Simple Point to Point Protocol (SPPP)

Chris Jenkins - Genesi USA, Inc.

Preliminaries

Terminology

- i.MX53™: Freescale ARM™ Cortex-A8 CPU, used on all Efika MX53 systems.
- UART: Universal Asynchronous Receiver / Transmitter. Translates physical serial protocol to / from parallel signals
- Housekeeping CPU (STM): Small(er) CPU on Efika MX53 systems responsible for low-level hardware initialization.

Prerequisites

- SPPP drivers: Linux kernel drivers for receiving / decoding datagrams from the STM. View the document “Building U-Boot and the Linux Kernel” for more information on how to get and build kernel source code.
- Modest familiarity with Linux kernel modules (see document “Building Kernel Modules”)
- Modulo (overflow) bit arithmetic

Overview: Simple Point to Point Protocol

- Protocol for communicating between the i.MX53 and the STM on Efika MX53 systems (such as the Drónov Slimbook)
    - Baudrate: 115200
- Linux Kernel subsystems involved include (but not necessarily limited to)
    - Keyboard
    - Trackpad
    - Real Time Clock (RTC)
    - Power.
- Communication is done via stateless, packet-based datagrams
    - Think UDP but simpler.
    - Important: SPPP has nothing to do with the data you are transmitting to a subsystem, only its “packaging” (receiver ID, beginning of message, end of message, simple checksum)

Overview: SPPP in the Linux Kernel

- SPPP module overview
    - `include/linux/sppp.h`: interface of sending / receiving SPPP messages, module global constants
        - implemented in `arch/arm/mach-mx5/spppdriver.c`
    - `arch/arm/mach-mx5/spppdriver.h`: UART-level register and bit-field definitions.
    - Drivers:
        - `drivers/input/keyboard/sppp_keyboard.c`
        - `drivers/input/mouse/sppp_trackpad.c`
        - `drivers/rtc/sppp_rtc.c`
        - `arch/arm/mach-mx5/sppp_power.c`

SPPP Clients

- SPPP clients are Linux kernel modules / drives
- They must include (and need only include) `include/linux/sppp.h`
- Module init function must register itself with SPPP main driver
  - Client must provide an ID to main SPPP driver and a callback function for receiving messages (not datagrams!) from the STM.
  - ID must be agreed statically, i.e. adding a new client means modifying main SPPP driver. Client IDs are defined in main driver header file.
- When a message is received from the STM, main SPPP driver unpacks the message and looks up the receiver in array of registered clients

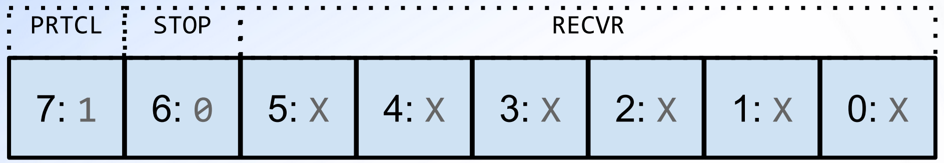
SPPP Datagram (types)

- There are three different types of datagram, each one byte long
    - START: indicates start of message
    - DATA: datagram containing message data
    - STOP: indicates end of message
- There can only be one START / STOP datagram per message

  There may be any number of DATA datagrams in between.
- Each datagram has reserved the MSB as a “protocol bit”. The START and STOP datagrams always have this bit set (these are the “protocol” datagrams); the DATA datagram never has this bit set.
    - This means a single DATA datagram can only contain 7 bits of message data (more on this below).
- Protocol datagrams have another special bit called the “stop bit” that follows immediately after the protocol bit.

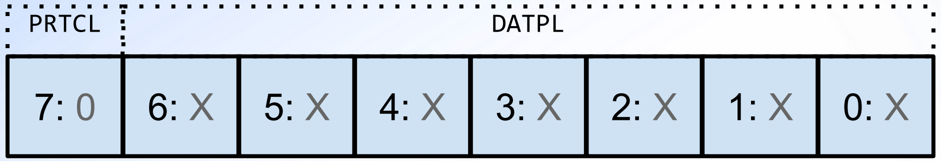
  This bit determines which type of protocol datagram it is: 0 for START, 1 for STOP
    - i.e. every protocol datagram reserves the two MSBs to say “I am a protocol datagram of type X”

SPPP Datagram (START)



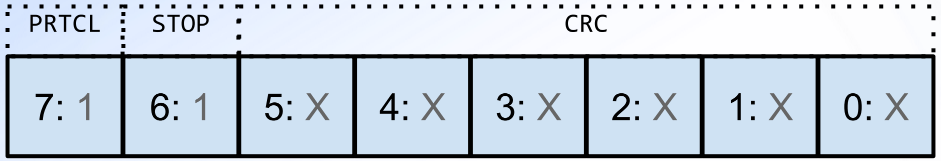
- The START datagram has the protocol bit (PRTCL) set and the stop bit (STOP) unset
- The remaining 6 bits (RECVR) are used to identify the receiver of the datagram
    - There is no protocol for dynamically setting, changing, or adding subsystem IDs. This is because the receivers of SPPP datagrams will be fixed when the hardware is created, i.e. will not change during the runtime of the software.

SPPP Datagram (DATA)



- The DATA datagram has the protocol bit unset
- The remaining 7 bits are the data payload (DATPL) being sent.
    - Should the message exceed 7 bits, it is “stream encoded”, i.e. split into 7 bit chunks each of which is sent in a DATA datagram. The message is then re-assembled after being transmitted via UART. The first bit of the message is the first bit of the first DATPL received.
    - This implies that there will be “carry” between datagrams. The very last carry is also stored, but separately.
    - An example is given after the next slide.

SPPP Datagram (STOP)



- The STOP datagram has the PRTCL bit set and the STOP bit set.
- The remaining 6 bits (CRC) are a **Cyclic Redundancy Check** to ensure against data corruption. This is done as a simple checksum
  - The sum is stored in 8 bits while the message is being sent / received
  - The initial value of the sum is the 6-bit receiver ID (RECVR) from the START datagram.
  - With each DATA datagram, the DATPL is added to the sum (overflow discarded)
  - The 6 LSBs of the sum are compared to the CRC. A discrepancy between these values means data was corrupted somewhere during transmission.

Example of SPPP Communication

Wish to send “SB” (**S**lim**B**ook): `01010011 01000010`. The column on the left shows the datagrams to be sent; on the right, the interpretation of the datagram. With each DATA, carry from previous DATA is written as message data and the new carry is stored*.

`10000001`	START; RECVR is 1 (arbitrary)
`00101001`	DATA; DATPL=`0101001`
`01010000`	DATA; DATPL=`1010000`
`01000000`	DATA; DATPL=`1000000` (last carry)
`11111010`	STOP; CRC=`111010`

```
Black: packaging              Underlined: carry            Green: CRC
Red: message padding          Blue: original message
```

Note that it’s the value of DATPL, not necessarily the message data, that determines the CRC. Also remember that the last carry is indeed stored and sent to reciever, but separately.

* Excepting first and every 8th DATA which are byte-aligned and do not need a carry from previous.

SPPP Code snippet: Receiving DATA

```
/* max input size reached */
if (sppp_rx->pos < MAX_RECV_PKG_SIZE) {
    /* add to CRC sum */
    sppp_rx->crc += data;
    if (sppp_rx->num != 0) {
        /* Shift current byte on right pos */
        sppp_rx->input[sppp_rx->pos] = ((data >> (7-sppp_rx->num)) | sppp_rx->carry);
        /* Input size */
        sppp_rx->pos++;
    }
    /* shift carry byte on right pos for next run */
    sppp_rx->carry = (data << (1 + sppp_rx->num));
    sppp_rx->num++;
    if (sppp_rx->num > 7)
        sppp_rx->num = 0;
} else {
    /* size is too big, drop this one */
    sppp_rx->sync = SPPP_NOSYNC;
}
```

SPPP Message Summary

- During kernel boot, the primary SPPP module makes an interrupt request on UART2
- Interrupt triggered means some number of datagrams are ready.
- Module global variable keeps track of message state.
- SPPP client drivers register themselves with the primary driver using a static (hardcoded) ID and callback for receiving messages (not datagrams) from STM
- For every datagram received by primary driver:
  - Check the type and handle accordingly (if datagrams sent are within max limit):
    - START => clear state; DATA => read data into buffer; STOP check CRC
  - When message has been successfully received, RECVR value used to fetch correct client. The callback provided by the client is then called with the message data.

genesi