Release Notes for the

Windows Embedded Compact 2013

Board Support Package 2.1

For TQMA335X platform

4/30/2015

Contents

1. Prerequisites

   Make sure you have at least the following software installed on your PC:
   - Microsoft Visual Studio Professional 2012
   - Microsoft Visual Studio Professional 2012 Update 4
   - Platform Builder for Visual Studio 2012
   - Windows Embedded Compact 2013 Update 5
   - All available Windows Embedded Compact 2013 Monthly Updates / Product Update Rollups (at least) for ARM.

2. BSP Installation / Image Creation

   In order to install the BSP, copy the entire "WINCE800" folder (contained in the "BSP" folder) of the Package over the existing WINCE800 folder in your WEC 2013 installation location. This adds the BSP under `WINCE800\platform\AM335x_TQS`, as well as an example OSDesign under `WINCE800\OSDesigns\TQMA335X`.

   It is recommended to use the supplied example OSDesign as a starting point. Please follow these steps:
   - Open Visual Studio and select FILE -> Open -> Project/Solution. Select the file `WINCE800\OSDesigns\TQMA335X\TQMA335X\TQMA335X.pbxml`.
   - The Example OSDesign should be opened and Visual Studio should switch to the PlatformBuilder Configuration.
   - Select BUILD -> Configuration Manager… and make sure "`AM335x_TQS ARMV7 Release`" is selected as Active solution configuration.
   - Select BUILD -> Build Solution. This will SYSGEN and Build the Bootloader(s) and Image.
   - After some time (depending on your system), the build process should finish. You can now proceed to booting the device (see 3.).

   Alternatively, you can create your own custom solution instead of using the provided example OSDesign:
   - Open Visual Studio and select FILE -> New -> Project.
   - Select Other Project Types -> Platform Builder and give it a name.
   - In the OSDesign Wizard, select AM335x_TQS: ARMV7.

3. Boot/Deployment

   3.1. Boot from external SD Card

   In order to use an external SD Card for booting, follow these steps:

   3.1.1. Prepare the card (only once)

   - Insert SD Card into PC cardreader.
   - Note the drive letter of the inserted card.

- Open a command shell.
- Navigate to the BSP "SCRIPTS" subfolder (e.g. `WINCE800\platform\AM335x_TQS\SCRIPTS`).
- Prepare the SD Card with the command "`prepsd drive letter`" (e.g. `prepsd L:`)

3.1.2. Copy the files to the card and boot (every time the Bootloader/Image changes)

- Copy the files "`MLO`", "`EBOOTSD.nb0`" and "`NK.bin`" to the SD Card. These files can be taken from the "Bootloader" and "Images" –folders of the BSP package or from the Release Directory of an OSDesign after building it (see 2.).
- When taking the "NK.bin" from the Release Directory, make sure the image was not built with the "write image to eMMC flash" option active. When using the image from the BSP package, select the "NK.bin" from "RAM_SDCard" subfolder
- Eject the SD Card from PC cardreader.
- Insert the card into the MBa335x SDCard slot (X10).
- Make sure dip switch S2 is configured for boot from MMC0/SD first (S2-[5:1] = 10111). See hardware documentation for details.
- Make sure dip switch S1 is configured to select 24MHz Quartz frequency (S1-[8-7] = 01). See hardware documentation for details.
- Connect a serial cable between the debug port (X15) and PC.
- Configure your terminal application for serial (115200Baud, 8, N, 1).
- Power up the board.
- Windows Embedded Compact should boot up. The default configuration is to autoboot "NK.bin" from the card. This can be confirmed by examining the debug messages on the serial console (X-Loader -> Eboot -> NK Image).

3.2. Boot from internal eMMC

In order to boot from internal eMMC, the eMMC X-Loader and Eboot must first be programmed using external SD Card boot. Follow these steps:

3.2.1. Prepare the external SD Card.

- Build a Retail Image (see 2.)
- Follow the same steps as under 3.1 (Boot from external SD Card), but copy the files "`MLO`" and "`EBOOTEMMC.nb0`" to the card instead.
- Rename the file "`EBOOTEMMC.nb0`" on the card to "`EBOOTSD.nb0`".

3.2.2. Boot from external card and download eMMC X-Loader to the board

- Connect the board to your network infrastructure using Ethernet port 1 (X11) of the board.
- Boot the board from the external SD card.

- Enter the EBOOT menu by pressing “Space” in the serial console when prompted (“Hit space to enter configuration menu”).
- Make sure the network configuration is set according to your infrastructure by entering the “Network Settings” menu item pressing “4”. After this, return to the main menu by pressing “0”.
- Make sure “Internal EMAC” is selected as boot device by pressing “2” (Select Boot Device) and pressing “1”.
- Optional: save the configuration by pressing “7”.
- Start network boot by pressing “0”.
- In Platform Builder, select Project -> XXX Properties and navigate to the Configuration Properties -> General section. Select “xldremmc.bin” from the “Target file name for debugger:” –dropdown menu. Confirm the setting and close the properties dialog.
- Select TARGET -> Attach Device.
- Wait for your device to show up under “Active target devices” (e.g. “AM335X-21879”). The target device string must match the EBOOT console output (e.g. “INFO: *** Device Name AM335X-21879 ***”)
- Select the device and press Apply.
- The X-Loader should be downloaded and flashed automatically. Confirm this by examining the serial console output for “INFO: XLDR/EBOOT/IPL downloaded, spin forever”.

3.2.3. Boot from external card and download eMMC EBOOT to the board

- Reboot the board
- Follow the same steps as under 3.2.2., but select “ebootemmc.bin” instead of “xldremmc.bin” as target file name. To do this, first select TARGET -> Detach Device from the PlatformBuilder menu, select the file in the Project Properties dialog and select TARGET -> Attach Device after this. If you previously saved the network settings using the Eboot menu, it is not necessary to re-configure it again.

3.2.4. Boot from internal eMMC

- Change the configuration of dip switch S2 for boot from MMC1 / eMMC first (S2-[5:1] = 11100). See hardware documentation for details.
- Remove the external SD Card from X10.
- Power up the board. The X-Loader and EBOOT should boot up from eMMC.
- Enter the EBOOT menu by pressing “Space”.
- Enter the “eMMC Management” submenu by pressing “5”.
- Format the eMMC disk by pressing “7” and confirm. Wait for the operations to complete. Note: this may take several minutes. Please be patient.
- Start Ethernet Auto Download by exiting the Eboot menu (press “0” several times).
- In PlatformBuilder, select the “NK.bin” file from the Target File Name dropdown list.

- Select TARGET -> Attach Device. The NK Image should be downloaded and executed on the device (RAM-Image).

3.2.5. Write NK Image to eMMC flash (optional)

- Select “write image to eMMC flash” in the Catalog under “Configuration” and rebuild.
- Download the created NK.bin to the device. The image should be written to eMMC and boot automatically (this can take some time).
- Reboot the board, enter the Eboot menu and select “2” (Select Boot Device). Press “2” to select “NK from eMMC”. Exit to the main menu by pressing “0”.
- Optional: save the configuration by pressing “7.
- Exit Eboot menu by pressing “0”. The NK Image should be loaded and started from eMMC.

3.2.6. Write Boot Logo to eMMC flash (optional)

Follow the same steps as under 3.2.2. but download the file “logo.nb0” to the device. In order to select this file for download, go to Configuration Properties -> General section. Type “logo.nb0” into the “`Target file name for debugger`:” –box. The boot logo should be downloaded and flashed automatically by the Eboot. The logo should be displayed on the display at the next boot.

4. Supported Drivers

The BSP supports the following drivers/devices:

4.1. Ethernet

The Ethernet driver supports the AM335x CPSW3G Ethernet Controller with one or two external ports in 10/100 MBit and GBit mode.

The single port configuration can be selected in the BSP-specific Catalog under drivers -> Ethernet -> 10/100/1000 Ethernet Support (Port 1). Port 1 is mapped to X11 on the baseboard.
The dual port configuration can be selected in the BSP-specific Catalog under Drivers -> Ethernet -> 10/100/1000 Ethernet Support (Port 1 & 2). Port 1 is mapped to X11 and Port2 is mapped to X12.

The Ethernet driver Registry is available under `WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\EMAC\cpgmacMiniport.reg`.

4.2. USB Host

The USB Host driver supports the AM335x USB0 Controller in host mode.
The USB Host driver can be selected in the BSP-specific Catalog under Drivers -> USB -> USBH driver.
The Host is connected to the USB2517 HUB on the MBa335x Baseboard. Three of the HUB-Ports are readily available as USB A-type connectors on X7 and X8.

The USB Host driver Registry is available under `WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\USB\usbhost.reg`.

4.3. SD Host

The SD Host driver supports the AM335x MMC0 controller in SD Memory mode.
The SD Host driver can be selected in the BSP-specific Catalog under Drivers -> Storage Devices -> SD Host Controller.
The driver supports SD, as well as SDHC memory cards. The memory card is accessible via `\Storage_Card` in the device filesystem.

Note: because of the standard hardware configuration of the MBa335x baseboard, there is no dynamic card detection available. Therefore, the SD Card has to be inserted at boot time in order to detect it.

The SD Host driver Registry is available under
`WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\SDHC\sdhc.reg`.

4.4. eMMC

The eMMC driver supports the AM335x MMC1 controller with the onboard 4GB eMMC memory in MMC memory mode.
The eMMC driver can be selected in the BSP-specific Catalog under Drivers -> Storage Devices -> SD Host Controller (internal eMMC).
The memory card is accessible via `\MMC` in the device filesystem (Note: the filesystem is automatically created after formatting/partitioning the eMMC from the bootloader menu).

The eMMC driver registry is available under
`WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\SDHC\sdhc.reg`.

4.5. UARTs

The UART driver supports the AM335x UART0 and UART3 devices. Because of the standard hardware configuration of the MBa335x board, UART0 and UART3 support RX/TX only. There are no flow control signals available.
The driver for UART0 can be selected in the BSP-specific Catalog under Drivers -> UART -> UART0 driver.
The driver for UART3 can be selected in the BSP-specific Catalog under Drivers -> UART -> UART3 driver.
The UART0 is accessible as the COM1: device, and UART3 as the COM4: device using the Win32 Serial Port API.

The UART Host driver Registry is available under
`\WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\UART\uart.reg`.

Note: although UART0 RX and TX are accessible through MBa335x Pins 41 and 42 on X5, additional RS485 logic on the board prevents the RX pin from functioning properly. In order to use UART0 over X5 in TTL mode, it is necessary to remove R1911 on the baseboard.

UART0 also supports RS485 mode. It can be enabled by selecting Drivers -> UART -> UART0 driver -> UART0 RS485 driver in the BSP-specific Catalog. RS485 works in fullduplex mode with the TX driver enabled permanently.

4.6. EEPROMs

The EEPROM driver supports the SE97B EEPROMs on the TQMa335x Module and MBa335x Baseboard with 256 Bytes capacity each.
The EEPROM drivers can be selected in the BSP-Specific Catalog under Drivers -> EEPROM -> SE97B EEPROM driver (module) and Drivers -> EEPROM -> SE97B EEPROM driver (baseboard).
The module EEPROM is accessible as the EEP1: device and the baseboard EEPROM is accessible as the EEP2: device using the file API (CreateFile(), ReadFile(), WriteFile(), SetFilePointer() etc.).

4.7. Temperature Sensors

The OAL supports the SE97B temperature sensors on the TQMa335x Module and MBa335x Baseboard. The current temperature can be read out using the following OAL IOControl call:

```
KernelIoControl( IOCTL_HAL_GET_TEMP, &index, sizeof(index), &temp,
sizeof(temp), NULL );
```

Where index is a DWORD in the range of 1-2 and temp is a DWORD receiving the returned temperature.
An index of 1 reads out the temperature of the module sensor, and an index of 2 reads out the temperature of the baseboard sensor. The temperature is returned in the format temperature in °C * 1000.

4.8. RTC

The OAL supports the onboard PMIC RTC. The RTC is used automatically by the system (OAL). Reading out and setting the time information is therefore done e.g. by the `GetSystemTime()` and `SetSystemTime()` functions or by the `date` and `time` console functions. The RTC is synchronized on system startup (read) and when changed to a new value (write).

Note: please inspect the hardware documentation for information of how to use the backup battery to save the system time between boots (jumpers on X3).

4.9. I2C

The (usermode accessible) I2C proxy driver supports the AM335x I2C0 and I2C1 interfaces.

The driver can be selected in the BSP-specific Catalog under Drivers -> I2C -> I2C driver (I2C0) and Drivers -> I2C -> I2C driver (I2C1).

The I2C proxy driver registry is available under `\WINCE800\platform\AM335x_TQS\Prebuilt\Registry\APP\I2CPROXY\i2cproxy.reg`.

The I2C0 interface is available as the I2C1: device and the I2C1 interface is available as the I2C2: device.

A simple test application with example source code is available under \WINCE800\PLATFORM\AM335x_TQS\SRC\test\testI2c.

The I2C proxy driver is accessible using the file API (CreateFile(), ReadFile(), WriteFile(), SetFilePointer()).
SetFilePointer() is used to select the base subadress which is accessed in subsequent ReadFile() or WriteFile() calls. To select the I2C device address and speed, the following IOControl Codes are available (defined in i2cproxy.h):

IOCTL_I2C_SET_SLAVE_ADDRESS
Sets the slave address of the I2C device to be accessed.
Parameter lpInBuffer: pointer to a DWORD containing the slave address.

IOCTL_I2C_SET_SUBADDRESS_MODE
Sets the subaddress mode.

Parameter lpInBuffer: pointer to a DWORD containing the desired subaddress mode. The available modes are (defined in sdk_i2c.h):

I2C_SUBADDRESS_MODE_0 : no device subaddresses

I2C_SUBADDRESS_MODE_8 : 1 Byte subadresses

I2C_SUBADDRESS_MODE_16 : 2 Byte subadresses

I2C_SUBADDRESS_MODE_24 : 3 Byte subadresses

I2C_SUBADDRESS_MODE_32 : 4 Byte subadresses

IOCTL_I2C_SET_BAUD_INDEX
Sets the I2C speed (baudrate).

Parameter lpInBuffer: pointer to a DWORD containing the desired baud index. The available baudrates are (defined in sdk_i2c.h):

SLOWSPEED_MODE : 100 KHz

FULLSPEED_MODE : 400 KHz

HIGHSPEED_MODE_1P16 : 1.6 MHz

HIGHSPEED_MODE_2P4 : 2.4 MHz

HIGHSPEED_MODE_3P2 : 3.2 MHz

4.10. SPI

The SPI driver supports the AM335x MCSPI0 interface. The driver can be selected in the BSP-specific catalog under Drivers -> MCSPI -> SPI driver (MCSPI0). The SPI driver registry is available under `\WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\MCSPI\mcspi.reg`. The SPI0 interface is available as the `SPI1:` device.

A simple test application with example source code is available under `\WINCE800\PLATFORM\AM335x_TQS\SRC\test\testSpi.`
The driver supports the following functions (defined in `sdk_spi.h`):

`HANDLE SPIOpen(LPCTSTR pSpiName)`
Opens the driver for subsequent use.
Parameter pSpiName: String containing the device name (“SPI1”).

Return value: Handle to the driver.

`VOID SPIClose(HANDLE hContext)`
Closes the driver after use.
Parameter hContext: Handle returned by SPIOpen().

`BOOL SPILockController(HANDLE hContext, DWORD dwTimeout)`
Locks the access to the driver to the current thread.
Parameter hContext: Handle returned by SPIOpen().
Parameter dwTimeout: Timeout for acquiring the lock.

Return value: TRUE if success, FALSE if failure.

`BOOL SPIUnLockController(HANDLE hContext)`
Unlocks the access to the driver.
Parameter hContext: Handle returned by SPIOpen().

Return value: TRUE if success, FALSE if failure.

`BOOL SPIConfigure(HANDLE hContext, DWORD address, DWORD config)`
Configures the SPI device for subsequent actions.
Parameter hContext: Handle returned by `SPIOpen()`.
Parameter address: Chipselect number (only CS0 is supported).
Parameter config: DWORD containing the desired configuration. The configuration has to be set corresponding to the `MCSPI_CH0CONF` register description in the AM335x Technical Reference Manual.

Return value: TRUE if success, FALSE if failure.

BOOL SPIEnableChannel(HANDLE hContext)

Enables the channel configured by the SPIConfigure() address parameter and therefore activates the corresponding chipselect.

Parameter hContext: Handle returned by SPIOpen().

Return value: TRUE if success, FALSE if failure.

BOOL SPIDisableChannel(HANDLE hContext)

Disables the channel previously enabled by SPIEnableChannel().

Parameter hContext: Handle returned by SPIOpen().

Return value: TRUE if success, FALSE if failure.

BOOL SPISetSlaveMode(HANDLE hContext)

Configures the SPI controller for slave mode.

Parameter hContext: Handle returned by SPIOpen().

Return value: TRUE if success, FALSE if failure.

DWORD SPIRead(HANDLE hContext, DWORD size, VOID *pBuffer)

Reads from the SPI bus.

Parameter hContext: Handle returned by SPIOpen().

Parameter size: Number of Bytes to read.

Parameter pBuffer: Pointer to the receivebuffer.

Return value: Number of Bytes actually read.

DWORD SPIWrite(HANDLE hContext, DWORD size, VOID *pBuffer)

Writes to the SPI bus.

Parameter hContext: Handle returned by SPIOpen().

Parameter size: Number of Bytes to write.

Parameter pBuffer: Pointer to the sendbuffer.

Return value: Number of Bytes actually written.

DWORD SPIWriteRead(HANDLE hContext, DWORD size, VOID *pOutBuffer, VOID *pInBuffer)

Reads and writes from/to the SPI bus simultaneously.

Parameter hContext: Handle returned by SPIOpen().

Parameter size: Number of Bytes to read/write.

Parameter pOutBuffer: Pointer to the sendbuffer.

Parameter pInBuffer: Pointer to the receivebuffer.

Return value: Number of Bytes actually read/written.

DWORD SPIAsyncWriteRead(HANDLE hContext, DWORD size, VOID *pOutBuffer, VOID *pInBuffer)

Reads and writes from/to the SPI bus simultaneously, using DMA.

Parameter hContext: Handle returned by SPIOpen().
Parameter size: Number of Bytes to read/write.
Parameter pOutBuffer: Pointer to the sendbuffer.
Parameter pInBuffer: Unused, set to NULL.

Return value: The value of the size parameter.

DWORD SPIWaitForAsyncWriteReadComplet(HANDLE hContext, DWORD size, VOID *pOutBuffer)

Waits for the DMA transfer to be completed.

Parameter hContext: Handle returned by SPIOpen().
Parameter size: Number of Bytes to write.
Parameter pOutBuffer: Pointer to the receivebuffer.

Return value: The value of the size parameter.

4.11. GPIO

The GPIO driver supports the AM335x GPIOs. The driver can be selected in the BSP-specific Catalog under Drivers -> GPIO -> Gpio driver. The GPIO driver Registry is available under `\WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\GPIO\gpio.reg`. The Gpio driver is available as the GIO1: device.

The GPIOs have to be identified by their GPIO ID. The BSP supports the following AM335x Gpios:

AM335x GPIO	GPIO ID
GPIO1_28	60
GPIO1_29	61
GPIO2_0	64
GPIO0_18	18

Additionally, the driver supports the GPIOs of the two GPIO expanders on the baseboard:

D900 GPIO	GPIO ID
IO0	128
IO1	129
IO2	130
IO3	131
IO4	132
IO5	133
IO6	134
IO7	135

D901 GPIO	GPIO ID
IO0	136
IO1	137
IO2	138
IO3	139
IO4	140
IO5	141
IO6	142
IO7	143

A simple test application with example source code is available under
\WINCE800\PLATFORM\AM335x_TQS\SRC\test\testGPIO.
The driver supports the following IOControl Codes (defined in gpio_ioctls.h):

IOCTL_GPIO_SETBIT
Sets the corresponding GPIO to level 1.
Parameter lpInBuffer: pointer to a DWORD containing the GPIO ID to set.

IOCTL_GPIO_CLRBIT
Sets the corresponding GPIO to level 0.
Parameter lpInBuffer: pointer to a DWORD containing the GPIO ID to set.

IOCTL_GPIO_GETBIT
Reads out the level of the corresponding GPIO.
Parameter lpInBuffer: pointer to a DWORD containing the GPIO ID to read out.
Parameter lpOutBuffer: pointer to a DWORD receiving the current level.

IOCTL_GPIO_SETMODE
Configures the mode of the corresponding GPIO.
Parameter lpInBuffer: pointer to an array of two DWORDs containing the GPIO ID (array element 0) and the mode (array element 1) to set.

The following modes are supported (defined in gpio_defines.h):
GPIO_DIR_OUTPUT : configures the GPIO to output
GPIO_DIR_INPUT : configures the GPIO to input
GPIO_INT_LOW_HIGH : enables rising edge interrupt
GPIO_INT_HIGH_LOW : enables falling edge interrupt
GPIO_INT_LOW : enables low level interrupt
GPIO_INT_HIGH : enables high level interrupt
GPIO_DEBOUNCE_ENABLE : enables debouncing

IOCTL_GPIO_GETMODE
Returns the current mode of the corresponding GPIO.
Parameter lpInBuffer: pointer to a DWORD containing the GPIO ID.
Parameter lpOutBuffer: pointer to a DWORD receiving the mode.

IOCTL_GPIO_GETIRQ
Returns the IRQ of the corresponding GPIO.
Parameter lpInBuffer: pointer to a DWORD containing the GPIO ID.
Parameter lpOutBuffer: pointer to a DWORD receiving the IRQ number.

IOCTL_GPIO_SET_DEBOUNCE_TIME

Sets the debounce time of the GPIO (bank).

Parameter lpInBuffer: pointer to an IOCTL_GPIO_SET_DEBOUNCE_TIME_IN structure (defined in gpio_ioctls.h) containing the GPIO ID and the debounce time to set:

```
typedef struct    {
    UINT                gpioId;
    UINT                debounceTime;
                    } IOCTL_GPIO_SET_DEBOUNCE_TIME_IN;
```

The debouncing time can be calculated as follows:
Debouncing time *= (DEBOUNCETIME + 1) × 31* µs. The debouncing time is global to all GPIOs corresponding to the same bank.

IOCTL_GPIO_GET_DEBOUNCE_TIME

Returns the debounce time for the GPIO (bank).

Parameter lpInBuffer: pointer to a DWORD containing the GPIO ID.

Parameter lpOutBuffer: pointer to a DWORD receiving the debounce time.

IOCTL_GPIO_INIT_INTERRUPT

Initializes the interrupt for the GPIO.

Parameter lpInBuffer: pointer to an IOCTL_GPIO_INIT_INTERRUPT_INFO structure (defined in gpio_ioctls.h):

```
typedef struct    {
    UINT    uGpioID;
    DWORD   dwSysIntrID;
    HANDLE  hEvent;
} IOCTL_GPIO_INIT_INTERRUPT_INFO,
*PIOCTL_GPIO_INIT_INTERRUPT_INFO;
```

uGpioID has to be set to the GPIO ID and hEvent must be set to an Event Handle. The used SysIntr is returned in the dwSysIntrID Element.

IOCTL_GPIO_ACK_INTERRUPT

Acknowledges a GPIO interrupt.

Parameter lpInBuffer: pointer to a IOCTL_GPIO_INTERRUPT_INFO structure (defined in gpio_ioctls.h):

```
typedef struct {
      UINT  uGpioID;
      DWORD dwSysIntrID;
} IOCTL_GPIO_INTERRUPT_INFO,
*PIOCTL_GPIO_INTERRUPT_INFO;
```

IOCTL_GPIO_DISABLE_INTERRUPT
Disables the interrupt of a GPIO.
Parameter lpInBuffer: pointer to a IOCTL_GPIO_INTERRUPT_INFO structure.

4.12. CAN

The CAN driver supports the AM335x DCAN0 and DCAN1 devices. The driver for DCAN0 can be selected in the BSP-specific Catalog under Drivers -> CAN -> CAN0 driver. The driver for DCAN1 can be selected in the BSP-specific Catalog under Drivers -> CAN -> CAN1 driver.

The DCAN0 is accessible as the CAN1: device, and DCAN1 as the CAN2: device.

The DCAN driver Registry is available under `\WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\DCAN\dcan.reg`.

A basic test application with example source code is available under `\WINCE800\PLATFORM\AM335x_TQS\SRC\test\testCAN`.
The driver supports the following IOControl Codes (defined in `sdk_can.h`):

IOCTL_CAN_COMMAND
Starts, Stops or Resets the DCAN controller.
Parameter lpInBuffer: pointer to an `IOCTL_CAN_COMMAND_IN` (`CAN_COMMAND`) enum (defined in sdk_can.h):

```
typedef enum {
            STOP,
            START,
            RESET
            } CAN_COMMAND;
```

IOCTL_CAN_STATUS
Retrieves the status of the DCAN controller.
Parameter lpOutBuffer: pointer to an `IOCTL_CAN_STATUS_OUT` (`CAN_STATUS`) struct (defined in sdk_can.h):

```
typedef struct {
LONG currentRxMsg;                    //current number of rx messages in
                                      the //driver buffer.
DWORD maxRxMsg;                       //max number of rx messages in the
                                      //driver buffer.
LONG currentTxMsg[NB_TX_PRIORITIES];  //current number of tx messages per
                                      //priority.
DWORD maxTxMsg[NB_TX_PRIORITIES];     //max number of tx messages in the
                                      //driver buffer per priority.
BUS_STATE BusState;
```

```
CTRL_STATE CtrlState;
LONG RxDiscarded;                    //number of discarded messages
LONG RxLost;                         //number of lost rx messages
LONG FilteredOut;                    //number of filtered out messages
LONG TotalReceived;                  //total number of received messages
LONG TotalSent;                      //total number of sent messages
UINT32 CANTEC;                       //TX error counter.
UINT32 CANREC;                       //RX error counter.
} CAN_STATUS;
```

The BUS_STATE member is defined as follows:

```
typedef enum BUS_STATE {
ERROR_ACTIVE,
ERROR_ACTIVE_WARNED,
ERROR_PASSIVE,
BUS_OFF,
} BUS_STATE;
```

The CTRL_STATE member is defined as follows:

```
typedef enum{
STOPPED,
STARTED,
} CTRL_STATE;
```

IOCTL_CAN_CONFIG

Configures the DCAN controller baudrate.

Parameter lpInBuffer: pointer to an IOCTL_CAN_CONFIG_IN structure (defined in sdk_can.h):

```
typedef struct {
    CONFIG_TYPE cfgType;
    union {
        DWORD BaudRate;
    };
} IOCTL_CAN_CONFIG_IN;
```

The only defined value for the cfgType member is BAUDRATE_CFG. The BaudRate member of the structure has to be set to the desired baudrate.

IOCTL_CAN_FILTER_CONFIG

Configures a DCAN controller receive filter.

Parameter lpInBuffer: pointer to an IOCTL_CAN_CLASS_FILTER_CONFIG_IN structure (defined in sdk_can.h):

```
typedef struct {
    FILTER_CONFIG_TYPE    cfgType;
    CLASS_FILTER          classFilter;
    RXCAN_PRIORITY        rxPriority;
    BOOL fEnabled;
} IOCTL_CAN_CLASS_FILTER_CONFIG_IN;
```

The FILTER_CONFIG_TYPE can have the following values:

```
typedef enum {
    CREATE_CLASS_FILTER_CFG,           //create a (hardware) filter
    ENABLE_DISABLE_CLASS_FILTER_CFG,   //enable/disable a (hardware) filter
    DELETE_CLASS_FILTER_CFG,           //delete a (hardware) filter
    ADD_SUBCLASS_FILTER_CFG,           //add a subclass (software) filter
    REMOVE_SUBCLASS_FILTER_CFG,        //remove a subclass (software) filter
} FILTER_CONFIG_TYPE;
```

The CLASS_FILTER member is of the following type, defining the can id and mask for the receive filter:

```
typedef struct{
    CAN_ID id;
    CAN_ID mask;
} CLASS_FILTER;
```

The RXCAN_PRIORITY can have the following values:

```
typedef enum {
    RXCRITICAL = 0,
    RXMEDIUM = 1,
    RXLOW = 2,
    NB_RX_PRIORITIES
} RXCAN_PRIORITY;
```

IOCTL_CAN_SEND

Sends one or multiple messages over the canbus.

Parameter lpInBuffer: pointer to an IOCTL_CAN_SEND_IN structure (defined in sdk_can.h):

```
typedef struct {
    DWORD nbMsg;                //number of messages to send
    DWORD nbMsgSent;
    DWORD timeout;              //timeout for the send operation
    TXCAN_PRIORITY priority;
    CAN_MESSAGE *msgArray;
} IOCTL_CAN_SEND_IN;
```

Priority is an enum with the following defined values:

```
typedef enum {
    TXCRITICAL = 0,
    TXMEDIUM = 1,
    TXLOW = 2,
    NB_TX_PRIORITIES
} TXCAN_PRIORITY;
```

msgArray is an Array of CAN_MESSAGE structures:

```
typedef struct {
UINT32 MDL;         //lower 4 Bytes of the can message
UINT32 MDH;         //higher 4 Bytes of the can message
BYTE length;        //message data length
CAN_ID id;          //can id of the message
}CAN_MESSAGE;
```

The id member represents a standard or extended can ID:

```
typedef union{
    struct {
        unsigned int id:29;
        unsigned int reserved:2;
        unsigned int extended:1;
    } s_extended;
    struct {
        unsigned int reserved0:18;
        unsigned int id:11;
        unsigned int reserved1:2;
        unsigned int extended:1;
    } s_standard;
    UINT32 u32;
} CAN_ID;
```

IOCTL_CAN_REMOTE_CONFIGURE_AUTO_ANSWER

Configures (sets or deletes) an auto-answer message.

Parameter lpInBuffer: pointer to an `IOCTL_CAN_REMOTE_CONFIGURE_AUTO_ANSWER_IN` structure (defined in sdk_can.h):

```
typedef struct {
    AUTO_ANSWER_CONFIG_TYPE     cfgType;
    CAN_MESSAGE                 msg;              //auto answer message
} IOCTL_CAN_REMOTE_CONFIGURE_AUTO_ANSWER_IN;
```

The AUTO_ANSWER_CONFIG_TYPE member can have the following values:

```
typedef enum {
    SET_AUTO_ANSWER,                  //enable the auto-answer message
    DELETE_AUTO_ANSWER                //delete the auto-answer message
}AUTO_ANSWER_CONFIG_TYPE;
```

IOCTL_CAN_REMOTE_REQUEST

Sends a remote request message. The remote answer (if any) is received through the normal receive buffer (if configured).

Parameter lpInBuffer: pointer to an `IOCTL_CAN_REMOTE_REQUEST_IN (CAN_REMOTE_REQUEST)` structure (defined in sdk_can.h):

```
typedef struct {
    CAN_ID id;            //can id of the remote request
} CAN_REMOTE_REQUEST;
```

IOCTL_CAN_RECEIVE

Reads out a number of received messages from the driver.

Parameter lpOutBuffer: pointer to an IOCTL_CAN_RECEIVE_OUT structure (defined in sdk_can.h):

```
typedef struct {
    DWORD nbMaxMsg;             //maximum number of messages to receive
    DWORD nbMsgReceived;        //actual number of returned messages
    DWORD timeout;              //timeout to wait for message receive
    CAN_MESSAGE *msgArray;      //array of CAN_MESSAGE structures
} IOCTL_CAN_RECEIVE_OUT;
```

4.13. Display

The display driver supports the ET0700 display on the parallel interface. The display driver can be selected in the BSP-Specific Catalog under Drivers -> Display -> LCDC Display driver and Drivers -> Display -> Panel -> ET0700 7'' display. The display has to be also selected in the Eboot menu under [9] Select Display Resolution -> [1] 7in EMERGING ETM07 (800x480@60Hz).

4.14. Touch

The touch driver supports the AM335x's analog touchscreen controller. The touchscreen driver can be selected in the BSP-Specific Catalog under Drivers -> Display -> Touch -> AM335x Resistive Touch.

4.15. USB Device

The USB Device driver supports the AM335x USB1 controller in device mode. The USB OTG driver can be selected in the BSP-specific Catalog under Drivers -> USB -> mUSB OTG driver. The USB Device connector is located at X9.

The USB Device driver registry is available under WINCE800\platform\AM335x_TQS\Prebuilt\Registry\DRIVERS\USB\usbotg.reg.

The USB Device driver is configured to USB serial mode by default. This can be changed by altering the environment variable "BSP_DEFAULT_USB_CLIENT". It is set to "USBSER_CLASS" in WINCE800\platform\AM335x_TQS\AM335x_TQS.bat by default.