Freescale Semiconductor
Application Note

Document Number:AN4553
Rev. 0, 7/2012

Using Open Source Debugging Tools for Linux on i.MX Processors

Contents

1 Introduction

The aim of this document is to introduce a Linux software integrator i.MX platform to quick debug and diagnosis methods. It presents Linux generic concepts that can be easily and practically extrapolated from the concrete i.MX 6 examples to all i.MX products. The document is not intended to be exhaustive. Instead, the purpose of the document is to help with a fast setup of the appropriate debug tools. You will find that, even though other tools are also mentioned, most of the content involves the GNU Project Debugger (GDB) (building it, connecting to a debug server, debugging user space applications and the kernel, and using Eclipse as front end). This document does not discuss profiling.

2 General Debug Models and Mapping on GDB

In a *host - target* development model, where the host is usually a PC holding most of the software development tools and source code, and the target is the hardware or simulator for which the software is being developed, we can identify two debugging models:

- *cross debugging* - the analogue of the cross building of the application. When the target runs the debugged application under the control of a native debug agent

© 2012 Freescale Semiconductor, Inc.

and the host runs the heavy cross debugger that is aware of target architecture and does the symbols parsing, source mapping and variables work and communicate with the target using a protocol, usually over serial or Ethernet physical connections. Some of the advantages of this model are that the source code is not required on target, the debugged application can be stripped out of symbols when the target has limited amount of storage space, the native agent adds minimal overhead, and the debug session can be integrated in a visual front end. It is suitable for intensive debugging of large amount of code. When a special debug agent is available, the method can be also used without operating system running on target. The terminology mapping on GDB is simple: native agent = gdbserver, cross debugger = gdb

- *native debugging* - all in one solution, when the actions described above (debug information parsing, user interface and run control) are performed directly on target. The advantages are that there is no need for an additional connection with a host or for a cross debugger. Also, this is usually a more stable solution but it needs an operating system on target. The target must be powerful enough to run both debugging and the debugged application. This method is especially suitable for quick debugging of small problems and fast diagnosing. It can be eventually used without sources or debug information, for example to attach to a running process that is blocked in a deadlock and to inspect each thread's call stack. The terminology mapping on GDB: native agent = cross debugger = gdb (one single instance).

3 Requirements

This chapter will briefly introduce several common constraints of a debug environment. First, both the cross debugger and the debug agent should be aware of the target architecture, in our case, ARM. This means that the cross debugger should mainly be capable of understanding the target ABI and to disassemble the target code, while the debug agent should be able to perform run control functions (e.g. use of ptrace in Linux).

To enable source code debugging, the most important requirements are:

- The application should be built with debug information, usually by providing some flags to compiler and assembler. For the GNU toolchain, these are:

```
gcc: -g / -ggdb / -gdwarf-2
as:   -g / -gdwarf-2 or -gdwarf2
```

- It is recommended to build the targeted application code as little optimized as possible (e.g. gcc -O0). This will enable stepping coherent with the source code. You will find that some applications, like the Linux kernel, are written in such a manner that they cannot by compiled with zero optimizations. In that case, the disadvantage is that stepping into source code is not always very accurate and you may want to switch to assembly stepping or using breakpoints.
- And lastly, the availability of the source code (only on the machine that runs the debugger user interface).

Another requirement is that the application must not interfere with the debugger (run control) mechanisms such as hardware registers and events that should not be modified or caught by the application. In the Linux application world, there is little risk of breaking the agent capabilities. You should pay special attention, however, while debugging in supervisor mode (e.g. a kernel).

To correctly catch process events like loading a shared library or creation of a thread, the shared libraries that implement these common functionalities must be deployed on target un-stripped of symbols. For Linux userspace those are *ld.so, libthread-db.so, libpthread.so*, and for the particular case of i.MX/LTIB, to achieve that uncheck *strip options* on LTIB Target Image Generation menu.

4 Building the Latest GDB for i.MX Processors

Since GDB is an evolving project and the board support package may not always provide the newest GDB, it is usually a good idea to find the latest stable sources and to build the tool with the toolchain from BSP and for the particular target that it will execute on.

General Business Information

This section outlines the steps that should be performed for building both cross and native GDB in a LTIB based environment, including the full command listing. It will become apparent that there is nothing inately specific to LTIB so the concepts can be applied to any build system and easily adapted to other targeted architecture. Should you need only the native debugger, skip steps 4 and 5. Otherwise, perform all steps as shown below:

1. Download the tarball, take the latest version (now 7.4) from http://ftp.gnu.org/gnu/gdb and uncompress it in a folder where will be performed the following steps scope="external" format="html"/>
2. Have cross gcc in PATH

```
$ export PATH=$PATH:/opt/freescale/usr/local/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/bin/
```

3. Set the target

```
$ export TARGET=arm-fsl-linux-gnueabi
```

4. Build cross version
   - Configure --target=$TARGET --prefix=<install destination>
   - Build
   - Install

```
$ ./configure --target=$TARGET --prefix=/usr/local/gdb-7.4-arm-fsl-linux-gnueabi
$ make
$ sudo make install
```

5. Clean the build folder

```
$ make distclean
$ find . -name config.cache|xargs rm
```

   - or better remove the entire source folder and uncompress the tarball again

6. Build native version (including gdbserver)
   - Select target architecture and runtime (CFLAGS)
   - If missing, copy libtermcap in cross gcc runtime folder (one time operation per runtime)
   - Configure --build=$MACHTYPE --host=$TARGET --prefix=<install destination>
   - Build
   - Install

```
$ export CFLAGS="-march=armv7-a -mfpu=neon -mfloat-abi=softfp"
$ sudo cp <rootfs>/usr/lib/libtermcap.* /opt/freescale/usr/local/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gnueabi/arm-fsl-linux-gnueabi/multi-libs/armv7-a/neon/lib
$ ./configure --build=$MACHTYPE --host=$TARGET --prefix=<rootfs>/usr/local/gdb7.4
$ make
$ sudo chmod a+w <rootfs>/usr/local/
$ make install
```

Note: MACHTYPE is an environment variable that is usually present on a Linux host system (e.g. i686-pc-linux-gnu). When it's missing, set it accordingly.

5 First Steps for Using GDB for Linux User-Space Debug

5.1 Cross Mode

As described in Building the Latest GDB for i.MX Processors, the cross mode requires an agent running on target, gdbserver (step 6), and a cross debugger running on host (step 4). Steps 4 and 6 should have been completed as specified in the previous section. The supported connections are over a serial port or TCP. Choose the most suitable connection for your environment.

General Business Information

Keep in mind that the latter is usually a faster connection. Read the gdbserver command manual which will help you to see all parameter signification and will help you to start debugging an application or attach to an already running process. Then, start the cross GDB on host and connect with the agent. Perform the steps as shown in the example below:

```
Run the agent on target:
$ gdbserver :<port> <my_app_to_debug>


Run the debugger on host:
$ my_built_cross_gdb


In GDB, use commands like:
file <my_app_to_debug> - to specify the debugged file
set sysroot <target_root_fs> - for library discovery
(set solib-search-path and solib-absolute-prefix <target_root_fs>/lib - for gdb-6.6)
target remote <target-IP>:<port> - connect with the agent
```

5.2 Native Mode

Both debugger and application run on target. To perform source level debugging, the source code should be made available by copying on the target root filesystem or, preferably, by using a (shared) NFS folder. You may chose to start the target Linux using the NFS root filesystem, or you can mount the NFS folder later with the source code. Since the debugged application is built on a host (using a cross compiler), the debug information will contain the full path of sources on host which may be different from the sources path on target. If the sources are not available at debug, you will need to perform path mapping as exemplified below.

Run debugger on target. If GDB does not start properly, refer to Other Runtime Analysis Tools for Linux.

```
$ gdb <myapp>
```

Use GDB command "set substitute-path <build-path> <actual-path>" for path mapping

5.3 First GDB Commands

Use GDB commands as in the following example and learn the highlighted shortcuts:

```
break <symbol>
break <file>:<line>
continue/run (note the differences!)
next
list
info thread
thread <id>
backtrace
frame
```

6 Other Runtime Analysis Tools for Linux

Sometimes setting up a heavy debug infrastructure is not necessary for quickly identifying the root cause of certain problems. For example, in a "Segmentation fault" situation, first use a system or library call tracing the tool directly on target. The above built native gdb crashes at run. Execute it under strace and you will find that one of the latest syscalls was the attempt at opening a missing file. To solve this problem, create the empty file. For this kind of tools, check **<fsl-arm-toolchain>/ /**

General Business Information

debug-root/usr/bin folder for pre-built target tools or build them from tarballs. First check the BSP build system that you use since some tools are already available and should only be selected, like strace in LTIB. Eventually, copy those tools on the target root filesystem in **usr/bin** folder.

```
strace, strace-graph
ltrace
duma (Detect Unintended Memory Access)
dmalloc
```

7 Using ECLIPSE Front End for Linux User-Space Debug

When doing intensive debugging, especially of a multithreaded application, the command line interface may prove cumbersome. In this case, the user needs to inspect threads stacks, variable values per each thread frame, registers, disassembly, to build own expressions that the tool evaluates or to change memory, variables, and registers values. A graphic front end makes those easier. This section shows how to quickly start using one of the most modern and flexible IDEs, Integrated Development Environments, in which the community, including Freescale and other major players in the embedded applications space, is continuously contributing both for IDE and C/C++ development components. Follow the next steps to install and setup this development tool to work with the previously built GDB:

1. The GUI is java based, so first install JRE (> 1.5) and add it in the PATH. Download it from the following link:

   http://www.oracle.com/technetwork/java/javase/downloads/index.html

2. Install the latest ECLIPSE for C/C++ developers using the following link:

   http://www.eclipse.org/

3. Run the installed ECLIPSE and
   - Create a project:
     - Create a Makefile Project with Existing Code
     - Choose Cross GCC
     - Optional: Configure paths to build tool in Project Properties
   - Configure a debugger (and remote system):
     - Create a Debug Configuration of type C/C++ Remote Application
     - Use GDB/DSF Manual Remote Debugger Launcher
     - Specify cross GDB in Debugger/Main tab (refer to step 4 in Building the Latest GDB for i.MX Processors)
     - Specify a gdbinit file in Debugger/Main tab that contains: set sysroot <rootfs>
     - Configure target IP and gdbserver port in Debugger/Connection tab
   - Prepare the target for debugging and start debugging
     - Run gdbserver on target
     - Try start from entrypoint mode
     - Try attach to process mode

The steps above (Configure a debugger (and remote system), Prepare the target for debugging and start debugging) are not different from those used in Cross Mode to start debugging in cross mode, but this time the ECLIPSE UI has been instructed to perform some of the steps. In addition to exploring the debugger front end features of ECLIPSE, you may also want to use it as a powerful code editor, project manager, and more. Start by opening *Project->Properties*and configure both the cross toolchain and how the project should be built. After that, try to make ECLIPSE connect with your version control system and use its capabilities for performing remote repository operations.

General Business Information

8 First Steps for Using KGDB for Linux Kernel Debug

Why is my kernel is stuck in early boot phase? Why is my NAND memory not correctly detected even though I made the required kernel configuration? What is eventually going wrong in my loadable module? Those are the kinds of questions that may lead to using a debugger. But first, take the easiest approach. Go into your Linux kernel Documentation folder and open *kernel-parameters.txt*. This contains exhaustive documentation for generic parameters that can be provided to kernel at startup. When you search for *debug* keyword, you will find that there are quite a few options. Ttry the most useful option for your situation. For example:

```
initcall_debug
```

Not enough? KGDB is a kernel feature designed for on-source debugging of the Linux kernel. It is introduced here because, at this point, the reader should be familiar with the cross debugging concept and its particular GDB implementation. To make it simpler, KGDB can be regarded as another GDB agent, which resides this time in the Linux kernel. It's documented at:

http://kernel.org/pub/linux/kernel/people/jwessel/kdb/.

The debug model is similar with what has already been described and so are the steps to be performed on the host machine.

As was the case with the previously discussed agent, there should be a connection between the target system and the host, and this time we'll use the serial line (UART driver), feature known in KGDB as "over console" (*kgdboc*). The best thing to do is to have a dedicated serial line for it, but it's still usable when there is a single serial line available on the target system, also used as Linux console. The options are documented at the link above. The easiest (initial) test can be to disconnect the terminal program (e.g. *minicom*) before connecting with gdb.

When using UART, the debugger works in polling mode so the kernel driver should implement two callbacks in

```
struct uart_ops:
          .poll_get_char
          .poll_put_char
```

Those are located in *drivers/tty/serial/imx.c* file and have been contributed to the mainline kernel but may still not be in older kernel sources. In this case, you can manually merge the implementation of those functions from the mainline kernel tree to local source. Check the availability before going further.

Next, the kernel should be configured with kgdb support. Check the following in kernel configuration:

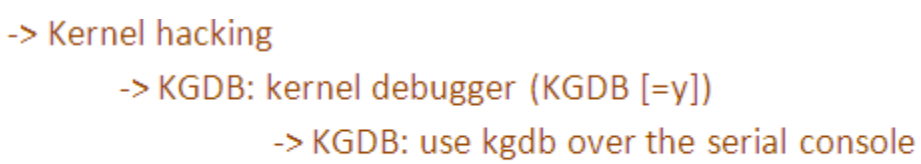

Figure 1. Kernel Configuration with kgdb 1

If you want to debug the kernel during boot, select the last entry above to be compiled in kernel (*) and not as module (M). Note that the latter also enables console poll.

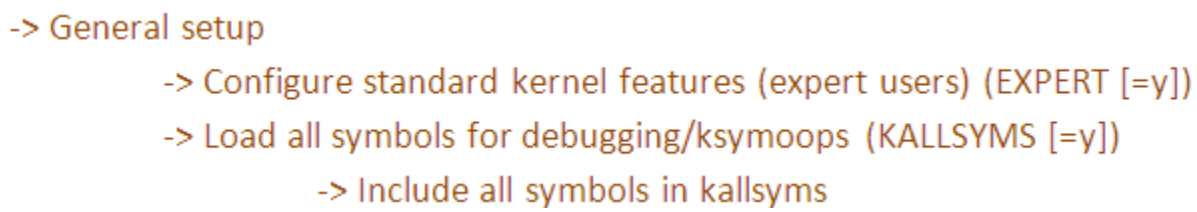

Figure 2. Kernel Configuration with kgdb 2

General Business Information

At run time, the *kgdboc* driver must be activated and instructed how to operate. It needs to know the serial line configuration for communication with the host debugger and also if it must stop the kernel boot process and wait for a debugger connection as soon as this driver is available. For that, there are two parameters to be provided from kernel command line:

```
kgdboc=<tty-device>,[baud] kgdbwait
```

While the meaning of the former is obvious, the latter is a kernel command line option that can be provided after the former to instruct the kernel to wait for a debugger connection as soon as the I/O driver has been activated. This being said, let's try a more practical approach for the following use cases such as debugging the kernel boot phase.

8.1 Debugging the Kernel Boot Phase

Add the following to kernel command line and start the boot process. Adjust the tty index according to your hardware (the following is used on SabreLite board with one serial connection available):

```
kgdboc=ttymxc1,115200 kgdbwait
```

Observe that the kernel boot process stopped with the following message:

```
kgdb: Registered I/O driver kgdboc.
kgdb: Waiting for connection from remote gdb...
```

(*) If you are using the same serial line for both console and debug, now is the time to disconnect the terminal application (e.g. *minicom*).

Then, start the host debugger using the vmlinux elf from your Linux build as input file:

```
$ cd ltib/rpm/BUILD/linux
$ my_built_cross_gdb vmlinux
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
```

That will initiate a debug session of the Linux kernel. Use the GDB commands which you have previously learned to inspect threads (mapped on Linux tasks), call stacks, variables, or to set/remove breakpoints. Note that, since the debugger works on polling (not interrupt), the host debugger is not capable of stopping the kernel execution on user request. You can use step (n command) or set breakpoints which, when hit, will stop the kernel and activate the debugger. There is a way to stop the kernel on request, for debug, and it will be discussed in Debugging the Kernel After Boot. Sysrq-g.

8.2 Debugging the Kernel After Boot. Sysrq-g

To configure and enable the *kgdb* connection after boot, use the following command on target. It is necessary only when the kernel did not receive the same configuration in its boot command line, as discussed above. A confirmation message will be printed out:

```
$ echo "ttymxc1,115200" > /sys/module/kgdboc/parameters/kgdboc
kgdb: Registered I/O driver kgdboc.
```

At any time, you can inspect the content of this virtual filesystem entry to check the driver configuration. It can also be disabled by writing an empty string into it:

```
$ echo > /sys/module/kgdboc/parameters/kgdboc
kgdb: Unregistered I/O driver kgdboc, debugger disabled.
```

After making sure that the feature is enabled, it's time to connect with the host debugger. As already discussed, when using *kgdboc* there is no way the user can trigger an interrupt from host debugger in order to enter a debug session, so a sysrq-g sequence should be initiated on target. It will be followed by a confirmation message:

General Business Information

```
$ echo g > /proc/sysrq-trigger
SysRq : DEBUG
Entering KGDB
```

Now the target waits for the host debugger to connect and the same steps from Debugging the kernel boot phase (*) should be followed.

8.3 Debugging the Kernel Loadable Modules

While the kernel body resides at well known virtual addresses (same as the load addresses from the *vmlinux* elf file), a loadable module (*.ko*) is just an archive, a collection of sections with no valid load address. The kernel loader allocates memory and decides where to load each loadable section of such a module. This means that, before module load (*modprobe*, *insmod*), we don't know where in the virtual memory it will stay, but we can find out immediately after the module load. The file *kernel/module.c* is the loader. Looking into this file, observe that some features are guarded by *CONFIG_KALLSYMS* definition. In order to export section load information to userspace, the kernel must be compiled with this macro enabled. This has already been done in First Steps for Using KGDB for Linux Kernel Debug, so per-section load information is available as follows:

```
/sys/module/<module-name>/sections/.<section-name>
```

Each file starting with the following characters: "." contains the virtual load address of a loadable section. Some of those are data sections (e.g. for variables), others are code sections. At the very least, use the values for *.text*, *.data*, *.rodata* and *.bss* to instruct the GDB debugger about the load addresses:

```
(gdb) add-symbol-file <filename.ko> <text_section_load_address> [-s .<SECT>
<SECT_LOAD_ADDRESS>]*
```

Example:

```
(gdb) add-symbol-file  ~ltib/rootfs/lib/modules/3.0.15-1359-g1b64ead/kernel/drivers/mxc/gpu-
viv/galcore.ko 0x7f000000 -s .data 0x7f01a250 -s .rodata 0x7f017b54 -s .bss 0x7f01a5b8
add symbol table from file "/home/agancev/mx6/ER5/minimal/ltib/rootfs/lib/modules/
3.0.15-1359-g1b64ead/kernel/drivers/mx
c/gpu-viv/galcore.ko" at
          .text_addr = 0x7f000000
          .data_addr = 0x7f01a250
          .rodata_addr = 0x7f017b54
          .bss_addr = 0x7f01a5b8
```

You should now be able to set breakpoints in the loadable module and debug it in the same session of a kernel debug.

Do you want to debug the module from its first function? It is more difficult, but not impossible. To do so, set a kernel breakpoint in the file mentioned above *kernel/module.c* at the place where the kernel calls the module's init function (*do_one_initcall(mod->init)*). At this point, all module sections have been loaded. Inspect kernel variables' values and try to find the load address for *.init.text* section and provide it to the debugger in the add-symbol-file command. You should then be able to set a breakpoint in the module's init function.

9 Using ECLIPSE Front End for Linux Kernel Debug

Similar to the process explained in Using ECLIPSE Front End for Linux User-Space Debug, an Eclipse project should be created for the Linux kernel. It is recommended to create a Makefile Project with Existing Code. Likewise, the next step is the configuration of a debugger (remote system), but this time we'll use the C/C++ GDB Hardware Debugging. This is not a default feature of the Eclipse C/C++ and should be installed as follows in Eclipse:

- Help -> Install New Software…
- Work with: Indigo http://www.download.eclipse.org/releases/indigo
- Mobile and Device Development-> C/C++ GDB Hardware Debugging

General Business Information

The configuration of the debugger should be made as follows:

- Create a Debug Configuration of type C/C++ GDB Hardware Debugging
- Select your Linux project and vmlinux C/C++ Application
- In Debugger tab:
  - Specify the cross GDB command (refer to step 4 in Building the Latest GDB for i.MX Processors)
  - Choose Generic Serial remote target and specify the host serial port in GDB Connection String edit box (e.g. /dev/ttyUSB0)
  - Select Using Standard GDB Hardware Debugging Launcher (at bottom of the page)
- In Startup tab, check ONLY Load Symbols from project *vmlinux*, leave others unchecked

Make the target wait for a remote connection (refer to Debugging the Kernel After Boot. Sysrq-g), disconnect the terminal program if required and hit Debug button.

To manually enter *GDB* commands from Eclipse, in Console window use the push-down button named *Display Selected Console* and choose the *GDB* console. Here you can type, for example, the *add-symbol-file* command for debugging a loadable module. Do not forget to type this command every time you restart *GDB*, otherwise source mapping will not be performed and the module will be debugged only on assembly code and its variables will not be available for inspection.

10 Conclusions

This document has provided a few ways to diagnose problems of software running on an embedded Linux system including the operating system and its drivers. They are all basic, well known techniques, and while some require only the configuration to enable comprehensive logging, others are more elaborate. There are many other debugging and logging methods that have not been discussed here. Profiling and optimizing have also not been the subject of this document. Regarding debugging techniques, one final observation is that open source and commercial tools use similar approaches for debugging Linux applications and, for the Linux kernel, on chip debuggers (e.g. JTAG) are usually more powerful and reliable.

General Business Information

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale and the Freescale logo are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited. ARM9 and ARM Cortex-A8 are the trademarks of ARM Limited.
© 2012 Freescale Semiconductor, Inc.

Document Number: AN4553
Rev. 0
07/2012