Support CAN

Detailed Requirements and Design

rm6775-drad-1_1.doc

RM:	*6775*
Revision:	*1.1*
Date:	*8/23/2023*

TABLE OF CONTENTS

1. Overview

This project enables the Linux CAN device driver, as well as appropriate user-space components required to support CAN (using the CANSocket Linux APIs), in the Linux BSP for the STM32H7-EVAL board.

2. Requirements

2.1. Detailed Requirements

The following are the requirements for this project:

1. Provide a Linux demo project combining all the requirements in this project.
    - *Rationale*: Needed to let Customer integrate results of this project into target embedded application.
      Implementation: Section: "Design: Demo project".
      Test: Section: "Test Plan: Demo Project".

2. Enable and configure Linux CAN device driver for the STM32H7 CAN controller.

    - *Rationale*: Explicit Customer requirement.
      Implementation: Section: "Design: Linux CAN Device Driver".
      Test: Section: "Test Plan: Linux CAN Driver".

3. Port CANSocket to the Linux STM32H7 BSP.

    - *Rationale*: Explicit Customer requirement.
      Implementation: Section: "Design: CANSocket".
      Test: Section: "Test Plan: CANSocket".

4. Validate successful execution of the test suite from the SocketCAN package.

    - *Rationale*: Explicit Customer requirement.
      Implementation: Section: "Design: CANSocket Test Suite".
      Test: Section: "Test Plan: CANSocket Test Suite".

2.2. Detailed Non-Requirements

The following are the non-requirements for this project that may otherwise not be obvious:

1. None

3. Design

3.1. Detailed Design

3.1.1. Design: Demo project

This project will enable the required CAN functionality in Linux configuration ("embedded project") called `rootfs`, which resides in a `projects/rootfs` directory, relative to the top of the Linux STM32H7 installation.

3.1.2. Design: Linux CAN Device Driver

The STM32H7 `M_CAN` interface driver `linux/drivers/net/can/m_can` will be enabled, and the respective changes will be added to the kernel `.dts` file.

The `M_CAN` functionality will be enabled in the Linux kernel configuration as follows:

- Go to `Networking support`
- Enable `CAN bus subsystem support` (`CONFIG_CAN`)
- Go to `Device Drivers` -> `Network device support`
- Enable `CAN Device Drivers` (`CONFIG_CAN_DEV`)
- Go to `Device Drivers` -> `Network device support` -> `CAN Device Drivers`
- Enable `CAN device drivers with Netlink support` (`CONFIG_CAN_NETLINK`) and `Bosch M_CAN support` (`CONFIG_CAN_M_CAN`)
- Go to `Device Drivers` -> `Network device support` -> `CAN Device Drivers` -> `Bosch M_CAN support`
- Enable `Bosch M_CAN support for io-mapped devices` (`CONFIG_CAN_M_CAN_PLATFORM`)

3.1.3. Design: CANSocket

The CAN socket API, described in details in `linux/Documentation/networking/can.txt`, will be enabled using the `Raw CAN Protocol (raw access with CAN-ID filtering)` and `Broadcast Manager CAN Protocol (with content filtering)` configuration options in the `Networking support` -> `CAN bus subsystem support` configuration menu.

3.1.4. Design: CANSocket Test Suite

The `can-utils` and `can-tests` packages will be used for verification of the functionality implemented in this project.

3.2. Effect on Related Products

This project makes the following updates in the related products:

- None

3.3. Changes to User Documentation

This project updates the following user documents:

- None

3.4. Alternative Design

The following alternative design approaches were considered by this project but then discarded for some reason:

- None

4. Test Plan

4.1. Secure Download Area

The downloadable materials developed by this project are available from a secure Web page on the Emcraft Systems web site. Specifically, proceed to the following URL to download the software materials:

for the STM32H7-EVAL BSP:

- https://www.emcraft.com/stm32h7addon/stm32h7-eval/can/

The page is protected as follows:

- Login: `CONTACT EMCRAFT FOR DETAILS`
- Password: `CONTACT EMCRAFT FOR DETAILS`

4.2. Downloadable Files

The following files are available from the secure download area:

For release 3.0.1:

- `3.0.1/projects-rootfs-3.0.1.patch` - patch to the `rootfs` project;
- `3.0.1/rootfs.uImage` - prebuilt bootable Linux image;
- `3.0.1/u-boot.bin` - prebuilt bootable U-Boot image;

Refer to the below sections for the instructions on how to install and use these files.

4.3. Test Set-Up

4.3.1. Hardware Setup

The following hardware setup is required for the STM32H753-EVAL boards:

- The STM32H753-EVAL board, with the serial console attached.
  Preparations are slightly different in case of the EVAL boards MB1246-B and MB1246-Exx:
  - MB1246-B. Close SB50 and JP2. Soldering is required to close SB50.
    - **PRECAUTION** Don't use USB on CN18 while SB50 is closed.
  - MB1246-Exx. Close JP2.
    - **PRECAUTION** Don't use USB on CN18 while JP2 is closed.
- USB to CAN Adapter connected to the PC (via USB) and to CN2 of the EVAL board.

  The PEAK PCAN-USB IPEH-002021 can be used (https://www.peak-system.com/PCAN-USB.199.0.html?&L=1).
  If the PEAK PCAN-USB is used, it should be connected to the EVAL board by a direct DB9F-DB9F cable.

4.3.2. Software Setup

The following software setup is required:

1. Download the files listed in Section: "Downloadable Files" to the top of the Linux STM32H7 installation.
2. Install the BSP, as per the respective "Installing and activating cross development environment" document in the "Docs" section on the Emcraft site.
3. From the top of the Linux installation, activate the Linux cross-compile environment by running:

```
$ . ./ACTIVATE.sh
```

4. From the top of the Linux installation, go to the `projects` sub-directory, and patch the `rootfs` project:

```
$ cd projects/
$ patch -p1 < ../projects-rootfs-can-3.0.1.patch
```

5. On the Linux PC intended for execution of the CANsocket test suite, ensure that the following software is installed (Emcraft used Linux PC running the `Fedora 16` (3.1.0-7.fc16.i686.PAE) installation; the other Linux distributives should work too, but may require some additional steps like compilation and installation of the CAN framework kernel modules):
    1. Install `can-utils` package on the Linux PC (commands below are for a Fedora host):

```
$ sudo yum install can-utils
...
$
```

    2. Install and build `can-tests` on the Linux PC:

```
$ cd ~
$ git clone https://github.com/linux-can/can-tests.git
$ cd can-tests
$ make
$ sudo DESTDIR=/usr PREFIX= make install
```

    3. Load the CAN kernel modules on the Linux PC:

```
$ sudo modprobe can
$ sudo modprobe can-raw
$ sudo modprobe slcan
```

6. Connect the PEAK PCAN-USB adapter to the Linux PC and configure it as follows:
    1. Get the PEAK PCAN-USB network interface name (in the example below it is `can0`):

```
$ dmesg | tail
[9948637.678251] usb 3-4.4.2: new full-speed USB device number 93 using xhci_hcd
[9948637.755452] usb 3-4.4.2: New USB device found, idVendor=0c72, idProduct=000c,
bcdDevice=53.ff
[9948637.755457] usb 3-4.4.2: New USB device strings: Mfr=10, Product=4, SerialNumber=0
[9948637.755460] usb 3-4.4.2: Product: PCAN-USB
[9948637.755462] usb 3-4.4.2: Manufacturer: PEAK-System Technik GmbH
[9948637.765874] peak_usb 3-4.4.2:1.0: PEAK-System PCAN-USB adapter hwrev 83 serial FFFFFFFF
(1 channel)
[9948637.766099] peak_usb 3-4.4.2:1.0 can0: attached to PCAN-USB channel 0 (device 255)
```

    2. Configure the PEAK PCAN-USB adapter to run with a 1Mbps CAN-bus speed, enable the corresponding network interface:

```
$ sudo ip link set can0 type can bitrate 1000000
$ sudo ifconfig can0 up
```

4.4. Detailed Test Plan

4.4.1. Test Plan: Demo Project

Perform the following step-wise test procedure:

1. Go to the `projects/rootfs` directory, build the loadable Linux image (`rootfs.uImage`) and copy it to the TFTP directory on the host:

```
$ cd projects/rootfs
$ make
```

2. Power cycle the board with installed U-Boot. While U-Boot is coming up, press any key on the serial console to enter the U-Boot command line interface:

```
Model: STMicroelectronics STM32H753i-EVAL board
DRAM:  32 MiB
Flash: 2 MiB
MMC:   STM32 SDMMC2: 0
Loading Environment from MMC... OK
In:    serial@40011000
Out:   serial@40011000
Err:   serial@40011000
Net:   eth0: ethernet@40028000
Hit any key to stop autoboot:  0
STM32H7-EVAL U-Boot >
```

For U-Boot installation instructions refer to https://emcraft.com/stm32h7-evk-board/running-linux-on-stm32h753i-eval-in-5-minutes.

3. Boot the loadable Linux image (`rootfs.uImage`) to the target via TFTP and validate that it boots to the Linux shell:

```
STM32H7-EVAL U-Boot >
STM32H7-EVAL U-Boot > setenv -f ethaddr
STM32H7-EVAL U-Boot > setenv ethaddr 34:56:78:9a:bc:12
STM32H7-EVAL U-Boot > setenv ipaddr 172.17.0.166
STM32H7-EVAL U-Boot > setenv serverip 172.17.0.1
STM32H7-EVAL U-Boot > setenv image rootfs.uImage
STM32H7-EVAL U-Boot > saveenv
STM32H7-EVAL U-Boot > run netboot
ethernet@40028000 Waiting for PHY auto negotiation to complete. done
Using ethernet@40028000 device
TFTP from server 172.17.0.1; our IP address is 172.17.0.166
Filename 'rootfs.uImage'.
Load address: 0xd0c00000
Loading: #################################################################
...
/ #
```

4.4.2. Test Plan: Linux CAN Driver

Perform the following step-wise test procedure:

1. In the kernel bootstrap messages, validate that the CAN driver has been successfully installed and activated:

```
...
[    0.747051] CAN device driver interface
[    0.765873] m_can_platform 4000a000.can: m_can device registered (irq=32, version=32)
...
[    5.115042] can: controller area network core
[    5.119874] NET: Registered PF_CAN protocol family
[    5.132211] can: raw protocol
```

```
[    5.135056] can: broadcast manager protocol
[    5.139241] can: netlink gateway - max_hops=1
...
```

4.4.3. Test Plan: CANSocket

Perform the following step-wise test procedure:

1. On the target, configure the CAN network:

```
/ #  ip link set can0 type can bitrate 1000000
/ # ifconfig can0 up
```

2. Test target to Linux PC transfers:
   - Run the capture utility on the Linux PC:

```
$ candump can0
```

   - Send packets from the target to the host:

```
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.00
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.01
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.02
/ # cansend can0 12345678#99.AA.BB.CC.DD.EE.FF.03
```

   - Validate that the packets have been captured on the Linux PC:

```
  can0  12345678   [8]  99 AA BB CC DD EE FF 00
  can0  12345678   [8]  99 AA BB CC DD EE FF 01
  can0  12345678   [8]  99 AA BB CC DD EE FF 02
  can0  12345678   [8]  99 AA BB CC DD EE FF 03
```

   - On the host, stop the capture utility by pressing `Ctrl-C`:

```
^C
$
```

3. Test Linux PC to target transfers:
   - Run the capture utility on the target:

```
/ # candump can0
```

   - Send packets from the Linux PC to the target:

```
$ cansend can0 123abcde#11.22.33.44.56.78.90.01
$ cansend can0 123abcde#11.22.33.44.56.78.90.03
$ cansend can0 123abcde#11.22.33.44.56.78.90.05
$ cansend can0 123abcde#11.22.33.44.56.78.90.07
```

   - Validate that the packets have been captured on the target:

```
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 01
```

```
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 03
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 05
  can0  123ABCDE   [8]  11 22 33 44 56 78 90 07
```

- On the target, stop the capture utility by pressing `Ctrl-C`:

```
^C
/ #
```

4.4.4. Test Plan: CANSocket Test Suite

Perform the following step-wise test procedure:

1. Run the `tst-raw` Linux PC to target test:
   - On the target:

```
/ # tst-raw -i can0
```

   - On the Linux PC:

```
$ sudo tst-raw-sendto -i can0
```

   - Observe the test data on the target, then press `Ctrl-C` and complete the test:

```
123  [3] 11 22 33
^C
/ #
```

2. Run the `tst-raw` target to Linux PC test:
   - On the Linux PC:

```
$ sudo tst-raw -i can0
```

   - On the target:

```
/ # tst-raw-sendto -i can0
```

   - Observe test data on the Linux PC, then press `Ctrl-C` and complete the test:

```
(1691612954.686415) 123  [3] 11 22 33
^C
$
```

3. Run the `tst-packet` Linux PC to target test:
   - On the target:

```
/ # tst-packet -i can0
```

   - On the Linux PC send a packet, then press `Ctrl-C` and complete the test:

```
$ sudo tst-packet -i can0 -s
^C
$
```

   - Observe the test packet on the target, then press `Ctrl-C` and complete the test:

```
123  [2] 11 22
```

```
^C
/ #
```

4. Run the `tst-packet` target to Linux PC test:
   - On the Linux PC:

```
$ sudo tst-packet -i can0
```

   - On the target, send a packet, then press `Ctrl-C` and complete the test:

```
/ # tst-packet -i can0 -s
^C
/ #
```

   - Observe the test packet on the Linux PC, then press `Ctrl-C` and complete the test:

```
123  [2] 11 22
^C
$
```

5. Run the `tst-filter` test on the target:

```
/ # tst-filter can0
---
testcase  0 filters : can_id = 0x00000123 can_mask = 0x000007FF
testcase  0 sending patterns ... ok
testcase  0 rx : can_id = 0x00000123 rx = 1 rxbits = 1
testcase  0 rx : can_id = 0x40000123 rx = 2 rxbits = 17
testcase  0 rx : can_id = 0x80000123 rx = 3 rxbits = 273
testcase  0 rx : can_id = 0xC0000123 rx = 4 rxbits = 4369
testcase  0 ok
---
testcase  1 filters : can_id = 0x80000123 can_mask = 0x000007FF
testcase  1 sending patterns ... ok
testcase  1 rx : can_id = 0x00000123 rx = 1 rxbits = 1
testcase  1 rx : can_id = 0x40000123 rx = 2 rxbits = 17
testcase  1 rx : can_id = 0x80000123 rx = 3 rxbits = 273
testcase  1 rx : can_id = 0xC0000123 rx = 4 rxbits = 4369
testcase  1 ok
---
testcase  2 filters : can_id = 0x40000123 can_mask = 0x000007FF
testcase  2 sending patterns ... ok
testcase  2 rx : can_id = 0x00000123 rx = 1 rxbits = 1
testcase  2 rx : can_id = 0x40000123 rx = 2 rxbits = 17
testcase  2 rx : can_id = 0x80000123 rx = 3 rxbits = 273
testcase  2 rx : can_id = 0xC0000123 rx = 4 rxbits = 4369
testcase  2 ok
---
<...>
---
testcase 15 filters : can_id = 0xC0000123 can_mask = 0xC00007FF
testcase 15 sending patterns ... ok
testcase 15 rx : can_id = 0xC0000123 rx = 1 rxbits = 4096
testcase 15 ok
---
testcase 16 filters : can_id = 0x00000123 can_mask = 0xDFFFFFFF
testcase 16 sending patterns ... ok
testcase 16 rx : can_id = 0x00000123 rx = 1 rxbits = 1
testcase 16 ok
---
testcase 17 filters : can_id = 0x80000123 can_mask = 0xDFFFFFFF
testcase 17 sending patterns ... ok
testcase 17 rx : can_id = 0x80000123 rx = 1 rxbits = 256
testcase 17 ok
---
/ #
```

6. Run the `tst-rcv-own-msgs` test on the target:

```
/ # tst-rcv-own-msgs can0
Starting PF_CAN frame flow test.
checking socket default settings ... ok.
check loopback 0 recv_own_msgs 0 ... ok.
check loopback 0 recv_own_msgs 1 ... ok.
check loopback 1 recv_own_msgs 0 ... ok.
check loopback 1 recv_own_msgs 1 ... ok.
PF_CAN frame flow test was successful.
/ #
```