

# Turbo BDM Light ColdFire interface

(c) 2006, Daniel Malík

rev 1.4

I am grateful to several people at Freescale Semiconductor. Creation of this interface would never have been possible without their kind help and guidance.

## 1.0 Introduction to TBLCF

### 1.1 Purpose of this document

This document describes the Turbo BDM Light ColdFire (TBLCF) interface and associated SW libraries and tools. TBLCF is a hardware interface which connects between a USB equipped computer and the BDM debugging port of Freescale Semiconductor ColdFire microprocessors/microcontrollers. It enables debuggers and other SW tools to communicate with the microcontroller, download code into its on-chip flash, etc.

### 1.2 Aspirations and roots of TBLCF

TBLCF has its roots in my previous TBDML project. There are many similarities between the two projects. Firmware for TBLCF is based on the firmware I have developed for the TBDML.

TBCLF was developed with the following requirements in mind:

- very low cost (sub \$10)
- ease of assembly and prototyping (widely available components)
- open-end SW interface with documented API for easy integration into debuggers and new standalone tools
- easy SW migration under Linux
- support for at least one widely used debugger
- modern and widely available interface for communicating with the computer (USB)

Unfortunately to achieve the performance required for this project I had to use the 68HC908JB16 MCU which does not come in a DIL package. Since the microcontroller is in SMD package, I have used majority of the remaining components in SMD packages as well. However the choice of packages is not important for functionality and can be completely arbitrary.

I am making all the SW I can open source. However some of the SW components are distributed as binaries only due to licensing restrictions. If you would like to receive source code to these components please obtain a permission from Freescale Semiconductor prior to sending me your request.

## 2.0 Description of TBLCF

### 2.1 What you get

The TBLCF package consists of

- complete HW description which enables you to build the interface
- firmware for the interface, USB drivers and DLL interface library (TBLCF.DLL) for Windows
- bootloader tool (TBLCF\_BT) tool for programming firmware into the cable after it is first built and for upgrades in case of bugs/enhancements
- unsecure tool (TBLCF\_UNSEC) for mass erasing of on-chip flash of secured devices
- interface for the CodeWarrior debugger

### 2.2 Hardware

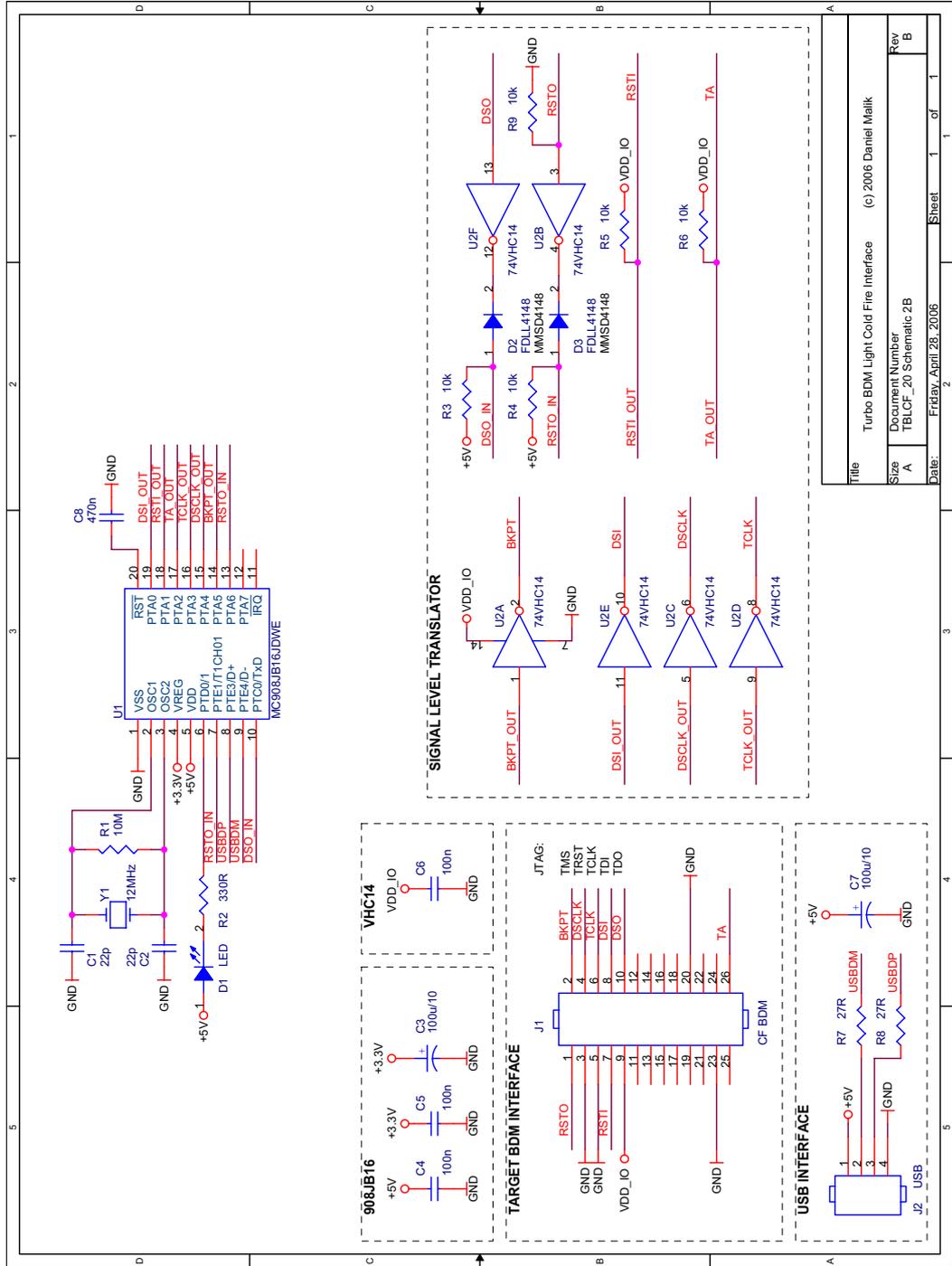
TBLCF uses USB as the means of talking to the computer. Here is why:

- I like the concept of USB.
- USB provides power to the interface; no bulky wall adapters and no inefficient regulators with hot heatsinks are needed.

The TBLCF is based on MC68HC908JB16 MCU from Freescale Semiconductor. My reasons for selecting this MCU are:

- TBLCF was based on MC68HC908JB8 and the JB16 variant is compatible with it to a large degree.
- The JB16 variant has a pre-programmed USB bootloader and therefore no special hardware tools are needed to program the firmware into the on-chip flash.

Schematic diagram of the TBLCF interface is shown in figure 1. The interface has two main parts: the MC68HC908JB16 MCU itself and the BDM interface driver based on 74VHC14 buffer with schmitt trigger inputs.



Title		Turbo BDM Light Cold Fire Interface		(c) 2006 Daniel Malik	
Size	A	Document Number	TBLCF_20 Schematic 2B	Rev	B
Date:	Friday, April 28, 2006	Sheet	1	of	1

**FIGURE 1. TBLCF schematic**

## 2.2.1 Remarks on the BDM interface driver

I have used the 74VHC14 to achieve low-cost translation of BDM signals with voltages anywhere between 3.3V and 5V to the 5V logic of the MCU. The VHC logic accepts overvoltage on inputs, however the output voltage swing is limited by the power rail voltages. When the 74VHC14 is powered by 3.3V source resistors R3 and R4 would not be able to pull the signals above the 3.3V rail and would only inject current into the power rail of the 74VHC14. 3.3V is below the minimum high level input voltage of the MC68HC908JB16 and the circuit would not be guaranteed to work. I have therefore added diodes D2 and D3 to increase the high level voltages. A better alternative would be to use two N-mosfet transistors, but that would increase the cost and complicate the PCB layout (which was my main reason for not using them).

You will also notice that the RSTO\_IN signal is brought to two different pins of the MCU. This is strictly speaking not needed and a connection to pin PTE1 would be sufficient. However connecting the signal to PTA6 as well as PTE1 simplified my PCB design.

## 2.2.2 ColdFire BDM connector

The ColdFire BDM connector has been here for a long time. In the past boards usually contained a lot of components and were fairly large. A 26-way connector with 0.1" spacing was therefore of a reasonable size. Size of boards is however shrinking and the connector is becoming too large for smaller applications. I have made two optional enhancements to the BDM connector:

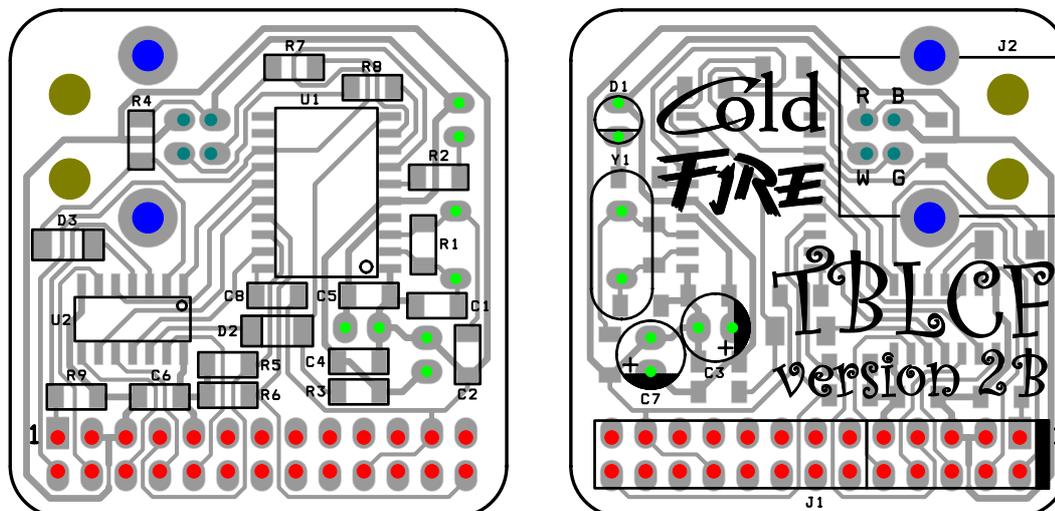
1. Where the 26-way connector is too large you can use a 10-way subset of the connector (pins 1 through 10). The only signal which is missing the  $\overline{TA}$  on pin 26, but this is only needed in systems with external memory bus where the debugger is configured incorrectly and accesses an area for which a Transfer Acknowledge is not generated (neither internally nor externally). I.e. the probability that it will be needed is quite low and the absence of the signal can be compensated for by a careful use of the debugger. See figure 4 for a photograph of the interface with a 10-way ribbon cable attached.

2. I have added the  $\overline{RSTO}$  signal to pin 1 of the connector which was so far unused. This enables the interface to detect resets of the microcontroller caused by for example the COP/watchdog circuit or a user RESET button.

Note that the above enhancements are suggestions only and the interface will operate even with the original 26-way connector. Pins 11 and 12 of the 26-way connector can be removed to make the interface compatible with both the 10-way and 26-way ribbon cables.

### 2.2.3 Printed Circuit Board

The PCB I have designed for TBLCF is shown in figure 2.



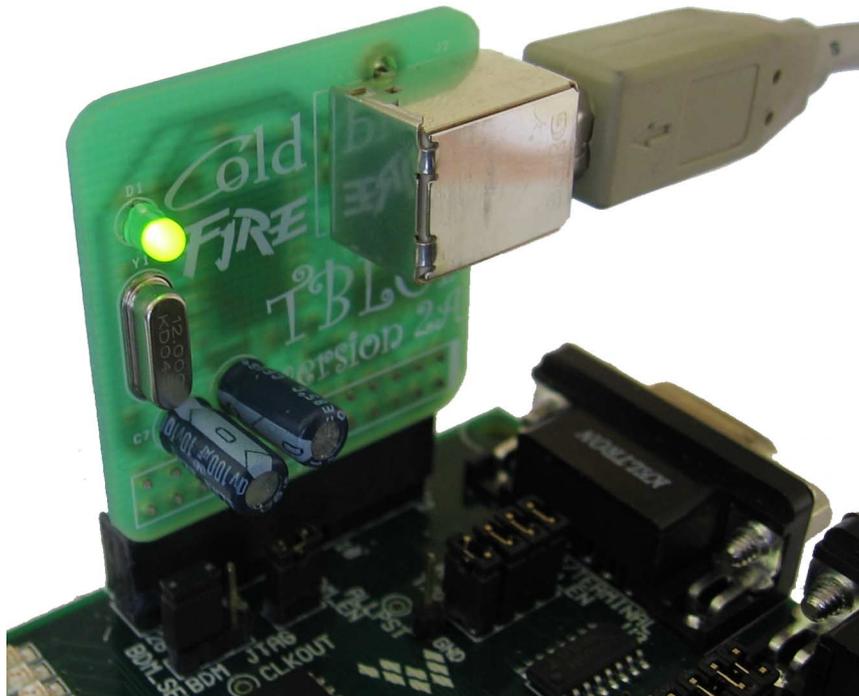
**FIGURE 2. TBLCF PCB**

The PCB is designed to be single sided only (with no wire links!). The design rules have been set to 12 mil spacing and 14 mil minimum copper width. The dimensions of the PCB are 37.08 x 38.1 mm. The PCB should be fairly inexpensive to produce (it is costing me around \$2 a piece in small quantity with solder mask and both silk screens).

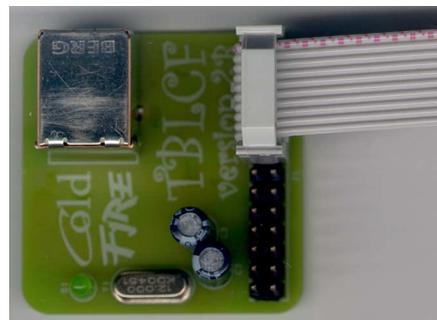
You can use my gerber files to have the PCB made professionally. Alternatively you can make the PCB in your garage or garden shed, design your own PCB or you can also populate the interface on a piece of prototyping board.

Note that the PCB has been provided with footprint for the USB B connector. This is actually a violation of the USB specification as low-speed devices should have the cable hard-wired to them, but I have found a detachable cable very useful. If you do not feel like violating the specification, you can solder the cable straight into the PCB. The PCB is marked with wire colours and two extra holes are provided for strapping the cable to the PCB in case you wish to do this.

Connector J1 can be populated with either a 26-way header (for use with a 26-way or a 10-way flat cable) or with a 26-way/10-way plug. Note that a plug needs to be populated from the bottom side to keep the pin assignment correct! I have populated my first prototype with a 26-way 90 degree plug as shown in figure 3.



**FIGURE 3. TBLCF plugged into MCF52235 evaluation board**



**FIGURE 4. Bottom and top view of TBLCF with a 10-way ribbon cable**

#### **2.2.4 Getting the components**

To make life slightly easier for you I have listed order numbers of the components needed to build the interface in the table below. However please note that you will probably be able to get the components at a much lower price at your local high street shop. You might also want to try to get some of the components for free as samples (Freescale Semiconductor and ON Semiconductor offer free samples of the parts I am using in the design and this is how I am getting parts myself).

Item	Quantity	Reference	Value	Farnel #	DigiKey #
1	2	C1,C2	22p	3606041	311-1154-1-ND
2	2	C3,C7	100u/10	9451080	493-1040-ND
3	3	C4,C5,C6	100n	644316	490-1825-1-ND
4	1	C8	470n	757688	PCC1901CT-ND
5	1	D1	LED	1142517	SSL-LX3044LGD
6	2	D2,D3	FDLL4148	9843710	FDLL4148CT-ND
7	1	J1	CF BDM	3167070	HRP26H-ND
8	1	J2	USB	1097897	609-1039-ND
9	1	R1	10M	9237119	RHM10MECT-ND
10	1	R2	330R	9337326	RHM330FCT-ND
11	5	R3,R4,R5,R6,R9	10k	9337016	RHM10.0KFCT-ND
12	2	R7,R8	27R	9337261	RHM27.0FCT-ND
13	1	U1	MC68HC908JB16JDWE		MC908JB16JDWE-ND
14	1	U2	74VHC14	1014104	74VHC14M-ND
15	1	Y1	12MHz	9712950	535-9037-ND

## 2.3 Software

The basic SW package for the TBLCF interface consists of six different components:

- firmware in the MC68HC908JB16
- interface DLL (TBLCF.DLL)
- USB driver (LIBUSB)
- SW interface for the CodeWarrior debugger
- bootloader utility for re-programming firmware of the cable (TBLCF\_BT)
- unsecure utility (TBLCF\_UNSEC)

All the components are intended to be used as binaries by majority of users. For those who would like to look deeper I am providing source code of the firmware, the interface DLL, unsecure utility and the bootloader application.

The LIBUSB is open source software available under combination of GNU general and lesser general public licenses.

The SW interface for the CodeWarrior debugger was created based on information which is not available in the public domain. The license attached to this information is preventing me from disclosing the source code.

### 2.3.1 TBLCF DLL API

Debuggers and other tools should primarily use the TBLCF DLL to interface to the TBLCF tool. This section describes the API the TBLCF DLL v1.0 offers.

#### **unsigned char tblcf\_dll\_version(void)**

Returns version of the DLL in BCD format (major in upper nibble and minor in lower nibble).

### **unsigned char tblcf\_init(void)**

Initialises the USB interface and returns number of TBLCF devices found attached to the computer. This function needs to be called before a device can be opened.

### **unsigned char tblcf\_open(unsigned char device\_no)**

Opens communication with device number *device\_no*. First device has number 0. Returns 0 on success and non-zero on failure. A device must be open before any communication with the device can take place.

### **void tblcf\_close(void)**

Closes communication with the currently opened device.

### **unsigned int tblcf\_get\_version(void)**

Returns version of HW (MSB) and SW (LSB) of the TBLCF interface in BCD format.

### **unsigned char tblcf\_get\_last\_sts(void)**

Returns status of the last executed command: 0 on success and non-zero on failure.

### **unsigned char tblcf\_request\_boot(void)**

Requests execution of the pre-programmed bootloader firmware on next power-up. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_set\_target\_type(target\_type\_e target\_type)**

This function sets target MCU interface type. *target\_type* can be either *CF\_BDM* or *JTAG*. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_target\_reset(target\_mode\_e target\_mode)**

Resets the target MCU to normal or BDM mode. *target\_mode* can be either *BDM\_MODE* or *NORMAL\_MODE*. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_bdm\_sts(bdm\_status\_t \*bdm\_status)**

*bdm\_status* is a pointer to user allocated structure which the function fills with current state of BDM communication. Returns 0 on success and non-zero on failure.

The structure has the following format:

```
typedef struct {  
    reset_state_e reset_state;
```

```
    reset_detection_e reset_detection;  
} bdmcf_status_t;
```

*reset\_state* can be either *RSTO\_ACTIVE* ( $\overline{RSTO}$  input low) or *RSTO\_INACTIVE* ( $\overline{RSTO}$  input high).

*reset\_detection* can be either *RESET\_DETECTED* (active to inactive transition detected on the  $\overline{RSTO}$  input) or *RESET\_NOT\_DETECTED*. *reset\_detection* defaults to *RESET\_NOT\_DETECTED* after each call of the function.

### **unsigned char tblcf\_target\_halt(void)**

Brings the target into BDM mode (i.e. debug mode with user code execution halted). Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_target\_go(void)**

Starts target code execution from current PC address. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_target\_step(void)**

Steps over a single target instruction. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_resynchronize(void)**

Resynchronizes BDM communication with the target. This call is useful when the interface is operating in a very noisy environment and the target picks up an extra pulse on the DSCLK signal. This function resynchronizes the target and the interface and ensures correct exchange of commands without the need for asserting the  $\overline{RSTI}$  signal. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_assert\_ta(unsigned char duration\_10us)**

Asserts the  $\overline{TA}$  signal for the specified duration (in 10us increments). The Transfer Acknowledge signal must be asserted by the interface in case the debugger has attempted to access a location on the external bus which is not configured for auto acknowledgement and there is no external hardware to assert the  $\overline{TA}$  signal. Returns 0 on success and non-zero on failure.

### **unsigned char tblcf\_read\_creg(unsigned int address, unsigned long int \* result)**

Reads control register from the specified address and stores the result in the user supplied variable. byte from memory at the supplied address. Returns 0 on success and non-zero on failure.

**void tblcf\_write\_creg(unsigned int address, unsigned long int value)**

Writes the specified value to the control register at the specified address.

**unsigned char tblcf\_read\_dreg(unsigned char dreg\_index, unsigned long int \* result)**

Reads contents of the specified debug register and stores the result into the user supplied variable. Returns 0 on success and non-zero on failure. Note that current ColdFire devices only support reading of the Configuration/Status Register (CSR).

**void tblcf\_write\_dreg(unsigned char dreg\_index, unsigned long int value)**

Writes the specified value to the specified debug register.

**unsigned char tblcf\_read\_reg(unsigned char reg\_index, unsigned long int \* result)**

Reads the specified data/address register and stores the result into the user supplied variable. Returns 0 on success and non-zero on failure.

**void tblcf\_write\_reg(unsigned char reg\_index, unsigned long int value)**

Writes the specified value to the specified data/address register.

**unsigned char tblcf\_read\_mem8(unsigned long int address, unsigned char \* result)**

Reads an 8-bit value from memory at the specified address. The result is stored into the user supplied variable. Returns 0 on success and non-zero on failure.

**unsigned char tblcf\_read\_mem16(unsigned long int address, unsigned int \* result)**

Reads a 16-bit value from memory at the specified address. The result is stored into the user supplied variable. Returns 0 on success and non-zero on failure. Note that the address will be aligned to a word boundary.

**unsigned char tblcf\_read\_mem32(unsigned long int address, unsigned long int \* result)**

Reads a 32-bit value from memory at the specified address. The result is stored into the user supplied variable. Returns 0 on success and non-zero on failure. Note that the address will be aligned to a long word boundary.

**void tblcf\_write\_mem8(unsigned long int address, unsigned char value)**

Writes the specified 8-bit value at the specified address.

**void tblcf\_write\_mem16(unsigned long int address, unsigned int value)**

Writes the specified 16-bit value at the specified address. Note that the address will be aligned to a word boundary.

**void tblcf\_write\_mem32(unsigned long int address, unsigned long int value)**

Writes the specified 32-bit value at the specified address. Note that the address will be aligned to a long word boundary.

**unsigned char tblcf\_read\_block8(unsigned long int address, unsigned long int bytcount, unsigned char \*buffer)**

Reads a block of 8-bit values from memory from the specified address. The result is stored into the user supplied buffer. Returns 0 on success and non-zero on failure.

**unsigned char tblcf\_read\_block16(unsigned long int address, unsigned long int bytcount, unsigned char \*buffer)**

Reads a block of 16-bit values from memory from the specified address. The result is stored into the user supplied buffer. Returns 0 on success and non-zero on failure. An 8-bit read is performed at the beginning of the block if the address is not aligned to a word boundary. An 8-bit read is performed at the end of the block in case the block does not end at a word boundary.

**unsigned char tblcf\_read\_block32(unsigned long int address, unsigned long int bytcount, unsigned char \*buffer)**

Reads a block of 32-bit values from memory from the specified address. The result is stored into the user supplied buffer. Returns 0 on success and non-zero on failure. 8-bit and/or 16-bit reads are performed at the beginning of the block if the address is not aligned to a long word boundary. 8-bit and/or 16-bit reads are performed at the end of the block in case the block does not end at a long word boundary.

**unsigned char tblcf\_write\_block8(unsigned long int address, unsigned long int bytcount, unsigned char \*buffer)**

Writes a block of 8-bit values to memory from the specified address. Returns 0 on success and non-zero on failure. Always returns 0 in case the firmware was not compiled with the WRITE\_BLOCK\_CHECK symbol defined.

**unsigned char tblcf\_write\_block16(unsigned long int address, unsigned long int bytcount, unsigned char \*buffer)**

Writes a block of 16-bit values to memory from the specified address. Returns 0 on success and non-zero on failure. Always returns 0 in case the firmware was not compiled with the WRITE\_BLOCK\_CHECK symbol defined. An 8-bit write is performed at the beginning of the block if the address is not aligned to a

word boundary. An 8-bit write is performed at the end of the block in case the block does not end at a word boundary.

**unsigned char tblcf\_write\_block16(unsigned long int address, unsigned long int bytcount, unsigned char \*buffer)**

Writes a block of 16-bit values to memory from the specified address. Returns 0 on success and non-zero on failure. Always returns 0 in case the firmware was not compiled with the `WRITE_BLOCK_CHECK` symbol defined. 8-bit and/or 16-bit writes are performed at the beginning of the block if the address is not aligned to a long word boundary. 8-bit and/or 16-bit writes are performed at the end of the block in case the block does not end at a long word boundary.

**unsigned char tblcf\_jtag\_sel\_shift(unsigned char mode)**

Transitions the JTAG state machine from RUN-TEST/IDLE state to SHIFT-DR state (*mode* equal to zero) or SHIFT-IR state (*mode* not equal to zero). This function is to be used in conjunction with `tblcf_jtag_write` and `tblcf_jtag_read` to write instructions and read/write data. Returns 0 on success and non-zero on failure.

**unsigned char tblcf\_jtag\_sel\_reset(void)**

Transitions the JTAG state machine from RUN-TEST/IDLE state to TEST-LOGIC-RESET state. This transition is required in order to execute the `LOCK-OUT RECOVERY` instruction which erases the on-chip flash. Returns 0 on success and non-zero on failure.

**void tblcf\_jtag\_write(unsigned char bit\_count, unsigned char exit, unsigned char \*buffer)**

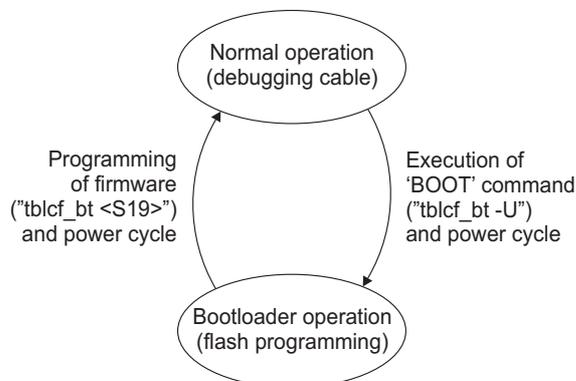
Writes JTAG data. Expects the JTAG state machine to be in the appropriate SHIFT state. Leaves the JTAG state unchanged (*exit* equal to zero) or transitions the state machine to RUN-TEST/IDLE state (*exit* not equal to zero). Data is shifted in starting with LSB of the last byte in the buffer. Parameter *bit\_count* specifies the number of bits to shift in.

**unsigned char tblcf\_jtag\_read(unsigned char bit\_count, unsigned char exit, unsigned char \*buffer)**

Reads JTAG data. Expects the JTAG state machine to be in the appropriate SHIFT state. Leaves the JTAG state unchanged (*exit* equal to zero) or transitions the state machine to RUN-TEST/IDLE state (*exit* not equal to zero). Data shifted out is stored starting with LSB of the last byte in the buffer. Parameter *bit\_count* specifies the number of bits to shift out.

### 3.0 Firmware (re)programming and installation

The interface operates in one of two modes. The bootloader (also called In-Circuit Programming = ICP) mode and the normal operation (debugging cable) mode. The transitions between the two modes are outlined in figure 5.



**FIGURE 5. Transitions between modes of operation**

Installation of the device drivers is a two stage process. After the interface is built it will default into the ICP mode. An ICP driver must be installed to work with the interface and program the on-chip flash with firmware. Once the interface is programmed, another driver must be installed to use it. You can skip the first stage of this process if your device is pre-programmed with the firmware using a different programming method (e.g. a standalone flash programmer) or programmed with firmware already using a different computer.

Installation of the ICP driver and programming of the firmware into the on-chip flash is described in section 3.1. Returning the interface into the ICP mode when firmware is already programmed (needed for firmware upgrade) it is described in section 3.4.

The following sections assume that you have downloaded the zip file with the binary version of the drivers and unzipped it into a suitable directory on your computer.

### 3.1 Programming firmware into TBLCF

1. After you build the interface it is ready to be connected to your computer. Windows will detect attachment of an unknown device and will start the driver installation procedure (see figure 6).



**FIGURE 6. “Found new hardware” window**

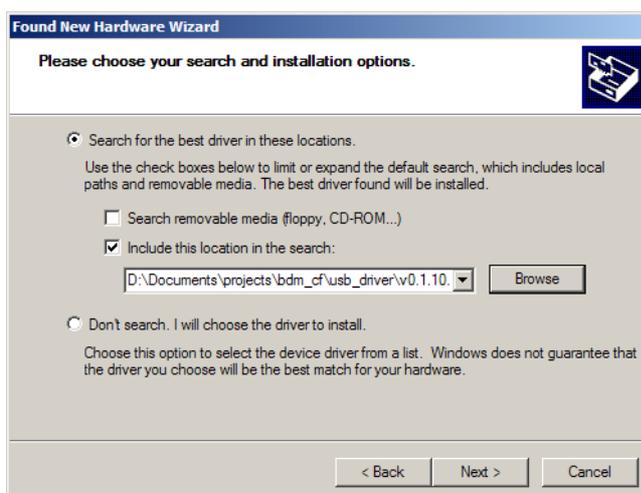
**Note:** Plug the USB cable in quickly. The power supply pins of the connector are longer to ensure that they make contact first. However if you plug in the cable very slowly the JB16 microcontroller will reset and start running the bootloader code before the signal pins make contact - Windows will not recognize the device. Unplug the cable and try again if this happens.

2. Select that Windows should not search for the drivers. Then select “Install from a list or specific location” when prompted for installation mode (see figure 7)



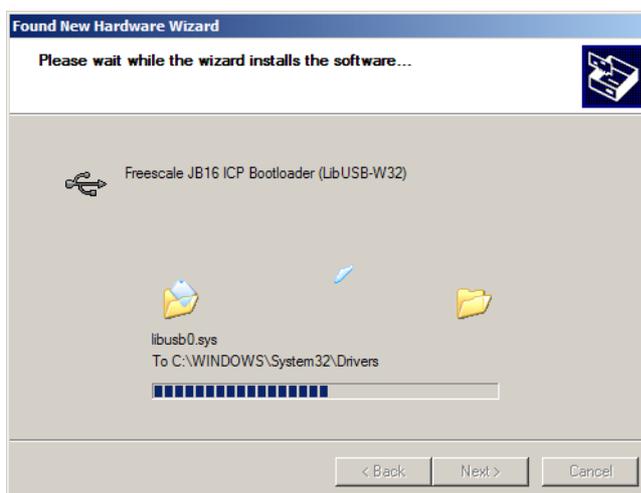
**FIGURE 7. Installation mode**

3. Point the installation wizard to the directory where you have unzipped the driver binary package (see figure 8).



**FIGURE 8. Location of drivers**

4. The wizard will then copy the ICP drivers (see figure 9).



**FIGURE 9. Copying the driver files**

5. After the driver files are copied the interface is ready to be programmed with firmware (see figure 10).



**FIGURE 10. Drivers for In-Circuit Programming (ICP) mode installed**

6. When the ICP drivers have been installed you can use the TBLCF\_BT tool to program the firmware into the interface. The command and a typical response of the tool are shown in figure 11.

```
D:\tblcf>tblcf_bt -B tblcf.abs.s19
Turbo BDM Light ColdFire Bootloader ver 1.0. Compiled on Apr 29 2006, 17:35:29.
S-rec: "D:\Documents\projects\bdm_cf\sw.jb16\tblcf_firmware\bin\tblcf.abs"
found 4 buses
found 1 HC08JB16 ICP device(s)
Boot sector contents different, performing mass erase
Mass erase done, programming boot sector
Programming done, verifying boot sector
Verification done, boot sector OK. You can start breathing again.
Erasing block at address: 0xF800
Programming block from 0xBA00 to 0xCD5A
Verifying block from 0xBA00 to 0xCD5A - OK
Programming block from 0xF9CF to 0xF9FF
Verifying block from 0xF9CF to 0xF9FF - OK
All flash programmed and verified, enabling the application
Flash programming complete, disconnect & reconnect the device
```

### **FIGURE 11. Programming the firmware into the on-chip flash**

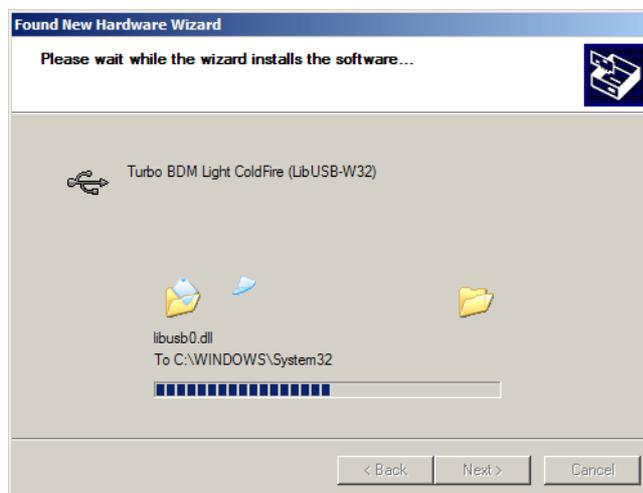
7. The interface is now fully programmed and ready to be used. You now need to disconnect the interface from the USB bus to reset the MCU.

Note: The algorithm of the bootloader is such that the code of the application will only be executed upon power-up in case the flash contents was marked as correct by the bootloader tool (after verification against the supplied S-record file). The only critical part of the programming procedure is upgrade of the boot sector contents. Should programming of the boot sector fail for whatever reason, DO NOT UNPLUG the interface from the USB bus of the computer. As long as power is maintained the interface remains in the bootloader mode and re-programming of the boot sector can be attempted several times until successful. Programming of the boot sector should only be required when programming a newly assembled interface. The bootloader code should remain the same for future versions of the firmware and no upgrades of the boot sector should be required.

## **3.2 Installing drivers for TBLCF use**

Once the interface is programmed with firmware and connected to the computer, Windows will detect attachment of an unknown device and will start the driver installation procedure. The steps required to install the drivers is exactly the same as detailed in section 3.1 on page 14. The correct driver is selected auto-

matically. The only difference you will see is in step 4 where the name of the attached device will be different (see figure 12).

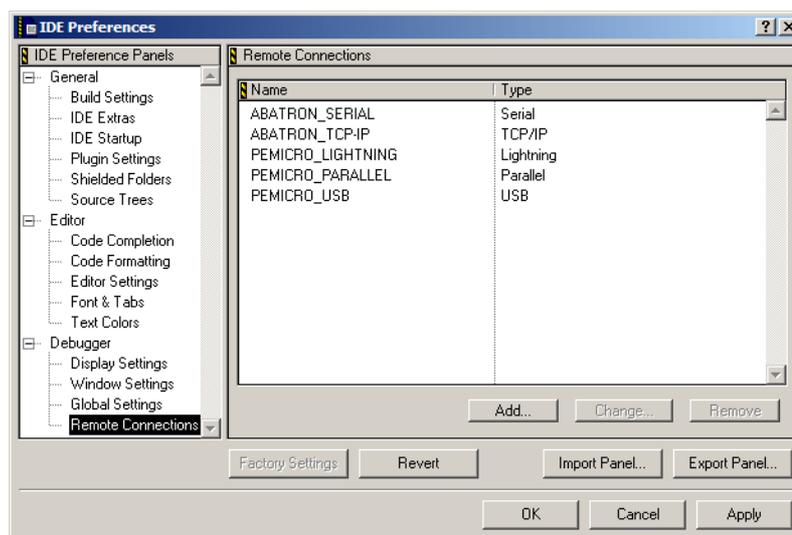


**FIGURE 12. Copying the driver and DLL files**

### 3.3 Using the GDI interface for the CodeWarrior Debugger

The procedure detailed in this section shows how to configure the CodeWarrior debugger to work with the TBLCF interface. Please make sure you download the latest version of the CodeWarrior tools (version 6.3 or newer) as older versions of the debugger do not support the GDI API.

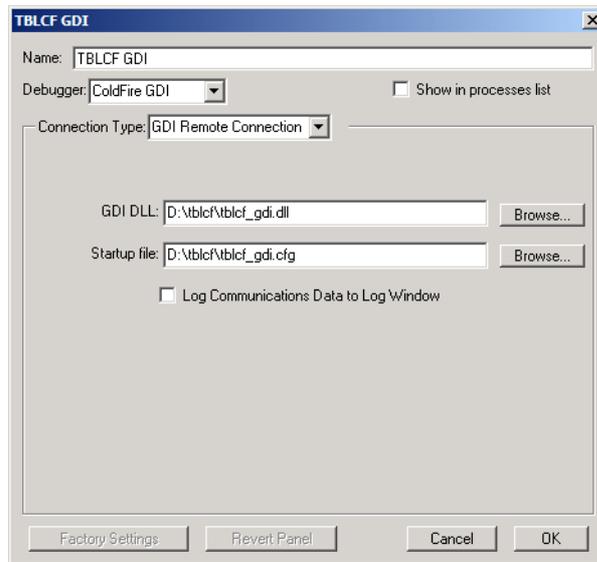
1. Open the CodeWarrior IDE and open your project.
2. Select “Preferences...” from the “Edit” drop-down menu.
3. Open the “Remote Connections” dialogue in the “Debugger” tree (see figure 13).



**FIGURE 13. “Remote Connections” dialogue**

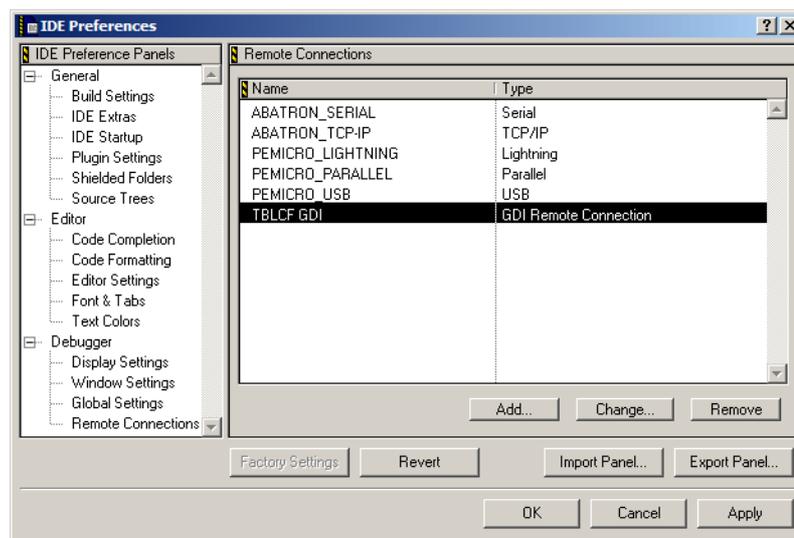
4. Click “Add...”. Enter a name of your choice (e.g. “TBLCF GDI”), select the “ColdFire GDI” from the “Debugger” drop-down list and enter path to the tblcf\_gdi.dll (see figure 14). Enter path to a configuration file in case you

need to pass start-up options to the GDI DLL (see section 3.3.1 for more details). Then click “OK”.



**FIGURE 14. “New Connection” dialogue**

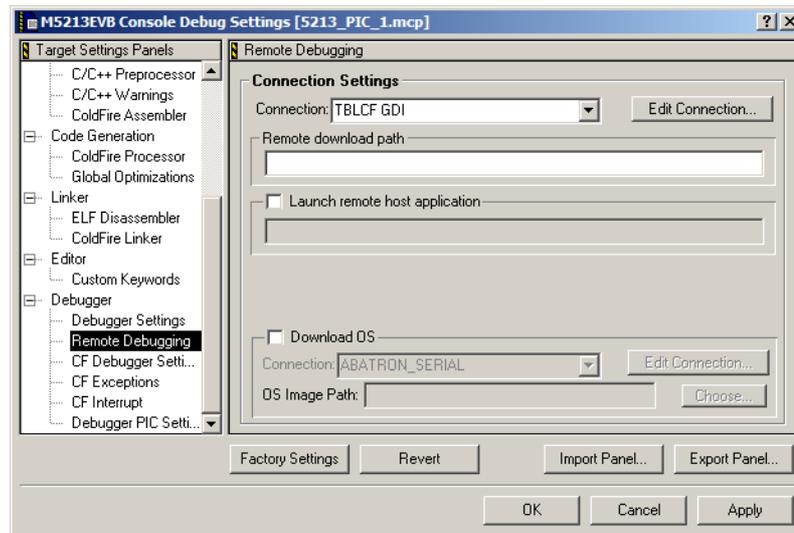
5. The new connection has now been created (see figure 15). Close the “IDE Preferences” dialogue by clicking “OK”.



**FIGURE 15. “IDE Preferences” dialogue with the new connection**

6. Select one of your project’s targets and open the target settings (Alt+F7). Then select “Remote Debugging” settings in the “Debugger” tree. Select

the connection created previously from the “Connection” drop-down list (see figure 16).



**FIGURE 16. “Remote Debugging” settings**

7. Close the target settings window by clicking “OK”. The target is now configured and the debugger will use the TBLCF interface when started.

Note that each target within a project can be configured differently. I.e. you will need to repeat steps 6 and 7 for every target within your project for which you want to use the TBLCF interface.

### 3.3.1 Start-up options of the GDI DLL

The debugger environment is capable of passing configuration options to the GDI DLL after start-up. These options can be used to alter functionality of the DLL based on the requirements of the particular project. Each start-up option is specified on a separate line, parameters are separated by a space character. No comments are allowed in the file. The list of options which are currently supported is given below.

#### USB\_DEVNO n

This option enables the user to specify which TBLCF interface to use in case there are multiple interfaces connected to the PC. The parameter n is a number between 0 and 9. The default value is 0.

## 3.4 Upgrading the firmware

Reprogramming of the firmware is performed in two steps. The first step is to mark the current firmware as invalid which returns the interface to the boot-

loader (ICP) mode (see figure 5). The command for performing this step and a typical response of the TBLCF\_BT tool are shown in figure 17.

```
D:\tblcf>tblcf_bt -U
Turbo BDM Light ColdFire Bootloader ver 1.0. Compiled on Apr 29 2006, 17:35:29.
found 4 buses
found 1 Turbo BDM Light ColdFire device(s)
HC08JB16 ICP will execute on next power-up.
Disconnect and reconnect the device.
```

### FIGURE 17. Marking firmware as outdated

Once the device is disconnected and re-connected to the USB bus of the computer it will start-up in the ICP mode. The new firmware is then programmed into the on-chip flash as per step 6 on page 16 (providing the ICP driver is already installed). In case the ICP driver has not been installed on your computer yet, you need to install it - see step 1 on page 14.

## 4.0 Utilities

### 4.1 Erasing secured devices

The newer members of the Cold Fire family are equipped with on-chip flash memory. The contents of the on-chip flash can be secured to prevent unauthorised copying and modification. Communication with the microcontroller over the BDM port is not possible when the on-chip flash is secured. The only possibility to unsecure the microcontroller and re-establish communication over the BDM port is to erase the whole contents of the on-chip flash. This is achieved by executing a special *LOCKOUT\_RECOVERY* JTAG command.

#### 4.1.1 Enabling the JTAG port

The JTAG port shares pins with the BDM port. A separate JTAG\_EN input signal of the microcontroller selects either the JTAG port or the BDM port. This input signal is typically connected to a jumper. Please make sure that the JTAG\_EN signal is in high logic state before attempting to unsecure the microcontroller. Make sure you switch the JTAG\_EN input signal to low again after the unsecure procedure is completed.

Note that on some boards (such as the M5223EVB) there is more than one jumper which needs to be changed when selecting the JTAG port. Please consult the board manual or schematic before making any changes.

#### 4.1.2 Executing *LOCKOUT\_RECOVERY*

A special utility is provided in the TBLCF package for execution of the *LOCKOUT\_RECOVERY* JTAG instruction. Since the *LOCKOUT\_RECOVERY* instruction differs from one microcontroller to another, the TBLCF\_UNSEC utility requires that the user supplies the binary code of the instruction and its length on the command line as parameters. The erasure procedure of the on-

chip flash requires precise timing. Since the TBLCF interface has no capability to measure the clock speed of the microcontroller it is also necessary to specify the flash clock divider. Details on the *LOCOUT\_RECOVERY* instruction can be found in the JTAG section of the reference manual (specific to the particular microcontroller). Procedure for calculating the flash clock divisor is detailed in the Flash section of the reference manual.

The command and a typical response of the tool when unsecuring the MCF52235 microcontroller are shown in figure 18. The *LOCOUT\_RECOVERY* instruction for the MCF52235 microcontroller is 0x17 in hexadecimal (first parameter) and is 5 bits long (second parameter). On the M5223EVB the device is clocked by a 25MHz crystal. This requires a clock divider between 125 and 166 for correct operation. The selected divider value of 0x4F (third parameter) corresponds to division factor of 128.

```
D:\tblcf>tblcf_unsec 17 5 4F
Turbo BDM Light ColdFire Unsecure Utility ver 1.0.
Compiled on May 11 2006, 13:21:20.
TBLCF DLL version: 10
Unsecure instruction: 0x17
Instruction length: 5
Flash clock divisor: 0x4F
found 4 busses
found 1 Turbo BDM Light ColdFire device(s)
Part ID: ACCAAAAB
Done.
```

**FIGURE 18. Unsecuring the on-chip flash**

## 5.0 Performance

### 5.1 Limits of target crystal frequency

In the current implementation the interface communicates with the target device at a fixed frequency. The ColdFire BDM specifies that the device must be running at a clock frequency at least 5 times higher than the BDM communication frequency. Based on this requirement I recommend to supply at least 4MHz clock frequency to the target ColdFire device to guarantee correct operation of the interface.

### 5.2 Response time and transfer rate

By the nature of the USB protocol the response time for low and full speed devices cannot be below 1ms. I have tried to optimize the communication protocol on the USB to achieve maximum throughput. Practical limitations (caused by the Windows operating system) cause additional delays however. Average

(since under Windows nothing is certain) execution times for different kinds of commands are detailed in the following table.

Command type	Description	Average execution speed
Short	Commands which transfer up to 5 bytes of data into TBLCF and require no return values.	3ms
Typical	Commands which transfer up to 5 bytes of data into TBLCF and request up to 8 bytes of return values.	4ms
Data transfer	Commands which transfer large blocks of data.	6.7 kB/s

When programming the flash of the target MCU there is additional overhead created by the flash programming routines. I have tested the flash programming speed on the MCF52235 device with the following results:

Operation	Throughput
Flash programming	6.6 kB/s
Flash programming and verification	4.6 kB/s

## 6.0 Ideas for further work

1. At the moment the TBLCF interface is supported only under the CodeWarrior debugger. It would be nice to create support for GDB under Linux. The LibUSBWin32 USB driver I am using should provide an easy migration path to LibUSB under Linux.
2. It would be nice to figure out how to support lower clock frequencies for the target device without compromising speed of operation of the interface.

## 7.0 Support

TBLCF is an interface which works with Freescale Semiconductor parts. However it is not a Freescale Semiconductor product. It is not sold nor supported by Freescale Semiconductor.

TBLCF is an open source project. It comes free of charge and so do not expect much in terms of support. If you discover any bugs or have difficulties with the interface please let me know. However, the amount of time I can dedicate to supporting this interface is limited, so please be patient.

## 8.0 License

I am making all the work (with the aforementioned exceptions) available for everyone under the GNU general public license version 2. I do not want to restrict support for the interface in commercial products and I can provide you

with a copy of the work under GNU lesser general public license which enables use of the work in commercial applications - ask for it if you need it.

## 9.0 References

1. LIBUSB documentation, <http://libusb.sourceforge.net/>
2. BDM documentation, COLDFIRE2UM.PDF available from Freescale Semiconductor