CDAC – BANGALORE

DISSERTATION

on

CAN DRIVER DEVELOPMENT FOR LPC1769 USING THE FREERTOS+IO FRAMEWORK

undertaken at

CDAC, Bangalore

Submitted in partial fulfilment of the requirement for the award of

DIPLOMA IN EMBEDDED SYSTEM DESIGN

Under the esteemed guidance of

Mr. Babu Krishnamurthy

Submitted by

Nishad Sabnis

Bharath Meduri

Barath Bushan

Niharika Singh

ACKNOWLEDGEMENT

This project would not have been possible without the help of many people and we would like to take this opportunity to thank all of them.

Firstly, our project supervisor, Mr Babu Krishnamurthy whose guidance, constant encouragement and high standards helped us every time we faced a problem.

We would also like to thank all the faculty and staff at CDAC for providing us with all the hardware and the excellent lab facilities.

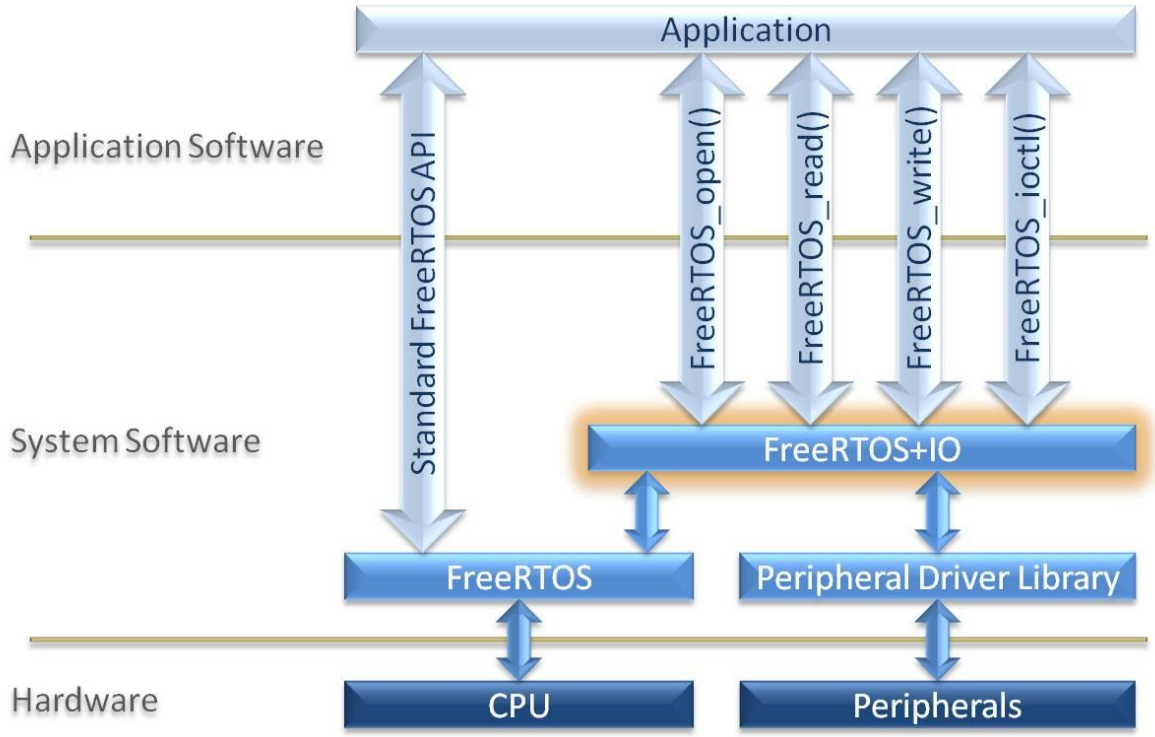
Last, but certainly not the least, we would like to thank our family members and our batch mates at CDAC without whose support we could never have finished the project.

REQUIREMENTS AND IMPLEMENTED FEATURES

Requirements:

FreeRTOS is a popular real-time operating system for embedded devices, which has been ported to a huge variety of microcontrollers.

FreeRTOS+IO provides a Linux/POSIX like open(), read(), write(), ioctl() type interface to peripheral driver libraries. It sits between a peripheral driver library and a user application to provide a single, common, interface to all supported peripherals across all supported platforms. The current board support package implementation(s) support UART, I2C and SPI operation, in both polled and interrupt driven modes.



Thus, there is a need to add to the current board support package by including support for other peripherals. As part of our project, support for the CAN peripheral was added to FreeRTOS+IO.

Hardware Details:

The following hardware components were used to setup the testing of the project:

- LPC 1769 Xpresso Stamp x 2
- Embedded Artists LPCXpresso Baseboard x 2
- TJA1040 CAN Transceiver(onboard the base board) x 2

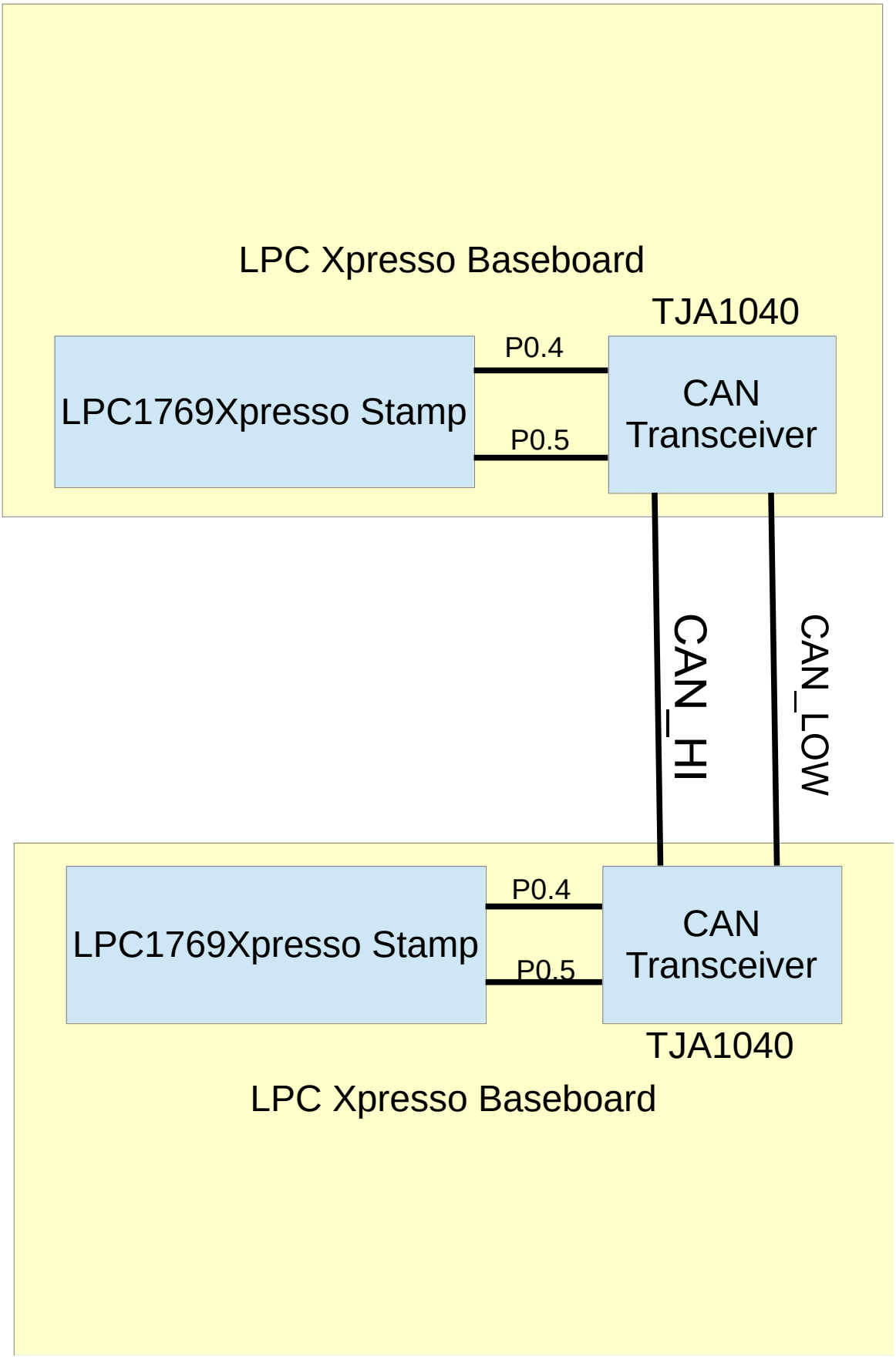
Software Details:

- LPCXpresso IDE

Implemented Features:

- CAN Driver which can be configured to disable/enable and change the acceptance filter on the fly.
- Both interrupt driven and polling mode possible.
- The transfer modes use semaphores/mutexes internally, so the user task does not have to worry about more than one device of the same type.
- Built-in facility to check the hardware in self-test mode.

HIGH LEVEL DESIGN



- As can be seen in the figure above, two identical setups are used to communicate with each other.
- Each setup consists of an LPC1769Xpresso stamp, mounted on a LPCXpresso Baseboard which has an onboard TJA1040 CAN transceiver.
- The LPC1769 from NXP has 2 CAN controllers, CAN1 and CAN2 but no on-board transceiver.
- A transceiver is needed so as to adjust bus levels according to the physical specifications of CAN and to protect the CAN controller.
- The LPCXpresso Baseboard routes the CAN2 controller RD and TD pins to the appropriate pins of the Can transceiver present on the board.
- The outputs from the CAN transceiver are then joined to the outputs of the CAN transceiver on the other baseboard.

Code Design

The workspace consists of four different projects which are used together to get the code working.

- **CMSISv2p00_LPC17xx:** This is the project that defines the entire CMSIS code specific to the Cortex-M3 core.

- **FreeRTOS-Plus-Demo-1:** This project is the one that has all the user-space tasks created to test the CAN driver.

- **FreeRTOS-Products:** This project consists of the actual source files for FreeRTOS+IO extension.

- **lpc17xx.cmsis.driver.library:** This is the peripheral driver library that is used as a low-level abstraction while developing the FreeRTOS+IO framework driver.

LOW LEVEL DESIGN

In this section, the important source files and the code flow will be explained.

Main.c – This is the entry point into the code. It sets up certain GPIO and software timers. Then calls the function which creates the CAN application task which is going to make use of the CAN driver. Lastly, calls vTaskStartScheduler() which gives control to the real-time kernel.

Can.c – In this file, the task that calls the FreeRTOS+IO API for the CAN peripheral is defined. The task is given a priority and a stack size.

Can.h – The prototypes for all functions defined in can.c and all the macros being used in can.c are defined here.

FreeRTOS_DriverInterface.c – The FreeRTOS_open() and FreeRTOS_ioctl() generic functions are defined here. Data extraction from the BSP is done.

LPCXPresso17xx_BSP.h – This is the main board support package file that holds a list of the peripherals available and supported with this particular package.

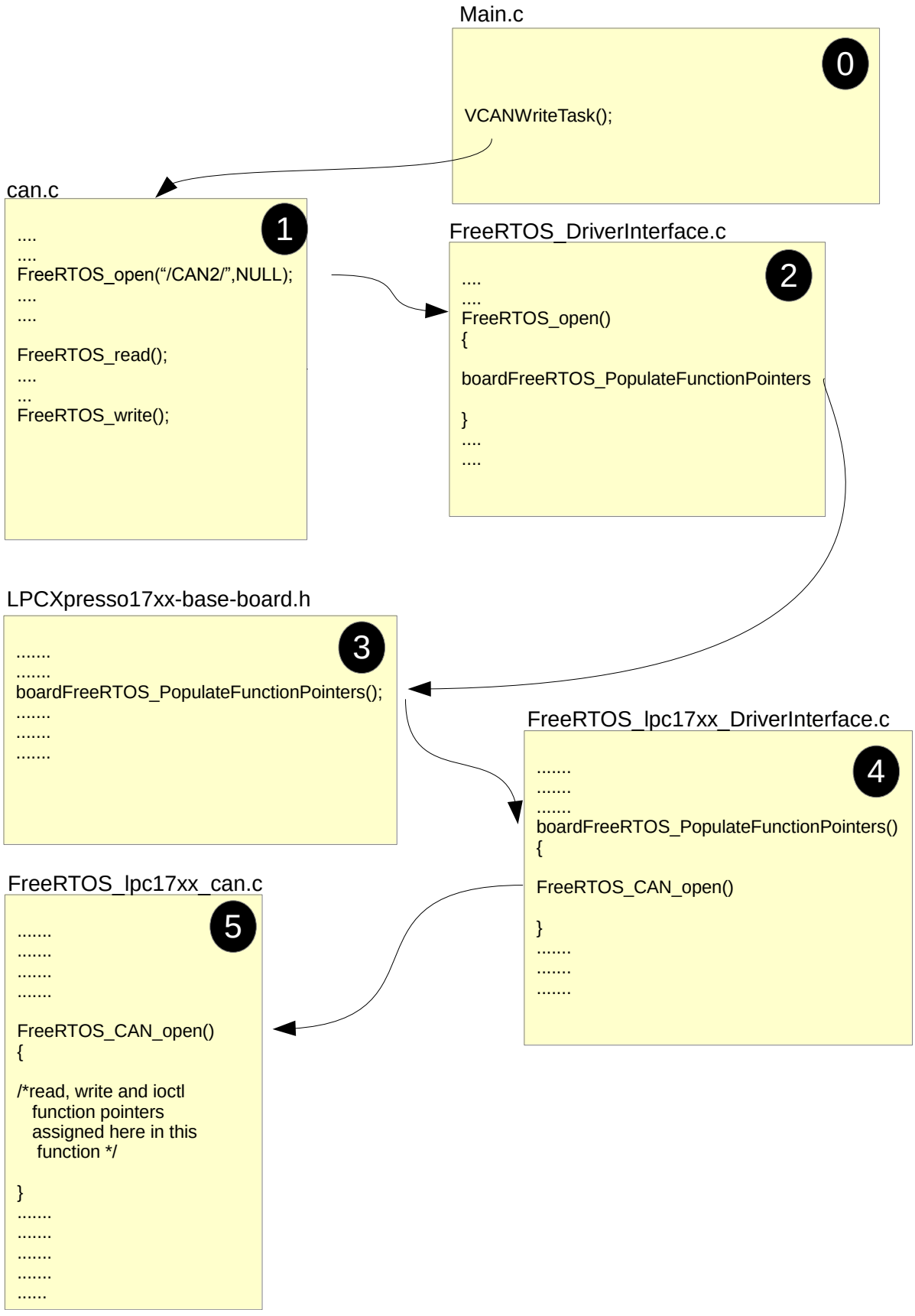
FreeRTOS_lpc17xx_DriverInterface.c – Switches control for the FreeRTOS + IO generic API to the FeeRTOS + IO device specific API.

FreeRTOS_lpc17xx_can.c – Our implementation of the FreeRTOS+IO CAN driver which has the device-specific open(), read(), write() and ioctl functions.

Lpc17xx_can.c – The CMSIS driver library for CAN which defines some APIs being used in the FreeRTOS+IO CAN driver

Uart.c – A small UART peripheral library so that UART can be used easily on the screen for debugging purposes.

FreeRTOS_open()



The above diagram can be explained as follows.

0. In the main function, a function to create the CAN task is called.

1. When a user application makes a call to FreeRTOS_open() control transfers to the function defined in FreeRTOS_DriverInterface.c.

2. In the open function, first the type of peripheral and peripheral number is extracted. Then the peripheral control structure (the device) is allocated memory and certain initializations are done. Once the control structure is initialized, a call is made to boardFreeRTOS_PopulateFunctionPointers().

3. This function extracts data from the BSP which is defined in LPCXpresso17xx-BSP.h. This data consists of a list of available devices on the board, pin configurations for each of the devices and some constants associated with the devices.

4. Using this information, it initializes the peripheral-specific parts of the control structure and then calls the peripheral specific FreeRTOS_CAN_open().

5. FreeRTOS_CAN_open() is defined in FreeRTOS_lpc17xx_can.c and it does the pin configurations for the given peripheral number at the given base address. The baudrate is also set up. In addition, peripheral-specific read(), write() and ioctl() functions are assigned to the peripheral control structure. In the end, the CAN2 controller is enabled and control goes back to the user application.

Once a call to open() has been made, the task can now call read(), write() and ioctl() as required to set up the peripheral for a particular transfer mode or speed.

The flow for read() and write() is much simpler as it is dictated by a simple macro definition which involves a function pointer call.

TEST CASES

Scenario	Input Given	Output expected	Output obtained	Comments
1 Acceptance filter -> Bypass mode, CAN Controller -> Self test mode, Interrupt -> Disabled.	LPC_CAN2-> CMR = 0x21	LPC_CAN2-> SR &= RBS = 1	LPC_CAN2-> SR &= RBS = 0	In self test mode the data sent by the CAN Controller is buffered back into the receive buffer. This mode is enabled only if STM bit of CAN mode register is set to 1. And SRR command is given to command register of CAN Controller.
2 Acceptance filter -> Bypass mode, CAN Controller -> Self test mode, Interrupt -> Disable.	LPC_CAN2-> CMR = 0x10	LPC_CAN2-> SR &= RBS = 1	LPC_CAN2-> SR &= RBS = 1	Command transit request does not work with self test mode. A message transmission is initiated by setting Self Reception Request bit in

				command register.
3. Acceptance filter -> Bypass mode, CAN Controller -> Self test mode, Interrupt -> Enable.	FreeRTOS_ioctl (“CAN2”, use_interrupt,1) LPC_CAN2-> CMR = 0x10	LPC_CAN2-> SR &= RBS = 1	LPC_CAN2-> SR &= RBS = 1	Ioctl request will enable the RIE bit in interrupt enable register of CAN Controller. The data available in the receive buffer of CAN controller is collected by the interrupt handler and it will request RRB request to CAN Command register to free the buffers.
4 Communication between two boards. Acceptance filter -> Bypass mode, CAN Controller -> normal mode, Interrupt -> Disable.	On board 1 LPC_CAN2-> CMR = 0x21	On board 2 LPC_CAN2-> SR &= RBS = 1	On board 2 LPC_CAN2-> SR &= RBS = 1	Two lpc 1769 boards are used for CAN communication. One board will send the data and other will receive. Acceptance filter is used in bypass mode hence all messages will be accepted.

5. Communication between two boards. Acceptance filter -> bypass mode, CAN Controller -> normal mode, Interrupt -> Enable.	On board 1 LPC_CAN2-> CMR = 0x21 On board 2 FreeRTOS_ioctl (“CAN2”, use_interrupt,1)	On board 2 LPC_CAN2-> SR &= RBS = 1	On board 2 LPC_CAN2-> SR &= RBS = 1	On board 2 ioctl request will enable the RIE bit in interrupt enable register of CAN Controller.
6. Communication between two boards. Acceptance filter -> Normal mode, CAN Controller -> normal mode, Interrupt -> Enable	On board 1 LPC_CAN2-> TID1 ->0x123 LPC_CAN2-> CMR = 0x21 On board 2 FreeRTOS_ioctl (“CAN2”, use_interrupt,1)	On board 2 LPC_CAN2-> SR &= RBS = 1	On board 2 LPC_CAN2-> SR &= RBS = 0	Acceptance filter in normal mode implement search algorithm supporting a large number of CAN Identifiers. The id available in the TID register does not match with the available identifiers.
7 Communication between two boards. Acceptance filter -> Normal mode, CAN Controller -	On board 1 LPC_CAN2-> TID1 ->0x121 LPC_CAN2-> CMR = 0x21	On board 2 LPC_CAN2-> SR &= RBS = 1	On board 2 LPC_CAN2-> SR &= RBS = 1	Message id available in TID matched with the available identifiers

> normal mode, Interrupt -> Enable	On board 2 FreeRTOS_ioctl (“CAN2”, use_interrupt,1)			

LESSONS LEARNT

- **BOOTLOADER ENABLING :**
  - On loading certain programs with a bug, a hard fault exception was encountered.
  - After this hard-fault, if we tried to flash the board again, it did not flash properly.
  - This was because the buggy program had changed the functions of the pins which do the debugging (JTAG/SWD).
  - In order to initialize the pins properly, the ISP bootloader had to be enabled.
  - This was done by pressing down on the BL_EN switch on the baseboard and then cycling power.
- **UART DEBUG SUDDENLY STOPPED WORKING :**
  - Initially, the UART program written to display debug messages on the screen was tested and working fine.
  - When this was included with the rest of the CAN source files, it started creating problems as the CAN controller was going into reset mode and the UART debug messages stopped completely.
  - After referring to the user manual for the LPC1769, we realized that the CAN1 and UART3 pins we were using were the same pins with different function numbers.
  - Changing the controller to CAN2 solved the problem and debug messages could be seen again.
- **SCHEMATICS FOR THE BASEBOARD**
  - All the example codes given for the CAN controller use pins p2.7 and p2.8 when they are selecting the CAN2 controller.
  - When this setup was used with the baseboard, we faced a few problems as the CAN controller kept going into reset mode every time we tried to pass a command to the command register.
  - Upon studying the datasheet further, it was seen that CAN2 controller has two RD and TD outputs and only one of them is routed to the CAN transceiver. (p0.4 and p05 instead of p2.7 and p2.8)

ADDITIONAL INFORMATION

Some of the important structures used in the code

```
typedef enum {
	STD_ID_FORMAT = 0,  /**< Use standard ID format (11 bit ID) */
	EXT_ID_FORMAT = 1   /**< Use extended ID format (29 bit ID) */
} CAN_ID_FORMAT_Type;
```

```
typedef enum {
	DATA_FRAME = 0,     /**< Data frame */
	REMOTE_FRAME = 1    /**< Remote frame */
} CAN_FRAME_Type;

typedef enum {
	CAN_Normal = 0,     /**< Normal Mode */
	CAN_AccOff,         /**< Acceptance Filter Off Mode */
	CAN_AccBP,                /**< Acceptance Fileter Bypass Mode */
	CAN_eFCAN                 /**< FullCAN Mode Enhancement */
} CAN_AFMODE_Type;

typedef enum {
	CAN_OPERATING_MODE = 0,   /**< Operating Mode */
	CAN_RESET_MODE,                 /**< Reset Mode */
	CAN_LISTENONLY_MODE,            /**< Listen Only Mode */
	CAN_SELFTEST_MODE,              /**< Seft Test Mode */
	CAN_TXPRIORITY_MODE,            /**< Transmit Priority Mode */
	CAN_SLEEP_MODE,                 /**< Sleep Mode */
	CAN_RXPOLARITY_MODE,            /**< Receive Polarity Mode */
	CAN_TEST_MODE                   /**< Test Mode */
} CAN_MODE_Type;
```

```
typedef struct {
	uint32_t id;                /**< 29 bit identifier, it depend on "format"
                                     value
                                     - if format = STD_ID_FORMAT, id
                                       should  be 11 bit identifier
                                     - if format = EXT_ID_FORMAT, id
                                     should  be 29 bit identifier
                                  */
	uint8_t dataA[4];           /**< Data field A */
	uint8_t dataB[4];           /**< Data field B */
	uint8_t len;                /**< Length of data field in bytes, should be:
                                     - 0000b-0111b: 0-7 bytes
                                     - 1xxxb: 8 bytes
                                  */
	uint8_t format;             /**< Identifier Format, should be:
                                     - STD_ID_FORMAT: Standard ID - 11 bit
```

```
                                                - EXT_ID_FORMAT: Extended ID - 29 bit
                                               */
    uint8_t type;                  /**< Remote Frame transmission, should be:
                                                - DATA_FRAME: the number of
                                                data bytes called out by the DLC
                                                 field are send from the
                                                 CANxTDA and CANxTDB
                                                 registers
                                                 - REMOTE_FRAME: Remote
                                                 Frame is sent
                                               */
} CAN_MSG_Type;
```

REFERENCES

LPC1769:

- PDFs:
  - lpcxpresso.getting.started
  - LPC17xx USER MANUAL
  - Board schematics for LPC1769 Xpresso board
- URL's:
  - http://www.embeddedartists.com/products/lpcxpresso/lpc1769_xpr.php

BASEBOARD:

- PDFs:
  - LPCXpresso baseboard Getting Started
  - LPCXpresso baseboard schematics
- URL's:
  - http://www.embeddedartists.com/products/lpcxpresso/xpr_base.php

CAN:

- PDF:
  - BOSCH CAN specification
  - LPC17xx USER MANUAL
  - CAN Basics(Microchip Application Note AN713)

FreeRTOS

- **PDF:**
  - Using FreeRTOS with LPC17xx
  - FreeRTOS API Reference
  - FreeRTOS Manual
- URL:
  - http://www.freertos.org/