to transfer text messages consisting of specific commands and accompanying data to facilitate interaction with devices of the PANMOBIL SCANNDY family.

All commands are treated as case sensitive. Command and data shall be separated by a colon (':'). If no data is appended, the colon may be dropped. In case a command is defined to use more than one parameter, subsequent parameters

The generic response upon successful completion is simply 'ok', unless otherwise noted. Unless otherwise noted the generic response upon failure is simply 'error', usually followed by an error code and a plain text error message, each separated by a colon (':').

The following commands are currently defined (the angle brackets denote parameters and do not actually appear in the message):

getversion:

request Scanndy ScomP version, available since firmware version 3696

beep:<f>,<d>[,<a>]

request Scanndy to beep for a duration of <d> ms at <f> Hz, both in decimal notation. If third parameter <a> is set, beep works non-blocking

vibrate:<d>

request Scanndy to vibrate for a duration of <d> ms, in decimal notation

leds:<leds>,<ctrl>[,<delay_on>,<delay_off>]

request Scanndy to control the <leds> according to <ctrl>.

<leds> is a bitwise combination of 1(green), 2(yellow) and 4(blue), you can't control the red led.

<ctrl> could be one of:

on

off

timer -> <leds> blink with given <delay_on>, <delay_off>

fwcontroled -> give back control of <leds> to firmware

print:<text>

request Scanndy to print a text string to the screen NOTE: the escape sequences '\r' and '\n' are interpreted as carriage return and carriage return plus line feed, respectively

setcsr: <x>, <y>

set cursor position, column and row coordinates in decimal notation

clrscr: all clear whole screen

clrscr: client clear client area

clrscr: title clear title bar

keydata: <k>

unsolicited message, result of a proactive keypress on Scannidy; contains a single printable character corresponding to the button pressed

barscan: <t>

request Scannidy to initiate a barcode scan for t ms

bardata: <data>

result of a barcode scan, <data> is presented as text string (sent as response to a barscan request or as unsolicited request upon user action)

NOTE: a failure to read any barcode data simply results in the response "bardata", without any actual data attached. No specific error response is generated.

rfidscan: tid[, <addr>, <len>]

request to scan an RFID TID, optionally addressed

rfidscan: epc[, <addr>, <len>]

request to scan RFID EPC data (UHF Gen2 only), optionally addressed

rfidscan: usr, <addr>, <len>

request to scan RFID user data

rfiddata: <data>

result of an RFID scan, 'data' is presented in hexadecimal notation (message is sent as response to a barscan request or as unsolicited request upon user action)

NOTE: a failure to read any RFID tag data simply results in the response "rfiddata", without any actual data attached. No specific error response is generated.

rfidwrite: usr, <addr>, <data>[, <maxTries>, <maxTriesPerBlock>, <reader>]

request Scannidy to write RFID user data, <addr> in decimal notation, <data> in hexadecimal notation.

<maxTriesPerBlock> is the number of times, Scannidy tries to write a block (atm only valid for ISO 14434).

reader is only interesting for comboreader Scannidys (ISO 15693 and ISO 14443):

reader = 0, only try reading ISO 15693 tags
reader = 1, only try reading ISO 14443 tags
all other values leading to all tag reading

rfidwrite:epc[,<addr>],<data>

request Scannidy to write EPC data (UHFGen2 only); if <addr> is specified, the EPC length stored in the PC word will not be updated.

getcache:

request Scannidy to send cached data

property:get,<section:entry>

request Scannidy to return a property value from config.ini
e.g: property:get,general:serial

SComP – Reference Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/time.h>

#include "crc32.h"
#include "scomp.h"

#define HDR_SIZE          10
#define TRL_SIZE          8
#define SND_TIMEOUT       100

static int dummycb( void *data, int size, int to )
{
    return 0;
    (void)data;
    (void)size;
    (void)to;
}

static struct {
    int use_crc;
    ScompCb_t snd;
    ScompCb_t rcv;
} scompOpt = {
    1,
    dummycb,
    dummycb,
};

static unsigned int scompSeq = 0;

static inline int xd2dd( int c )
{
    char h[] = "0123456789ABCDEFabcdef";
    char *p = strchr( h, c );
    int d;
    if ( !p || !*p )
        return -1;
    d = p - h;
    if ( 15 < d )
        d -= 6;
    return d;
}

static unsigned long xtoul( const char *s, int *rl )
{
    int d;
    unsigned long res = 0;

    while ( *rl && 0 <= ( d = xd2dd( *s ) ) )
    {

```



```

        res = res * 16 + d;
        ++s;
        --*rl;
    }
    return res;
}

int ScompSetOption( enum ScompOpt_enum opt, ScompOpt_t val )
{
    int res = SCOMP_ERR_OK;
    switch ( opt )
    {
        case SCOMP_OPT_USE_CRC:
            scompOpt.use_crc = val.i;
            break;
        case SCOMP_OPT_SNDCB:
            scompOpt.snd = val.cb;
            break;
        case SCOMP_OPT_RCVCB:
            scompOpt.rcv = val.cb;
            break;
        default:
            res = SCOMP_ERR_BADOPT;
            break;
    }
    return res;
}

const char *ScompStrErr( int e )
{
    const char *msg[] = {
        "no error", // SCOMP_ERR_OK
        "timeout", // SCOMP_ERR_TIMEOUT
        "send", // SCOMP_ERR_SND
        "receive", // SCOMP_ERR_RCV
        "crc mismatch", // SCOMP_ERR_CRC
        "sequence number mismatch", // SCOMP_ERR_SEQ
        "message type mismatch", // SCOMP_ERR_TYPE
        "parameter out of range", // SCOMP_ERR_RANGE
        "number encoding", // SCOMP_ERR_BADNUM
        "bad option specifier", // SCOMP_ERR_BADOPT
        "buffer overflow", // SCOMP_ERR_OVER
    };
    if ( 0 > e || ( sizeof msg / sizeof *msg ) <= (unsigned)e )
        return "unknown";
    return msg[e];
}

static inline void incSeq( void )
{
    scompSeq = ( scompSeq + 1 ) % ( SCOMP_MAXSEQ + 1 );
}

int ScompSend( char *data, int size, int *seq, int type )
{
    int res = SCOMP_ERR_OK;

```

```

char hdr[HDR_SIZE + 1];
char trl[TRL_SIZE + 1] = "00000000";
crc32_t crc;

if ( SCOMP_MAXPAYLOAD < size )
    return SCOMP_ERR_RANGE;
if ( 0 > *seq )
    *seq = scompSeq;
sprintf( hdr, "%04X%c%04X#", *seq, type, size );
if ( scompOpt.use_crc )
{
    crc32_init( &crc );
    crc32_update( &crc, (uint8_t *)hdr, HDR_SIZE );
    crc32_update( &crc, (uint8_t *)data, size );
    crc32_final( &crc );
    sprintf( trl, "%08X", crc );
}
if ( HDR_SIZE != scompOpt.snd( hdr, HDR_SIZE, SND_TIMEOUT )
    || size != scompOpt.snd( data, size, SND_TIMEOUT )
    || TRL_SIZE != scompOpt.snd( trl, TRL_SIZE, SND_TIMEOUT ) )
{
    res = SCOMP_ERR_SND;
}
incSeq();
return res;
}

```

```

int ScompSendResponse( char *data, int size, int seq )
{
    int sseq = seq;
    return ScompSend( data, size, &sseq, SCOMP_RESPONSE );
}

```

```

int ScompRcv( char *data, int *size, int *seq, int *type, int to )
{
    int rl = 0;
    int len = 0;
    char hdr[HDR_SIZE + 1];
    char trl[TRL_SIZE + 1];
    crc32_t crc = 0x00000000;

    // get header
    rl = scompOpt.rcv( hdr, HDR_SIZE, to );
    if ( 0 == rl )
        return SCOMP_ERR_TIMEOUT;
    else if ( HDR_SIZE != rl )
        return SCOMP_ERR_RCV;
    hdr[rl] = '\0';
    // extract sequence number
    rl = 4;
    *seq = xtoul( hdr, &rl );
    if ( 0 != rl )
        return SCOMP_ERR_BADNUM;
    // extract type field (request or response)
    *type = hdr[4];
    // extract length field
    rl = 4;
    len = xtoul( hdr + 5, &rl );
}

```

```

    if ( 0 != rl )
        return SCOMP_ERR_BADNUM;
    if ( len > *size )
        return SCOMP_ERR_OVER;
    *size = len;

    // get payload data
    if ( 0 < len )
    {
        rl = scompOpt.rcv( data, len, to );
        if ( 0 == rl )
            return SCOMP_ERR_TIMEOUT;
        else if ( rl != len )
            return SCOMP_ERR_RCV;
    }

    // get trailer
    rl = scompOpt.rcv( trl, TRL_SIZE, to );
    if ( 0 == rl )
        return SCOMP_ERR_TIMEOUT;
    else if ( TRL_SIZE != rl )
        return SCOMP_ERR_RCV;
    trl[rl] = '\0';
    // check crc
    if ( scompOpt.use_crc )
    {
        crc32_init( &crc );
        crc32_update( &crc, (uint8_t *)hdr, HDR_SIZE );
        crc32_update( &crc, (uint8_t *)data, len );
        crc32_final( &crc );
        rl = 8;
        if ( xtoul( trl, &rl ) != crc || 0 != rl )
            return SCOMP_ERR_CRC;
    }
    return SCOMP_ERR_OK;
}

int ScompExch( char *query, int qsz, char *resp, int *rsz, int to )
{
    int sseq = -1;
    int rseq;
    int type;
    int r = ScompSend( query, qsz, &sseq, SCOMP_REQUEST );
    *resp = '\0';
    if ( SCOMP_ERR_OK == r && SCOMP_ERR_OK == ( r = ScompRecv( resp,
rsz, &rseq, &type, to ) ) )
    {
        if ( rseq != sseq )
            r = SCOMP_ERR_SEQ;
        else if ( type != SCOMP_RESPONSE )
            r = SCOMP_ERR_TYPE;
    }
    return r;
}

/* EOF */

```