

ICEbear User Manual

Martin Strubel <hackfin@section5.ch>

7.3.2010

Revision:

v1.4.1

Preface

The ICEbear is a USB JTAG connector designed to run with the Blackfin CPUs from Analog Devices and the GNU development tools via the Blackfin Emulation Library (libbfemu - see Chapter 2). The libbfemu is part of the ICEbear package and provides a documented API to test, debug and remote control Blackfin hardware via JTAG. The main applications that the ICEbear is designed for, are:

- Target programming and debugging
- Flash programming
- Hardware test for production

Blackfin™ is a registered trade mark of Analog Devices Inc. Cygwin™ is a registered trademark of Red Hat Inc. Linux™ is a registered trademark of Linus Torvalds. Windows™ is a registered trademark of Microsoft Inc.



The ICEbear is a part of an evaluation or development kit and *not* intended to be an end customer device. The developer is responsible for compliance with the local EMI rules when integrating the ICEbear into a system.

The ICEbear uses RoHS compliant components since May 2006.



The ICEbear consists of recyclable electronic parts. Do NOT dispose into the usual garbage. If your ICEbear is assumed to be defective, return it to us. Please contact us first for an RMA (return to manufacturer authorization), or your delivery may be returned.

0.1 Warranty

Martin Strubel / section5 software solutions (so forth 'section5') warrants that the Hardware ("Hardware") shall be free from material defects in design, materials, and workmanship and will function, under normal use and circumstances, materially in accordance with this documentation for a period of *one year* from the date of delivery. Components or the design of the Hardware are subject to change without notice, as long as functionality or operation are not affected.

Defective Hardware returned to section5 within the warranty period will be replaced and sent back to the customer at no charge, solely upon confirmation of a defect or failure of Hardware to perform as warranted.

The foregoing warranties shall be void due to any of the following:

1. if the Hardware has been opened, modified, altered, or repaired, except by section5 or its authorized agents
2. if the Hardware has not been installed or maintained or used in accordance with instructions provided by section5
3. misuse, abuse, accident, thermal or electrical irregularity (voltage excess), fire, water or other peril
4. damage caused by containment and/or operation outside the environmental specifications for the Hardware
5. connection of the Hardware to other systems, equipment or devices or use with other third party software without the prior approval of section5

6. removal or alteration of identification labels or EEPROM serials on the Hardware or its parts
7. failure to comply with all warranty return terms and conditions as set forth herein

To request a warranty service, please contact section5 via the link provided in Appendix C, describing the problem or malfunction. section5 will contact the Customer and confirm the warranty request. The Customer must package the Hardware in the original or appropriate packaging such that further defects during shipping can be avoided. Shipments without confirmation of the warranty request will not be accepted. section5 is not liable for loss or damage during shipment and requires the shipping to be insured for its full value outside the EC.

section5 shall not be responsible for any software, information, or memory data of Customer contained in, stored on, or integrated with any Hardware returned to section5 for replacement whether under warranty or not. Customer is responsible for backing up its programs and data to protect against loss or corruption.

The Hardware and Software is not designed, manufactured or intended for use in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems, direct life-support machines, or any other application in which the failure of the Hardware could lead directly to death, personal injury, or severe physical or property damage or environmental damage (collectively, "High Risk Activities"). section5 expressly disclaims any express or implied warranty of fitness for such High Risk Activities.

Getting Started

1.1 ICEbear variants

1.1.1 ICEbear (classic)

This is the popular ICEbear variant with support for debugging, flash programming and target testing.

If not noted otherwise, all sections in this manual apply to this device.

1.1.2 ICEbear light

This variant is restricted as follows:

1. Support for flash programming only via the ICEbear 'light' software (bfloder)
2. Windows 32 bit support only
3. No support for custom backend development provided by section5

Please find more information in the README files provided with the ICEbear light software.

1.1.3 ICEbear plus

A faster variant supporting USB 2.0 High Speed mode and JTAG clocks up to 30 MHz. Apart from that, it is compatible with the ICEbear classic, except that some extra pins can be used as UART. The ICEbear plus features a status LED.

1.2 Supported platforms

The supported and tested Blackfin CPUs are listed below, in brackets are CPUs that are not extensively tested but are known to work.

1. BF527, (BF52x)
2. BF533, BF531, (BF532)
3. BF537, BF536, (BF534)
4. BF539, BF539F, (BF538)
5. BF54[8,9]
6. BF561

Please see the section5 website listed below for a list of recommended evaluation hardware.

The ICEbear will basically work with all hardware based on the above CPU types and designed according to the Application Note EE68 from Analog Devices ([bib_ee68]C).

The software supporting the ICEbear is working under Linux and Windows NT 32 bit operating systems. For other operating systems and 64 bit variants, please check the web site for updates. Please find all up to date software and documentation on the website <http://www.section5.ch/>

1.3 Installing the software

This section contains the installation instructions for various system configurations. It is assumed, that you either have a CD that came with the ICEbear or that you have downloaded the software at the section5 homepage.

The list of officially supported and tested systems:

- Windows XP, Windows 2000 (32 Bit)
- Debian, Ubuntu (i86 and amd64 Versions)

1.3.1 Windows/Cygwin



It is very recommended to install a Cygwin system on your Windows PC. However, the basic software does not require Cygwin to run. See <http://www.cygwin.com> for more info. Alternatively you may use MSYS/MinGW (<http://www.mingw.org>).

Run the Self Installer (`ICEbear-<version>.exe`, on the CD, this is in `win32`). The installation process should be self explaining. If you have trouble installing the drivers, you can manually install them by right clicking on the file `ftd2xx.inf` in the `drivers` folder and choosing **Install**. When experiencing difficulties, please refer to Appendix A.

After finishing the installation, plug in the ICEbear to complete the driver installation. You can safely ignore the Windows Certification warnings. Now you can start the programs from the ICEbear program menu.

Note that your personal firewall (which might be active on your machine) must allow the port 2000 to be accessed locally. If you get a warning when starting `gdbproxy`, you can safely unblock it.

For development of standalone or uClinux programs, you may also want to install the Blackfin cross compiler toolchain for Windows. This can be found as a self installing executable in the download area of the Blackfin GNU toolchain at <http://blackfin.uclinux.org>. For compiling the uClinux kernel, a Linux development PC is required.

1.3.2 Linux

The ICEbear is known to run with the following configuration:

- Kernel 2.4.X or later (2.6.x recommended)
- `libusb-0.1.7` or greater

Depending on your linux distribution, you will want to choose an installation method from the sections below. The ICEbear software package contains some support files (examples, scripts) that might be important for the user. See package description below about the location of these files.

Debian / Ubuntu

It is assumed that you have a recent Debian installation like the following (tested):

- Debian 4.0 (etch)
- Knoppix 3.2 or later
- Ubuntu 7.x or later

Debian 'lenny' was reported to work not 100% smooth because of a missing libftdi package. To automatically download and update the Debian packages via the web based package manager (apt-get), you may want to put the following entry in your APT configuration file /etc/apt/sources.list:

```
deb http://www.section5.ch/debian <your distribution> main non-free
```

When using a different package manager, just specify the above entry as new package source option.

For supported distributions, please find the most up to date information on the section5 software page (<http://www.section5.ch/software>)

Update your package list with the command **apt-get update**. Then you can automatically install or upgrade your ICEbear software with the command **apt-get install icebear-gdbproxy**. Note that you may have to add other Debian mirrors to `sources.list`, if library dependencies (libftdi0, libusb) can not be resolved.

Other packages that you may want to install:

bfinsight Blackfin variant of Insight debugger (including GDB). This provides the graphical debugger described in Section 2.2 as well as the classical GDB console.

bfloader Blackfin flash loader. See Section 2.3 for description.

bfbin-elf-toolchain The ELF toolchain for blackfin 'standalone' development (no OS). Experimental packages, created from the blackfin.uclinux.org SVN repositories. No support!

libbfemu-dev Developer version of libbfemu. This is required when you want to compile bfloader yourself or want to develop own test routines using the bfemu library. This package also contains the bfemu API documentation

libbfemu-python The python module for libbfemu. This package is not supported in the standard distribution.



When downloading DEBIAN files from section5.ch, you will receive a warning about untrusted sources. This is because section5 is no certified Debian Distributor. If you trust that no third party has been tampering with the binaries, you can ignore the warning at your own risk. Alternatively, see Section B.1 about registering the section5.ch repository as trusted source.

Important support files

- Documentation (you will have to install the libbfemu-dev package also):
`/usr/share/doc/libbfemu-dev/html/index.html`
- Examples: `/usr/share/doc/libbfemu-dev/examples`

Any distribution (from tar file)

Make sure you have one of the newer kernels (2.4.XX or later) with USB device filesystem enabled and libusb installed. Unpack the ICEbear tar file by

tar xzf ICEbear-<version>.tgz

In the `lib/` folder you will find the necessary DLLs (.so's) to link against. If you are not installing them into one of the standard library directories, add their path into the `LD_LIBRARY_PATH` environment variable and modify your Makefile library flags ('-L'), if necessary.

If you are installing the source package, you will have to compile and install each software separately. Read the installation notes inside the directories.

For recent distributions that are using the udev hotplug system, a udev permission file is provided in the `udev/` folder of the ICEbear distribution tar file. Copy this file (while being root) into `/etc/udev/rules.d` and restart udev via the command `/etc/init.d/udev restart`. This file then properly sets the permissions whenever an ICEbear is plugged in.

1.4 Connecting the ICEbear

First, remove the power from the board or unplug the ICEbear from the USB port. Avoid to plug the ICEbear into the board while any software accessing the ICEbear is running or when the ICEbear is plugged into the USB port. In worst case, this can damage the hardware if there are ground potential differences between the USB port and your target.



The ICEbear plus has a status LED showing whether it is in a JTAG session with the target. Do not unplug while it is lit, rather, quit the session on the host PC first.

Connect the ICEbear as shown in Fig. 1.1 below. The JTAG connector has a key on pin 3 which prevents it from plugging it in the wrong way.

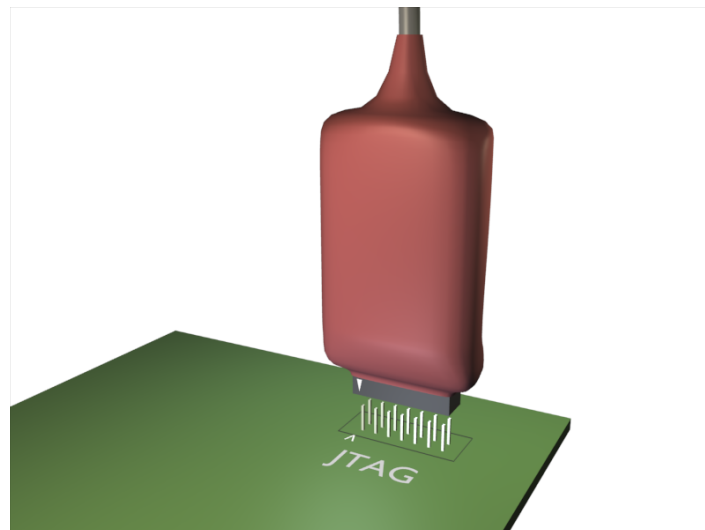


Figure 1.1: Connecting the ICEbear to the JTAG port



Do not connect the ICEbear to a board that was modified to run with another JTAG adapter that needs voltage supply from the board. This will possibly destroy the ICEbear and make all warranty void.

When removing the ICEbear, hold the bottom connector firmly with two fingers and gently wiggle it out lengthwise. Do not apply too much force to not bend your JTAG pins or stress your PCB. The JTAG connector is a high quality header that withstands thousands of connection cycles, but less stress will extend its lifetime.

1.5 Running the software

To test whether the adapter can talk to the target, execute one of the examples, for instance in Linux:

icebear-dumpreg

or **Run Demos** from the ICEbear program menu in Windows.

If everything is properly connected, this will give you a register dump of the current cpu state.

Alternatively, you can start gdbproxy and fire up the debugger - see next chapter.



The ICEbear debugger toolchain is a very powerful and complex environment. Make sure you read the manual thoroughly and follow the Application Notes to get a feeling for the behaviour of the debugger.

Software

Please note that there is no free support for the GNU toolchain. This software is supplied AS IS, please read the GNU license and usage statements. However, we will try our best to keep the GNU toolchain working optimally with the *ICEbear*. New with the 1.0 release is the Insight debugger. This is a GDB port made by Red Hat and Cygnus Solutions, adapted to Blackfin by section5. More details below.

Developer documentation is found in [bib_bfemu]C.

2.1 gdbproxy

For debugging a JTAG hardware target, **gdbproxy** serves as a remote debugging daemon which the GNU Debugger (GDB) can connect to. **gdbproxy** translates all the GDB commands into the appropriate JTAG sequences via *libbfemu*, which is driven by an added target 'bfm'. To get help about the target specific options for **gdbproxy**, type **gdbproxy --help bfm**. Omitting the target choice 'bfm' shows the common options.

An important option is the `--speed` option. For example, to use maximum speed of the *ICEbear*, start **gdbproxy** with the following arguments:

gdbproxy bfm --speed=0

The complete list of options is show below.

New with version 0.9 is the auxiliary telnet server, tunneling stdio functionality (see *bfpeek* library, Section 4.3). This allows simple file I/O on the target using a tiny *newlib* wrapper that calls the *bfpeek* library.

The telnet server currently supports a simple UART emulation only. See for Section 4.3 application examples and how to connect to the server.

Option name	Description
<code>speed=<value></code>	Change JTAG clock speed value. The maximum JTAG clock frequency on the <i>ICEbear</i> (6 MHz) will divide by the value plus one. The default speed value is 0.
<code>buffersize=<size></code>	Change write cache buffer size to specified size. If 0, disable write cache.
<code>dev=<index></code>	Selects the attached JTAG device, in case several are attached.
<code>cpu=<index></code>	Selects CPU <index> in a multi core environment such as the BF561: 0 = Core A, 1 = Core B
<code>debug</code>	Enables debug output mode. Use this mode only, when you suspect a bug in the target handling
<code>config=<infile></code>	Selects a INI file for multiple devices chain configuration. See Section 4.2.
<code>auxport=<portnum></code>	Set auxiliary telnet server port

Table 2.1: *gdbproxy* Blackfin specific options

2.2 Insight Debugger

The Insight debugger is a GDB variant with built in GUI. It has some extra features and is very powerful. For beginners, it is recommended that you start with Insight. Note that Insight is free software, source code is available at <http://sources.redhat.com/insight/>. Please make sure you use the version from the section5 website.



Other variants such as the uClinux versions are *not* compatible and will cause problems when loading programs or while debugging.

2.2.1 Setting up and configuring Insight

Before you can start debugging your target, you will have to do a bit of setting up your environment. For example, if the executable that you want to run on your target lives in SDRAM, you will have to do the necessary setup of the SDRAM controller (Synchronous EBIU). You can do all these setups manually by writing to a specified address, but GDB scripting can simplify that task greatly.

Compiling Insight

This section only applies to advanced users who wish to compile Insight themselves from the source. It is assumed that the user is familiar with the typical `configure/make` procedure.

1. Unpack the source tar file: `tar xzf insight*.tgz`. We will refer to this unpacked directory by `$INSIGHT_SRC`
2. Create a build directory somewhere on a scratch partition: **`mkdir build`**
3. Enter this build directory by **`cd build`**, then execute: **`$INSIGHT_SRC/configure --target=bfm-elf --disable-sim`**
4. If the configure script fails, you may need to install additionally required packages. Otherwise, you can now run **`make`** to build.
5. **`make install`** will finally install the entire package in `/usr/local/`.

Auxiliary scripts

A few auxiliary GDB scripts are provided in the location below. These are:

`mmr.gdb` Some of the important MMR register definitions. These are common to all blackfin architectures.

`mmr_bf*.gdb` Platform specific MMRs

`init.gdb` Some example initialization scripts for standard boards

`dump.gdb` Some register dump helper functions to display interrupt status, etc.

`catch_exc.gdb` Example on how to catch exceptions

`auxlinux.gdb` Some linux auxiliary macros

OS-dependent locations:

Windows	<code>\$(PROGRAM_FILES)/section5/ICEbear/scripts</code>
Debian Linux	<code>/usr/share/gdbscripts</code>

To load these scripts, either use the 'source' command or load it via the Menu **File->Source**. You can then use these register definitions for example in a Watch window (see below in Section 2.2.2).

.gdbinit Startup script

For a specific project, you certainly do not want to repeat initialization procedures over and over. Therefore, it is recommended to write a .gdbinit script in your project's directory. When you start **Insight** from that directory, it will automatically load the .gdbinit script. The sample script below demonstrates how to automatically connect to the target (provided that gdbproxy is running) and do the necessary initialization. Note that you have to strip the comments (including the '#') when copy pasting the file.

```
file blink.dxe                # Select blink.dxe executable
target remote :2000           # Connect to the local gdbproxy

source /usr/share/gdbscripts/mmr.gdb  # Load MMR definitions
source /usr/share/gdbscripts/dump.gdb # Load Dumper functions

set prompt (bfin-jtag-gdb)\      # Set the prompt

define target_init            # Define target initialization function
    monitor reset              # Reset the target
end
```

If your program uses SDRAM, you will have to put some SDRAM initialization code in your function 'target_init', like below:

```
define target_init            # Target initialization with SDRAM
    monitor reset
    set *$EBIU_SDGCTL = 0x0091998d
    set *$EBIU_SDBCTL = 0x0025
    set *$EBIU_SDRRC = 0x0817
end
```

Note that this example is for a specific board. See `init.gdb` for a few default initializations. For your own board, you must possibly adapt these values to SDRAM configuration and system clock. See your Blackfin Hardware Reference.



If the EBIU and system settings are not correct, the Blackfin can core fault on SDRAM access. This can not be caught by the debugger. You will then have to reset the system.

You can also put system wide declarations in a .gdbinit script residing in your home directory. Note however, that all variable declarations are cleared when you switch executables. So, calling `mmr.gdb` from `$(HOME)/.gdbinit` has no effect. However, a function definition ('define') remains. If you are accidentally including a script with function definitions twice, Insight will complain.

Insight is not very communicative on startup script errors. Before trusting a script, test it by starting Insight with the '-n' option (no startup script execution) and load it explicitly in the console using the 'source' command. Whenever a user defined command such as `target_init` fails, the console will display an error message.



When starting Insight in Windows via the Start menu, the `.gdbinit` script in the scripts directory (as specified in Section 2.2.1) is executed by default. You can change the working directory via the **Properties** menu of the Insight debugger's red bug icon (Right click on the icon to call this menu).

Setting up Insight

Before you can connect to the target, you need to specify some parameters. This is done by calling the configuration dialog below (Fig. 2.1) via **File->Target Settings**.

When your program fails and you need to reconnect and download again, you would want to call the initialization function `target_init()` whenever you connect or run the program. Click on 'More Options' to expand the dialog like shown below and enter 'target_init' in the text field.

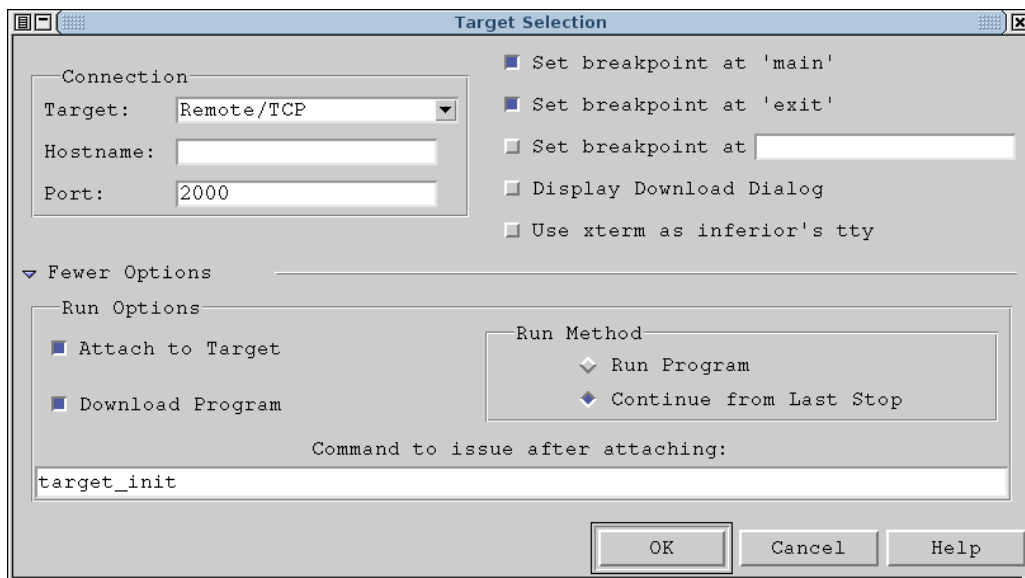


Figure 2.1: Target configuration

After you have set up your target, make sure that `gdbproxy` is running and has properly detected your target. Then you can connect to the target within Insight via **Run->Connect to target**. The next step is, to download the program (**Run->Download**) and call '**Continue**'. You can also execute '**Run**' immediately. The target will then halt at the first breakpoint ('main', as specified in the Target Selection dialog).

When you need to reload the program, you can either cycle through **Disconnect/Run** via the **Run** menu or define a script with the necessary initializations. Read more below on running and debugging example code on the target.

Special commands

Please refer to the GDB documentation for all GDB commands. All 'monitor' commands are implemented in `gdbproxy` and considered 'special'. These are:

monitor reset Resets the target (core and system)

All GDB commands can be entered via the console window (**View->Console**).

2.2.2 Examining programs and variables

Download and run

For a simple example session with Insight, we provide a rudimentary standalone program that does not more than a bit of LED blinking on a few common evaluation boards (like ADI EZKITS). The source can be found at <http://www.section5.ch/software>. See the README file in the `blinky.tgz` tar file for compilation instructions.

Open the executable after a successful 'make' call via the menu **File->Open**. If it is compiled with the `-g` flag, the source code will be displayed in the window.

To download the executable onto the target, connect to it as described in Section 2.2.1 and execute **Continue** to start.

In Section 3.1.1, a few step by step instructions are given for debugging `blinky` using the GDBs command line interface. If you are new to debugging with GDB, it is recommended to go through this little exercise.

Source code and assembly debugging

By default, and if you have compiled your program with the `'-g'` flag, you will see the native source code (C, C++, Assembly) in the main window. The current location of the program counter is marked by a green bar. With the drop down menus on the top you can select other files and functions. On the right drop down, you can change the display mode. For example, the mixed mode displays interleaved C source and disassembly as shown in Fig. 2.2.



Note that mixed or assembly mode display can take a long time to read from the target, when your program lives in L1 onchip memory. If you want to avoid that, use the technique explained below.

If you run a program without debugging information, there is no source code or disassembly display available. You can still look at the executed opcodes via the console (**View->Console**). The command `display/i $pc` displays the assembly command at the current PC location. For convenience, you can step into or step over using the `si` and `ni` command on the console.



Simply pressing 'Return' on the console repeats the last command

Variable watching

You can view the current content of a variable any time by moving the mouse pointer over its location in the source code. It will automatically retrieve the value and display it. Note that this can take a long time on variables such as buffers or strings. You can turn this behaviour off via **Preferences->Source->Variable Balloons**.

If you want to repeatedly watch a register or global variable, you can use the Watch window (opened via **View->Watch Expressions**). For example, if you want to look at the interrupt state of the core, you can display the IPEND register (which is defined in `mnr.gdb`) by entering the expression `*$IPEND` into the text field left to the **Add Watch** button like shown below (Fig. 2.3). Normally, variables are displayed as decimal. If you want to change the display, click with the right mouse button on the variable entry and choose via the **Format** menu.

2.2.3 Finding bugs

Finding a bug can sometimes be an art, especially on a complex embedded system as a Blackfin driven platform. Most common errors when starting with a standalone development (without

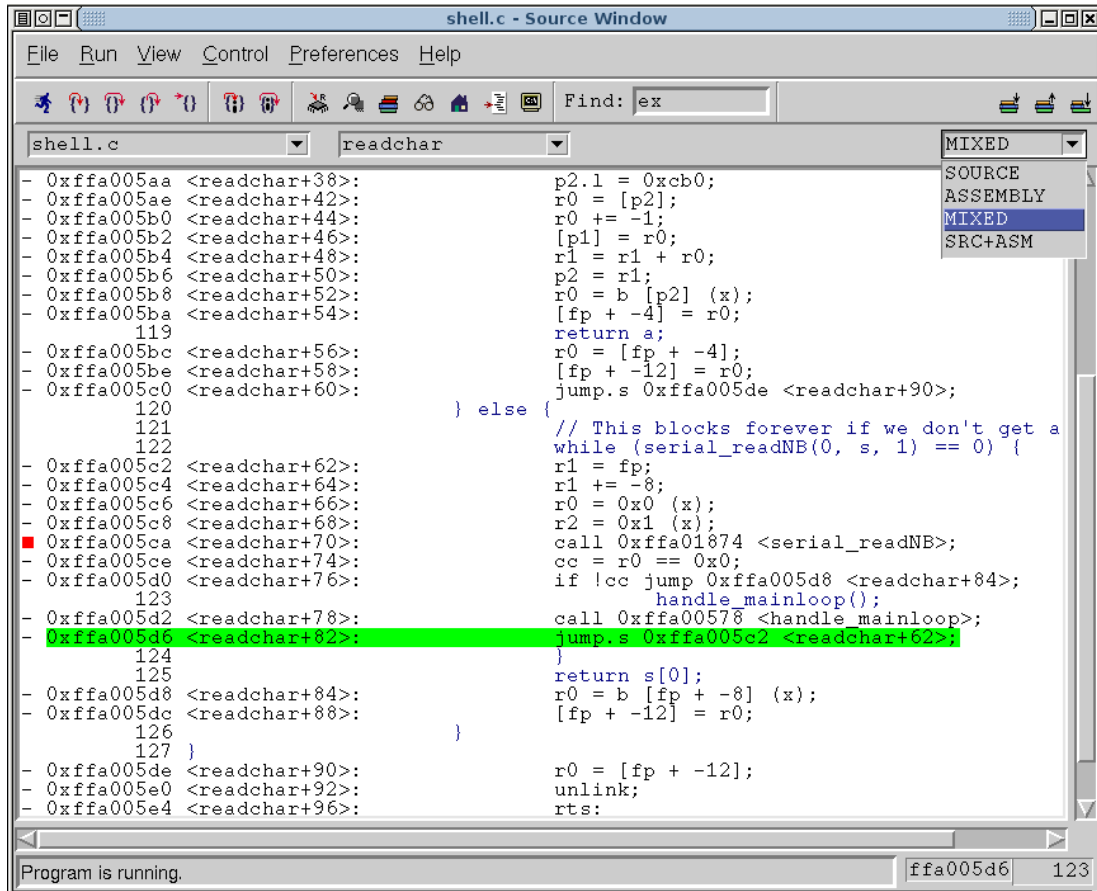


Figure 2.2: Source Window

uClinux) on Blackfin systems are due to hardware or other exceptions. Generally, it is a good idea to install exception handlers in L1 SRAM and set break points on the exception handler routines - look at the `catch_exc.gdb` script. This must obviously be called *after* the exception vectors have been initialized. For concrete tips on how to track down a problem, please see 'Debugging Tips' section in the the support forum at <http://www.section5.ch/forum/>. A few general hints are listed below.

If Insight stalls and does not react to user input after a certain sequence of stepping over the code, it is mostly not to blame to a debugger flaw. Find out by reproducing the situation what exactly happens on the target. Single instruction stepping should always work, however, stepping over C source code can sometimes jump into subroutines and single step through them which can take a long time. If you can wait, it helps to be patient. In case of such events, you can try to track down what is going on by several ways:

- Look at the gdbproxy console. If there are core faults, they mostly are due to a failing exception handler, for example a handler living in SDRAM which was not properly initialized. Such errors cause double faults and make the core stall. You then have to reconnect (within Insight) or reset the target via 'monitor reset' or in heavy cases by pressing the reset button.
- Enable the debug mode of gdbproxy by enabling the `--debug` option. If you are lost, send a dump of this output to section5, possibly we can help you.
- Write a script that single steps though the code until a certain event occurs. This can be

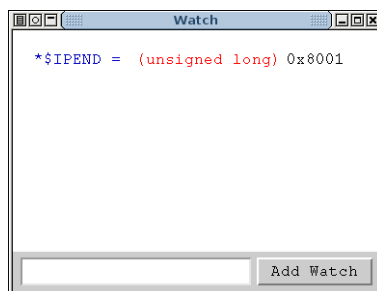



Figure 2.3: Watch window

done with a while/end loop, see GDB script syntax in the GDB online manuals

Find more practical examples in Chapter 3.

2.3 bfloder

bfloder stands for 'Blackfin Flash Loader' and consists of a collection of GDB scripts and a standalone executable to program the flash of common Blackfin targets. This software is provided for free and as source code. Users can easily adapt the source for the backend target to their customized hardware. The bfloder distribution can be downloaded from <http://www.section5.ch/software>. Documentation can be found in the `doc/` sub folder of this package. Below, only the standalone flash loader commands are listed.

 To compile bfloder from source, you need the *libbfemu-dev* and the *libelfg0-dev* package from the Debian packages, or the header files from the distribution tar file. In the latter case, you might have to adapt the header locations in the Makefile.

2.3.1 Usage

This standalone executable does not require a debug agent (gdbproxy) to run. Parameters are passed with option flags like shown below:

```
flashload --info --backend=ezkit_bf533.dxe --eraseall --program=uboot.ldr
```

The description of command options is listed in 2.2. Running the flashloader without arguments displays a list of supported options.

Command options

Note that all file related actions (program, dump, compare) can only be used at a time.

Examples

Here is a list of examples on how to use the flashloader. Note that this assumes the appropriate backend (`bfloder.dxe`) in the current working directory.

Erase and program a flash with a full flash image `flashload --eraseall --program=flash.img`

Program a flash with a partial flash image from offset 0x10000 `flashload --program=flash.img --offset=0x10000`

Partial erase of flash `flashload --erasesectors=0,32`

Display of Flash sector table `flashload --table`

Verify flash content against binary file image `flashload --compare=flash.img`

Read first 1M partition of a boot flash `flashload --dump=flash_read.img --offset=0x0
--size=0x100000`

2.4 libbfemu

The Blackfin Emulation library *libbfemu* (or *bfemu*) is a library which supports you in testing your hardware via JTAG. In conjunction with the ICEbear, it is a very important component of a hardware production chain. A short feature overview:

- Register read/write
- Memory read/write
- Program sequencer control: Stop, go, single step
- Software breakpoints. Hardware breakpoints are currently not used due to Blackfin anomalies.

libbfemu comes with examples. The examples have been tested under Linux and MinGW/Cygwin. The *bfemu* documentation is included in the ICEbear software distribution. For automated system tests, section5 has also developed a Python scripting interface that allows downloading of programs and complete remote control of the target using the simple Python scripting language. This is available on request and not part of the standard ICEbear software distribution.

Option name	Description
info	Displays information about flash driver and flash geometry
table	Displays the flash sector geometry table
version	Displays the program version
program=<filename>	Write the binary image in file with name <code>filename</code> to flash. If none of the <code>size</code> and <code>offset</code> options are specified, <code>offset</code> is assumed 0 and <code>size</code> the size of the file.
dump=<filename>	Dump the content of the flash specified by <code>offset</code> and <code>size</code> to a file. Specifying <code>size</code> is mandatory.
compare=<filename>	Compare flash content with the binary image in the given file. If specified, the comparison is started from <code>offset</code> up to <code>size</code> . Otherwise, <code>offset</code> is assumed to be 0 and <code>size</code> to be the size of the file.
check=<filename>	Compare flash content like 'compare', but use much faster CRC32 checksum method.
config=<filename>	Specify JTAG chain configuration file (see Section 4.2)
cpu=<index>	Select specific CPU in chain (only with adapters supporting multiple JTAG devices)
dev=<index>	Specify device index when using more than one ICEbear. Default is 0 (first device). See Section 4.1 for more information.
eraseall	Erase entire Flash chip. This can take a few minutes, so do not interrupt the process.
erasesectors=<off>,<n>	Specify the number of sectors <code>n</code> to erase, starting from sector number <code>off</code> . You must know the sector geometry of the Flash in order to determine the corresponding addresses (see <code>table</code> option) This option is only relevant when you use a partitioned flash.
offset=<Address in hex>	Specifies an offset in hexadecimal format. You must use a prefix '0x' for this hexadecimal number, e.g. <code>--offset=0x1000</code> .
size=<Size in hex>	Specifies a maximum size to read, write, or compare. The format is the same as in the <code>offset</code> option.
noverify	Turn off verify after write. Verify is always on by default, if this option is not specified.
backend=<backend>	Specify the loader backend to load on initialization. If not specified, the default compiled in backend name ('bfloder.dxe' in the current working directory) is used. This must be a valid bfloder backend, VDSP backends can not be used.
speed=<value>	Change JTAG clock speed. 0: fastest, 255: slowest Default is 1.
unit=<index>	Specify internal flash unit when backend driver supports several attached devices such as the SPI driver. See driver source code for detail.

Table 2.2: List of bfloder command options

Application Notes

In this chapter, a few typical debugging applications are described. This is just a limited selection, please find more information on the section5 user forum at <http://www.section5.ch/forum>.

3.1 Standalone debugging examples

This describes a typical debugging session with a standalone program running on the target. Standalone means, that there is no operating system running, the executables run in supervisor mode and have access to the hardware. All the examples can be found at <http://www.section5.ch/software>. It is assumed that you are equipped with a complete bfin-elf-toolchain including the typical GNU build tools such as 'make', etc.

3.1.1 blinky

The blink program, being considered the "hello world" of the embedded domain, is a very primitive example just driving a few LEDs. The only extra is some assembly startup code needed to run a main program. This code is found in `crt0.asm`, but does normally not need to be touched, unless more setup options or alternative starting options are desired. The actual blinking code is found in `main.c`. The file implements a `do_blink()` function for each supported board type. The list of currently supported boards is printed when typing "make" inside the `blinky/` subdirectory.

We will now practise download and debugging using the command line version of GDB. While Insight appears nicer with its GUI, you might find yourself being faster with the command line version when debugging a simple standalone executable.

For the following description, all commands that you enter are marked bold, surrounding output appears in normal style.

1. Compile blinky for your board. For example, for the CM-BF537 board from Bluetechnix, enter the `blinky/` directory within a shell and type
make BOARD=CM_BF537 clean all
When the build was successful, you will find a file `blinky.dxe` in the current directory.
2. To download this file onto the target, you need to start `gdbproxy` on another console. Make sure the ICEbear is plugged in and connected to the target, then start the debug agent by
gdbproxy bfin
`gdbproxy` should then come up with a bit of output, ending with
notice: gdbproxy: waiting on TCP port 2000
3. Start up the Blackfin GDB from the `blinky/` directory:
bfin-elf-gdb blinky.dxe
GDB will automatically load the `.gdbinit` script residing in this directory. This script defines a few commands, making things easier. However, it is not yet connecting to the target, as we want to leave a bit of exercise to you.
4. Connect to the target (more precisely, to the `gdbproxy` debug agent) with the following command inside GDB:
(`bfin-jtag-gdb`) **target remote :2000**

This internally opens a TCP connection to the local host (your PC) to the above port (it is of course possible, to run the proxy on another PC and/or port). Once connected, you will read the following on the gdbproxy console:

```
notice: gdbproxy: connected
notice: Detected CPU: BF-537(536), rev: 2
```

This means that the gdbproxy agent was able to talk to the target hardware.

5. Now you can poke around on the target. However, we first would like to download the blinky executable. This is done by the command sequence

```
(bfin-jtag-gdb) monitor reset
(bfin-jtag-gdb) load blink.dxe
```

If you look into the .gdbinit file, you will see a user defined command named 'init' defining exactly this. When typing 'init', GDB will output the following on a successful download:

```
Loading section .text, size 0x1fc lma 0xffa00000
Start address 0xffa00000, load size 508
Transfer rate: 290285 bits/sec, 508 bytes/write.
```

So, you can see that only the L1 program memory is used.

6. Start blinky with the "continue" command:

```
(bfin-jtag-gdb) load blink.dxe
```

You should now see the LEDs blinking a few times. If you hit Ctrl-C inside GDB, the program will stop:

```
Program received signal SIGINT, Interrupt.
delay_loop () at auxiliaries.asm:24
24 delay_loop: nop;
Current language: auto; currently asm
```

The break location might be different, depending on when you interrupt. While LEDs are blinking, it is most likely that you catch the program in the above delay loop (where it is nothing but burning CPU cycles in between the LED toggling). You are now inside assembly code, if you wish to return to the main program, type "fin". This continues running the program until it returns from the current subroutine.

7. The program should now be in the do_blink() routine. If you type "bt", you should see the nested stack frames, for example:

```
(bfin-jtag-gdb) bt
#0 do_blink () at main.c:72
#1 0xffa001f8 in main () at main.c:135
```

If it isn't there, it may already have hit the end of the blinking sequence. You can simply reload by starting again in (5).

8. If you type "list", you see the current source context. Obviously, you are inside a while() loop, and you just stepped out of a function delay(). You'll notice a local variable i.

Examine this variable using

```
(bfin-jtag-gdb) print i
```

If i is below a certain value, the LED I/Os will keep being toggled. So you can force the end of this loop by modifying i:

```
(bfin-jtag-gdb) set variable i = 40
```

When now continuing with 'c', the LEDs will stop blinking. Interrupting the program will show it being stuck in the endless loop at the end of main().

9. You could now start setting breakpoints. Restart the program by typing "init" (this is a user defined command in `.gdbinit`, resetting the target and reloading the executable), and before you continue, type
(`bfm-jtag-gdb`) **break do_blink**
This will set a breakpoint in the blink routine, the program will immediately halt after a 'continue':
Breakpoint 1, do_blink () at main.c:63
63 *pPORTG_FER = 0;
Note that the above source code is board specific and might differ.
You could now step through the code by typing the 's' command. Note that this way of stepping is a source code related stepping. If you want to see single assembly commands being stepped through, see 'si' (single instruction) command.

3.1.2 shell

The shell example is a bit more complex as it makes use of the C library (newlib) and allows user input via the console. To build, you need to install the `libbfemu-dev` DEBIAN package or adapt the Makefile to find the `bfpeek` headers and library in the path to your `bfemu/ICEbear` software installation.

See `shell/README` for more information and how to talk to the shell using a terminal program. Typically, the reason for running executables via JTAG is, that things don't work, crash, hang, or produce other spurious, unexpected results. So we will provoke some errors that are not caught by the shell environment (because it's not an operating system, and you can poke around on supervisor resources).

For example, unaligned accesses such as word access to odd addresses, cause an exception and the shell to hang. To verify this, type

dw 1 1

on the shell console, which is supposed to dump 8 words from the address given in the first argument (hex). It does now no longer respond to commands and you'll see the following on the gdb console:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to thread 1]
EXC_HANDLER () at ../../crt0.asm:316
316 emuexcpt;
```

Obviously, the program jumped out from the main loop due to an exception whose exception handler routine was initialized (as it should be) by the startup code. To determine the cause of the exception, you would have to examine the various system status registers. This is where the GDB scripting comes in handy, it is already simplified with a number of user commands from the `dump.gdb` script which is included from the `.gdbinit` script in the current directory. So you can use the user defined 'dump_exc' command to find out what happened:

```
(bfm-jtag-gdb) dump_exc
-----
-- Exception !!
-- EXCAUSE: 24
-- Reason: "Address error / misalignment"
-- Instruction: 0xffa016b0 <dump_memory_short+52>: r0 = w [p5++] (z);
-----
```

Likewise, the 'dump_hwerr' command displays reasons for possibly occurred hardware errors. To improve detection of these errors, there is another auxiliary script 'catch_exc.gdb' that sets breakpoints into the exception handlers. Because they are initialized during runtime, you must load this script *after* the program has initialized the jump vectors in the `$EVT` array (the `$EVT`

variable is defined in the GDB script `mmr.gdb`). This can be tricky to setup. You may, right after initialization, set a breakpoint that executes the script once (see 'help command' within gdb). For the shell program, you can just interrupt after startup and load the script manually. Since the shell initializes most of the exception vectors to the above `EXC_HANDLER` routine, explicit setting of a breakpoint is not necessary, because the `emuexcpt` instruction will automatically trigger the JTAG debugger.

Loading the script works with the command

```
(bfin-jtag-gdb) source /usr/share/gdbscripts/catch_exc.gdb
```

On Windows, this script is in a different location, see installation details in Section 2.2.1.

3.2 Debugging the uClinux kernel

When developing new drivers or tracing down spurious crashes, hardware problems, you may want to debug very close to the hardware using the ICEbear. This section describes how.



For user space application debugging, you may want to use the 'gdbserver' tool via TCP/IP. See [bib_bfinuclinux]C for more information.

3.2.1 Preparing

Recompile the uClinux kernel with the option `CONFIG_DEBUG_INFO=1`. This can also be passed to `make` as parameter. This enables all the debug information necessary for source code debugging. Enabling other kernel debugging options may be also helpful.

There are two methods of downloading to the target. The first one is recommended when a bootloader and an Ethernet interface is present or the kernel is booted from flash. The second method downloads the ELF executable directly via JTAG and can be somewhat slower than the first.

Before loading the uClinux kernel onto the target, you have to properly configure the SDRAM. Not doing so will result in Core Faults when downloading to the target.

3.2.2 Debugging with the Boot-Method

This describes shortly how to debug and attach to a booted/running uClinux system:

1. Start `gdbproxy`
2. Reset and boot your uClinux system on the target
3. Start GDB with the "vmlinux" image specified in the argument
4. Connect to the target using
target remote :2000
On success, the target will halt (as you can verify on the console) and the current execution stack can be shown using 'bt'.
5. Use the `catch_exc.gdb` script to set breakpoints in exceptions, or set your breakpoint in a kernel driver module, etc. For example, a breakpoint can be triggered in `cpuinfo_open()` by typing
cat /proc/cpuinfo
on the uClinux console.

If the error happens too early, i.e. uClinux does not boot successfully or output anything on the console, you will have to intercept the system at an earlier stage. Please see section5 forum in the section 'Debugging Tips'. Also, have a look at the 'dmesg' macro defined in `auxlinux.gdb` of the gdb script collection.

3.2.3 Debugging via JTAG download

Before you download, note that you have to preinitialize a few things, if not done previously by u-boot. Make sure the SDRAM is configured right, also, you may want to change PLL parameters (if the kernel does not reconfigure it).

Downloading via gdbproxy is quite slow, due to its architecture. There are optional scripting solutions (not included with the default ICEbear software package, contact section5) that allow faster downloads. Alternatively, you can write your own debugging frontend using the bfemu library.

Download of the vmlinux kernel works as with any ELF executable via the “load” command. However, since it is loaded into SDRAM, preconfiguration must take place via GDB scripts. See the init.gdb auxiliary script for a few board defaults. The download can take a long time, therefore you might want to only download the kernel and put the root file system on a flash partition. This method is only recommended when you have no way to talk to u-boot via a serial console.

3.3 BF561 Core B debugging under uClinux

The uClinux distribution for the EZKIT-BF561 or similar platforms is designed such that uClinux is running on Core A while Core B can be used for DSP applications. For debugging Core B while not touching the system on Core A, we describe a possible way here.



For this feature, you will need at least gdbproxy 0.8.5 with the proxy driver v1.3 or greater

The example used is a simple blink program running on Core B, called **core_blink**. It is compiled with the bfin-elf-toolchain variant, as opposed to a linux executable compiled with the bfin-uclinux-toolchain. You can find this example plus more information in the blinky source distribution found on the section5 software home page at <http://www.section5.ch/software>.

1. Copy the executable onto the target or into a directory that is NFS mounted on the target
2. Start gdbproxy:
`gdbproxy bfin --cpu=1`
3. Run the corebdl application on the target to download the executable to core B:
`corebdl core_blink`
4. Connect to the target from GDB/Insight
5. Debug the application on Core B. You can now reload the application from GDB using (bfin-elf-gdb) **`load core_blink`**

Note that first time you will need to download via uClinux to unlock the core for memory accesses. A workaround may be found for future versions. Currently, you will receive an error message from GDB when trying to download the executable to a locked Core B.

3.4 GPIO debugging

In many cases, when not using a port multiplexer framework like provided by uClinux or the shell code, concurring accesses to peripherals can cause lots of confusion when interfaces just don't work. This is most likely due to silent reconfiguration of the GPIO ports by another portion of your code.

To check enabled ports, use the `dump_gpio` script from the `dump.gdb` auxiliary. This will show you the CPU specific settings of your GPIO configuration MMRs. With some architectures, the register bits are parsed and the configuration is printed verbosely.

Some of the register defines are specified in the `mmr_*.gdb` architecture specific files. See also Section 2.2.1.

Advanced

This chapter lists a few options for advanced users.

4.1 Connecting multiple ICEbear units

Multiple units per computer are supported, however you must use all ICEbear or all ICEbear plus at a time.

Mixed device combinations will not work. Also, this option does not apply to the ICEbear light. The enumeration of the devices is very OS dependent. So as of now, there is no deterministic rule given how the device indices are spread among the connected physical devices.

To address a device, utilities like bfloder or gdbproxy have a `--dev` option. See tools description in Chapter 2.

4.2 Multiple core support

The ICEbear can handle multiple devices in a chain. With daisy-chained Blackfin devices, the CPU types are automatically detected.

On the ICEbear Plus, these need not to be necessarily Blackfin devices.

To support alien devices, a INI file with parameters must be specified to gdbproxy using the `--config` option.

bfloder v2.0 also supports configurations using the same option flag. To address a specific CPU in the chain, use the `--cpu` option.



section5 does not guarantee that this works with your hardware. It is absolutely important that you design the JTAG chain according to the high speed considerations listed in EE-68 ([bib_ee68]C).

4.2.1 INI file syntax

The syntax of a INI file is common to the usual configuration files from Windows and Linux. Each device forms a section which contains a number of parameters. Example:

```
[Blackfin_BF527]
irsize = 5           # Size of Instruction register
idcmd  = 2           # Command for requesting IDCODE
id     = 0x027e00cb  # IDCODE BF527
type   = Blackfin   # Device type (required for Blackfin)
idmask = 0x0fffffff # Mask (ANDed before comparison with IDCODE)
```

Optionally, a 'desc' option can be given to specify a name of the device. If this is omitted, the group name will be used.

To define a chain, you must list each device as shown above. Some tools might require a match of the device with the given ID.

The type field can be omitted for non Blackfin devices. It must be specified for Blackfin units in order to recognize the ID, as it is bit reversed.

4.3 Non emulation messaging (bfpeek)

The target side emulation library 'bfpeek' allows to exchange messages between host and target without stopping the CPU, that is, emulation mode is not entered. The JTAG interface works like an extra interface in this mode.

When gdbproxy is active and the target is running, it is automatically in 'peek mode', that means, it is listening to messages from the JTAG interface. The 'peek' handler inside the bfemu library automatically handles incoming packets via callbacks.

Currently supported from the Peek Module in bfemu are:

1. Standard input/output tunneling: Basic console emulation
2. File I/O: `fopen()`, `fread()`, `fwrite()` and `fclose()` support via remote protocol. A target can open a file relative to the working directory where gdbproxy was started from.
3. User defined data dumping functions such as image windows (not supplied with the standard ICEbear software)

All documentation concerning the bfpeek target library is found in the bfemu API documentation.

4.3.1 Printing debug messages

To enable your program to print debug output on the gdbproxy console or within the auxiliary TCP channel provided by gdbproxy, you have to do the following:

1. Use `bf_set_timeoutfunc(timeoutfunc)` to set your user defined timeout function
2. Use `bf_puts(fd, string)` or `bf_write(fd, data, count)` to send printable strings to the front end (gdbproxy). For `fd`, you must use one of [`MSG_DBG`, `MSG_STD`, `MSG_ERR`]

The message channels differ as follow:

MSG_DBG Sends the message to the gdbproxy console

MSG_STD Sends the message to the auxiliary TCP channel

MSG_ERR Sends the message to the internal error report function (normally, gdbproxy standard error)

4.3.2 Using standalone file I/O



Allowing file I/O to the target is a potential security risk. Do not run unknown programs on your target, malicious code may access your files.

To open a file on the target, you need to use the thin `syscalls.c` wrapper from the shell code, see Section 3.1.2. Then, you can use the standard `fopen` and `open` calls to open files for read and write access. Note that only very basic functionality is implemented, the files need to be created on the target before they can be accessed. You need to use at least the shell code v1.1. Within the `syscalls` wrapper environment, you can also redirect `printf` and `fprintf` outputs to the bfpeek channels. This works like on a standard POSIX system using the `freopen` function.



Avoid using standalone I/O from interrupts. The routines are not reentrant, so your debugger may hang when a I/O call is issued from within an interrupt while another I/O call was executed outside the interrupt routine.

Due to the packetizing of the C library, file I/O happens in chunks of normally 1024 bytes. This slows down the speed significantly. To increase the speed, modify your C library to use larger block sizes or use `write()` directly.

4.3.3 bfpeek I/O console on u-boot

For systems which are not supposed to output boot messages on the UARTs, the entire console code can be redirected to bfpeek channels. The console is then running on the auxiliary TCP server inside gdbproxy.

Installing libbfemu console wrapper

First, u-boot needs to be modified to use the bfpeek library. This is simply done by replacing the file `$UBOOT_DIR/cpu/blackfin/jtag-console.c` with a customized `uboot-bfpeek.c`. Then, the platform specific flags must be adapted to use libbfpeek (which is installed under `/usr/bfin-elf/` when installing the DEBIAN packages of libbfemu-dev)

```
PLATFORM_CPPFLAGS = -I/usr/bfin-elf/include
PLATFORM_LIBS = -L/usr/bfin-elf/lib -libfpeek
```

The flags can be set 'ad hoc' in `$UBOOT_DIR/config.mk` or in the platform specific rules files. Please read the u-boot source code for more information. For obtaining the appropriate wrapper files, please check the section5 forum or see <http://www.section5.ch/software>.

Using the bfpeek console

A short step by step description to connect to the u-boot bfpeek console:

1. Start gdbproxy, verify that the TCP server is reported running on port 4000 or defined via the `auxport` option.
2. Use nc or telnet to connect to the server:
nc localhost 4000
telnet localhost 4000
3. If u-boot is already residing in the flash: Skip this step.
Start `bfin-elf-gdb` with u-boot as argument and load it onto the target like the shell (note that you will need to go through the same initialization script procedure to properly initialize SDRAM).
> **bfin-elf-gdb u-boot**
(bfin-elf-gdb) **init**
(bfin-elf-gdb) **c**
4. If u-boot is in flash, you can simply reset the target with the reset button - do not disconnect from JTAG or powerdown. If you have no reset button, you can reset the target via gdb as follows:
> **bfin-elf-gdb -n**
(bfin-elf-gdb) **target remote :2000**
(bfin-elf-gdb) **monitor reset**
(bfin-elf-gdb) **c**
5. If u-boot is running correctly, you will see the boot messages on your telnet or nc console window.

Since there is no terminal emulation supported in u-boot, commands sent via telnet will possibly be executed twice, because an extra CR is sent. Disable this behaviour via `unset crlf` in telnet's command mode. To get into command mode, you'll have to use the escape character (normally `Ctrl+]`). Only line mode works well, char mode output is broken due to missing CR/LF from the u-boot console output.

With netcat (`nc`), the CRLF problem does not show up with the default settings.

4.4 Python API

New with the ICEbear plus software release is the Python API. This allows to remote control and test the target using Python scripts.

The modules included with the `libbfemu-python` package are self-documenting via the python documentation system.

A typical python script for hardware testing:

```
from bfemu import blackfin
from bfemu import loadelf
from bfemu.blackfin import Blackfin
import memtest

def init_ebiu(cpu):
    cpu.setMMR("EBIU_SDGCTL", 0x0091998d)
    cpu.setMMR("EBIU_SDBCTL", 0x0013)
    cpu.setMMR("EBIU_SDRRC", 0x03a0)
    cpu.setMMR("EBIU_AMGCTL", 0x00ff)
    cpu.setMMR("EBIU_AMBCTL0", 0xffc0)
    cpu.setMMR("EBIU_AMBCTL1", 0xffc0)

# Open the JTAG adapter with wait states = 3 for ICEbear plus
cpus = blackfin.open("0", 3)

# Determine CPU type and return CPU handle:
c = blackfin.determine(cpus[0])
c.reset(3) # Reset CPU
c.init_pll(20, 4) # Initialize PLL
init_ebiu(c) # Initialize the EBIU / SDRAM

# Now run the memory test:
size = memtest.test_pages(c)
print "Memory size: 0x%08x (%d MB)" % (size, size / 0x100000)

# Run block test. Automatically raises an exception on failure.
memtest.run_all(c, 20)
```

Technical Specifications

These specifications apply to hardware as listed under Section 1.2.

Supported Blackfin CPUs	BF527, BF53[1,2,3,4,6,7,(8),9], BF54x, BF561
USB compatibility	1.1, 2.0
Memory read speed	10kB/s (min), 80kB/s (typ), 130kB/s (peak)
Memory write speed	40kB/s (min), 120kB/s (typ), 240kB/s (peak)
Memory read speed ('plus')	Up to 500 kB/s
Memory write speed ('plus')	Up to 1 MB/s
Multi device	max. 4 Units per PC

Table 5.1: General features

Blackfin CPUs that are listed in brackets are supposed to be recognized, but not extensively tested.

Memory access times depend on OS and on used block sizes. Via GDB/Insight, the download speed can be significantly lower.

VCC Supply Voltage	5 V (USB bus powered), 6V maximum
Supply current	max. 100mA (typ. 25mA)
Operation Temperature Range	0 - 70°C

Table 5.2: Absolute maximum ratings

max. JTAG clock (TCLK)	6 MHz ('classic/light'), 30 MHz ('plus')
TCLK rise time	typ. 6ns
Output voltage levels	L: 0V H: 3.3V +- 5%
max. DC output current	20mA
Mean Time between JTAG failure	not measureable within 150 hours

Table 5.3: JTAG operation

The JTAG pinout of the adapter is shown in Fig. 5.1. Note the following:

- The UART RX and TX pins are only supported on the ICEbear Plus
- All GPIO pins are tristated by default. Make sure to not use software that drives them, when your target grounds these pins



Never use the ICEbear software with similar, modified JTAG adapters. It may damage your target.

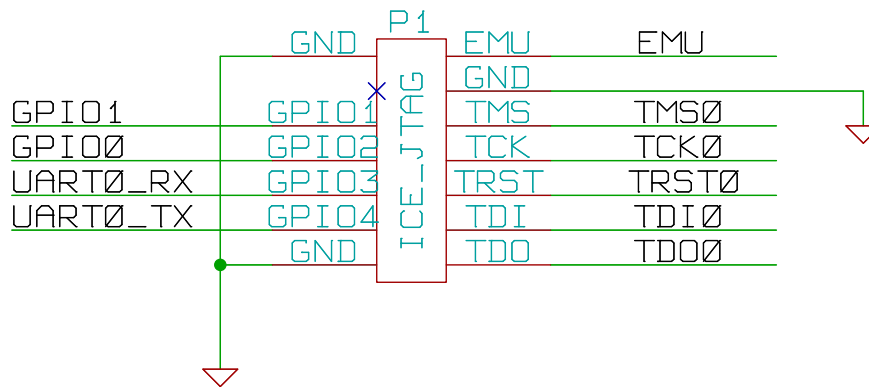


Figure 5.1: JTAG pinout, top view

Note the missing pin 3 for the key. The pinout is displayed mirrored, i.e. just as you normally see it when plugging the adapter into the target JTAG. The LED on the ICEbear plus models monitors, whether the JTAG device is accessed ('opened') by the Host PC. This is an auxiliary for factory programmers to determine whether an operation has completed and that it is safe to unplug from the target.

Troubleshooting

This is a collection of the most common errors that we could think of. If your question is not answered here, please contact section5, see Appendix C.

A.1 General

The ICEbear is not recognized

See Section A.3 for more information.

The software recognizes the ICEbear, but not the target board

Make sure the JTAG connector is properly attached and that the power is supplied to the board. If your board is not listed in the hardware support list (Section 1.2), make sure that the JTAG connector has the pinout and voltage specifications according to [bib_ee68]C . See also Section 1.2 for more information. If you can not solve the problem, consult us. On the ICEbear plus, the onboard LED is turned on once the target was opened for access.

My Debian installer shows a missing key warning when downloading the package.

See Section B.1.

A.2 gdbproxy errors

This section helps you to determine a connection problem according to the output of gdbproxy (>v0.8). When gdbproxy fails to recognize the the ICEbear, you will get the error message:
error: Could not open ICEbear connection. The following possible errors are explained in detail below:

error: Make sure that no other program/driver (ftdi_sio?) is claiming the port

You have a 2.6.X kernel: see section 'USB problems'. Are you sure there is no gdbproxy or other program running and grabbing the device?

error: Could not detect target, check connection and power

The ICEbear was recognized, but connection to the target could not be made (On the ICEbear plus, the onboard LED will light up for a short period). That means, that the target ID was not properly read. Please go through the following check list:

- Is your Blackfin CPU in the list of supported CPUs?
- Are you sure that there is no USB problem such as listed below (Section A.3)?
- Does your JTAG chain only contain the Blackfin device?
- Could there be issues with the JTAG frequency or extra cable lengths to the target? See also 'Target hardware issues' below. If the target is randomly detected, this case might be likely.

A few more possible errors:

Program download fails, GDB reports packet errors

Please make sure you are using the Insight version downloaded from the section5 website. Versions from blackfin.uclinux.org are not compatible with the section5 gdbproxy due to different register mapping. Please use the bfin insight package or compile from the source provided by section5.

Debugging does not work, no Blackfin registers are shown

If the **info register** command dumps register names like EAX, etc., then you have an intel variant of Insight running. Make sure, when compiling, that you specify the option `--target=bfin-elf`.

Download is very slow

When downloading the program to the target, it may take a long time when the packet size is configured too small. Verify with the `-debug` option passed to gdbproxy that the memory writes happen in reasonably sized packets (> 256). If this is not the case, you can change the remote packet sizes as follows within gdb or the startup script:

```
set remote memory-write-packet-size 1024
set remote memory-read-packet-size 1024
```

If this setting has no effect, caching for a specific memory region might inhibit gdb from sending the configured packet size. You may possibly want to disable 'mem' statements in your init scripts.

A.3 USB problems

Windows: The driver shows a yellow '!' mark in the Device Manager

Try to reinstall the driver by right-clicking on the device icon, choosing **Uninstall...**, and installing the FTDI driver again. Try running the FTDClean utility from the FTDI website (www.ftdichip.com) to clean up old driver relicts. If the problem persists, check whether the ICEbear works on another computer. If this fails, please contact us.



Make sure you always install the driver before plugging in the ICEbear!

Windows: The ICEbear is not recognized

Do you have any USB to serial port converter installed on the system? Try again with unplugging this converter and re-plugging the ICEbear, possibly to another port.

Windows: The ICEbear does not properly work anymore after interrupting a test program

When hitting Ctrl-C on a console program such as **memtest.exe**, it may happen on some systems, that the ICEbear can not be initialized again after. This is something we can not fix (except by catching Ctrl-C), as obviously the USB driver failed to free resources when the program was cancelled. Workaround: Unplug the ICEbear, wait a few seconds, and plug it back again.

Linux 2.4.X kernels: The ICEbear is not recognized

Try the following to recover from the problem

1. Log in as 'root' and type 'dmesg' after plugging in the ICEbear. If the following message does not appear, your kernel might not properly be configured for USB (missing usbdevfs module?). If you contact us, report your Linux distribution.

```
hub.c: new USB device 00:04.2-1, assigned address 4
usb.c: USB device 4 (vend/prod 0x403/0xc140) is not claimed by any active driver
```

Also, check /proc/bus/usb/devices. The above vendor/product number should be identified as from FTDI.

2. Make sure that no other software is using the device, for example another USB port serial driver.
3. Make sure you have the permissions to access the USB port in /proc/usb/<hub_number>. To grant a user the permission, use an entry like the one below in your /etc/fstab.

```
usbdevfs /proc/bus/usb usbdevfs defaults,devmode=0666 0 0
```

Linux 2.6.X kernels: ICEbear not recognized

Make sure you are member of the 'plugdev' group (you can verify this with the 'id' command on the bash console). If this is the case, try the following steps:

1. Type **dmesg** after plugging in the ICEbear. If it was properly recognized, the following message appears:

```
usb 2-1: new full speed USB device using uhci_hcd and address 2
usb 2-1: configuration #1 chosen from 1 choice
```

2. Look at /proc/bus/usb/devices (you can also use 'lsusb -v'). You should find an entry with the lines

```
P: Vendor=0403 ProdID=c140 Rev=5.00
S: Manufacturer=section5
S: Product=ICEbear JTAG adapter
```

The ICEbear 'light' shows 'c141' instead of 'c140', ICEbearPlus has 'c142'.

If you see the following, the firmware is broken. Contact section5 with your invoice number.

```
P: Vendor=0403 ProdID=6010 Rev=5.00
S: Manufacturer=FTDI
S: Product=Dual RS232
```

If you do not see any of these lines after plugging in the ICEbear, there might be something wrong with your USB connection. If the ICEbear neither works under Windows, a hardware error has to be assumed - please consult us before returning the ICEbear for check-up.

3. Check user permissions: The file /etc/fstab should contain a line like:

```
usb /proc/bus/usb usbfs defaults,devmode=0666 0 0
```

If in doubt, try running the ICEbear software as root to verify a permission problem.

Newer Debian or Ubuntu distributions include the new "udev" hotplug service. To make the ICEbear accessible for all users, copy the file udev/45-icebear.rules (see ICEbear distribution tar file) to /etc/udev/rules.d/ and restart the udev service (usually /etc/init.d/udev restart). The Debian/Ubuntu packages automatically install this file. To make the updated udev service recognizes the new permissions, the device must be reconnected physically.

The ICEbear is not found when plugging in a FTDI USB->Serial converter

- Try plugging in the ICEbear first, then the USB->Serial
- Use the `--dev` option with `gdbproxy` or `bfloader` to specify a device index



Multiple devices are only supported with one ICEbear variant at a time.

A.4 Target hardware issues



Generally, when connecting to custom hardware, failures can occur due to too long JTAG cabling. You must keep your JTAG connection between CPU and ICEbear as short as possible (< 7cm). If this can not be achieved, try decreasing the JTAG clock speed (see Section 2.1).

Emulation not ready

If the error message “Emulation not ready” (`ERR_NOTREADY`) appears, the system may have hung completely and can not be reset. Try the hard reset button or reconnecting the power. When doing that, make sure that any program driving the emulation is no longer running. Also, make sure that the hardware can not be in reset while the ICEbear software is trying to access the target. On some hardware, reset watchdogs may interfere.

SDRAM Memory test fails

If you get memory read or write failures when accessing SDRAM, this is most likely not an issue with the ICEbear adapter. There can be several reasons:

1. The SDRAM is not configured correctly, either by your embedded application or by your program using `bfemu`. If you are using GDB, read the notes about initialization in `[bib_bfemu]C`.
2. Your target hardware can not handle the clock speed of the *ICEbear*. Try lowering the clock frequency by using the `'-speed=<wait cycles>'` option for `gdbproxy` or `bfloader`. Normally, you should only need to use clock wait cycles from 0 to 4. See also `gdbproxy` options (Section 2.1).

Register does not read correctly

Try to repeat the procedure several times when the target system is halted. If you still get garbage values, the JTAG connection is probably instable. This should never happen, if your board is following the design rules in EE-68 (see Section 1.2). Check for externally connected hardware that may cause a problem. Also, there were very few PCs found with a “bad” power supply, which introduced spikes. Before assuming that the ICEbear is broken, you should always test on a different PC.

Target memory reading is much faster under Windows than under Linux

This is due to the USB driver implementation under Linux. Even if memory reading is implemented via an efficient command queue, it can not be accelerated further with the Linux driver.

Security

B.1 Debian Repository key

To register the section5 DEBIAN repository as trusted source and get rid of the warnings, you may want to fetch the public key from this location:

<http://www.section5.ch/section5-debian.gpg>

Key information:

Fingerprint: 0EE1 7E10 8392 6EAE 8550 90DD 4FCA 765D B426 CFC5

You can add this key to your APT trusted key ring using the command **apt-key add section5-debian.gpg**.

Literature and Links

For remarks or support, please use the order/feedback form at <http://www.section5.ch/order.php>.

C.1 Bibliography

A list of documents and further pointers:

- [bib_bfemu] **bfemu - The Blackfin emulation library**
08/2005, section5::ms <hackfin@section5.ch>
URL: <http://www.section5.ch/blackfin>
- [bib_ee68] **EE-68 JTAG Emulation Technical Reference Application Note**
Analog Devices
URL: <http://www.analog.com>
- [bib_bfinuclinux] **Blackfin uClinux The documentation Wiki for uClinux on the Blackfin**
ADI/uClinux
URL: <http://docs.blackfin.uclinux.org>