BA22-SDK

BA22 Command Line SDK

User's Guide

September 2011

IP Product Version
4.0p1

Document Signature
BA22-SDK–CUG–4.0p1–101

CAST, Inc.

CONFIDENTIAL

Document Version

This document with its associated Release Notes applies to the version(s) of the core specified on the cover. See the Release Notes for any updates and additional information not included here.

Document Version History

Version	Date	Person	Changes from Previous Version
1.00	18 July, 2011	P.D.	First draft and approval
1.01	16 Sept, 2011	N.Z.	Restructured and reviewed

CAST, Inc.
11 Stonewall Court, Woodcliff Lake, NJ 07677
Phone: +1 201-391-8300 Fax: +1 201-391-8694
info@cast-inc.com www.cast-inc.com

Table of Contents

1. Introduction

The BA22 Command Line SDK is a complete GNU toolchain and includes the gcc, gdb, newlib and many other utilities. The SDK is available for Linux and Cygwin.

The Cygwin environment can be freely installed from www.cygwin.com

Administrative privileges are not required to install the BA22 Command Line SDK in a Cygwin or Linux environment.

2. Supplied Software

The BA22 Command Line SDK is supplied in a single gzip archive named ba-elf-*VersionNr*.i686-*Version*.tar.gz; for instance ba-elf-r252771.i686-cygwin.tar.gz.

Typical contents of the Command Line SDK package:

./ba-elf- VersionNr.i686- Version/	Various GCC tools like: ba-elf-gcc, ba-elf-g++, ba-elf-objcopy, etc.; gcc libraries, header files, gcc manuals and other binaries necessary to run the entire SW flow.
./example/	Example projects together with compilation scripts (makefile)

3. Installation

It is recommended to install the SDK in the /home/<user_name> directory either on Cygwin or Linux. The SDK package can be extracted to another directory.

3.1 Cygwin SDK version

1) Start a bash shell
2) Copy the provided GZIP package of SW Tools to your Cygwin home directory (/home/<user_name>). In the Windows32 terminology, it's usually C:\cygwin\home\<user_name>\
3) Go to your Cygwin home directory:

   cd ~
4) Extract the package:

   tarxvfz ~/ba-elf-VersionNr.i686-cygwin.tar.gz

   The extraction will put the executable gcc at ~/beyond/ba-elf/bin
5) Set the Cygwin environment by adding the SDK tools locations to the PATH variable. It can be done by performing one of the following steps:
   a) Run the beyond.env script available in the $HOME directory:

      . ~/beyond.env
   b) Manually change the PATH variable:

      export PATH=~/beyond/ba-elf/bin:~/beyond/ba-elf/ba-elf/bin:$PATH
   c) Add the PATH modification to your Cygwin profile file (recommended). This will cause automatic setting of the BA22 SDK Tools with every launch of Cygwin bash. To do so, the ~/.bash_profile (or .~/.profile) file has to be modified using VI, EMACS or other text editor. The following line should be added at the end of the .profile file:

      export PATH=~/beyond/ba-elf/bin:~/beyond/ba-elf/ba-elf/bin:$PATH
   d) Add the beyond.env script to your Cygwin profile file (recommended). This will cause automatic setting of the BA22 SDK Tools including a short information of the tools usage with every launch of Cygwin bash To do so, the ~/.bash_profile (or .~/.profile) file has to be modified using VI, EMACS or other text editor. The following line should be added at the end of the .profile file:

      . ~/beyond.env

3.2 Linux SDK version

1) Start a bash shell
2) Copy the provided GZIP package of SDK tools to your Linux home directory (/home/<user_name>).
3) Go to your home directory:

   cd ~
4) Extract the package:

   tarxvfz ~/ba-elf-VersionNr.i686-linux.tar.gz

   The extraction will put the executable gcc at ~/opt/ba-elf-VersionNr.i686-linux/ba-elf/bin
5) Change the PATH variable either in the present shell or by modifying the ~/.bash_profile file:

   export PATH=$HOME/opt/ba-elf/bin:$PATH

After the successful installation, the user may want to check out the ba-elf-gcc availability and version. This can be done by:

whichba-elf-gcc

The expected response (assuming SDK was installed in $HOME) should be as follows:

Cygwin: *$HOME/beyond/ba-elf/bin/ba-elf-gcc*

Linux: *$HOME/opt/ba-elf/bin/ba-elf-gcc*

The gcc version can be checked by calling ba-elf-gcc as follows:

ba-elf-gcc -v

4. BA22 GNU Command Line Tools

The BA22 Command Line SDK contains gcc, binutils, gdb, basim, newlib and more. Some of the most popular tools included in the BA22 SDK are described below.

4.1 ba-elf-gcc- compiler

Syntax:

ba-elf-gcc<options>FileName.c compilation to an object

Output: *FileName.o*

Most common options used with compilation:

-c	compile or assemble the source file arguments, but do not link. The output is in the form of an object file for each source file. To compile and link all the sourcefiles, skip the '-c' switch
-Os:	optimize for size, e.g. *ba-elf-gcc -Os -march=ba2 FileName.c*
-O2	optimize for speed, e.g. *ba-elf-gcc -O2 -march=ba2 FileName.c*
-march=ba2	mandatory BA22 architecture definition
-g:	include symbolic information that enables the output file to be debugged with the gdb debugger
-o <file_name>	output file
-Wall	enables all the warnings about constructions that some users consider questionable, and that are easy to avoid.
-nostartfiles	no startup files included in the compilation. Note that the –nostartfiles switch only applies when the linking process is performed; "-c" switch is not used. The startfiles can perform various functions depending on the requirements. In the most common scenario, they set up the stack pointer, initialize memory controllers, set clock ratios, copy appropriate data from FLASH to RAM, etc. The sample of startfiles is listed in section

Less common options used with compilation:

-fomit-frame-pointer	don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. More information about frame pointers and call stack can be found at http://en.wikipedia.org/wiki/Call_stack

Sample usage:

ba-elf-gcc -c -march=ba2 FileName.c

More information about the *ba-elf-gcc* is available after typing (in Cygwin or Linux shell):

man ba-elf-gcc

4.2 ba-elf-size – displays object's size

Syntax:

ba-elf-size FileName.o display size of the compiled object

Output:

```
text    data    bss     dec     hex     filename
24779   0       744     25523   63b3    FileName.o
```

- text - size of the code instructions
- data - size of initialized data
- bss – size of uninitialized data
- dec – total program size (code and data)
- hex – total program size in hexadecimal

All the values above are given by *ba-elf-size* in decimal except the 'hex' info.

More information about the *ba-elf-size* is available after typing (in Cygwin or Linux shell):

ba-elf-size -H

4.3 ba-elf-objdump – displays object information

Syntax:

ba-elf-objdump<options>FileName.o display information from object

Most common options used with *ba-elf-objdump*:

-d display the assembler mnemonics for the machine instructions from object file. This option only disassembles those sections which are expected to contain instructions.

-h: display summary information from the section headers of the object file

-t display the contents of the symbol table(s)

-S: display source code intermixed with disassembly if possible. Implies -d

To redirect output to a file use:

>FileName.d

after the command.

Sample usage:

ba-elf-objdump –d FileName.o>FileName.d

More information about the *ba-elf-objdump* is available after typing (in Cygwin or Linux shell):

ba-elf-objdump -H

4.4 ba-elf-objcopy – converts format of object file

Syntax:

ba-elf-objcopy<options>FileName.oFileName.bin copy information from the object to another file in a specified format

Most common options used with *ba-elf-objcopy*:

-Obinary copies information from object using binary format

-v display information about performed copying

Sample usage:

ba-elf-objcopy –O binary –v FileName.o FileName.bin

More information about the *ba-elf-objdump* is available after typing (in Cygwin or Linux shell):

ba-elf-objcopy -H

4.5 ba-elf-ld – link objects

Syntax:

ba-elf-ld combine/link a number of object and archive files, relocates their data and ties up symbol references

Most common options used with*ba-elf-ld*:

-o <output_file> output file definition

-T <linker script> read linker script. More information about the linker script can be found in ram.ldsection. The linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that is compiled into the linker executable.

-verbose: display default linker script.

-m EMULATION set emulation (needed for debugging purposes). EMULATION should be set to ba2_elf

Sample usage:

ba-elf-ld-mba2_elf-T ram.ld -o OutFileName.ba2 reset.o hello.o

where:

OutFileName.ba2 – linked output file

reset.o – object file of a reset source file. This file could be a startfile.

hello.o – object file of a C source file.

More information about the *ba-elf-ld* is available after typing (in Cygwin or Linux shell):

ba-elf-ld-help

Another source of information: *http://www.kpitgnutools.com/manuals/ld.html*

4.6 ba-elf-nm

Syntax:

ba-elf-nm<options>FileName.o list symbols from object files

Most common options used with *ba-elf-nm*:

-l use debugging information to find a filename and line number for each symbol

-n sort symbols numerically by address

Sample usage:

ba-elf-nm –l –n FileName.o

More information about the *ba-elf-nm* is available after typing (in Cygwin or Linux shell):

ba-elf-nm -H

4.7 Next step to more GNU tools…

Note that the described tools don't make a complete list of the provided GNU tools. The full repository of the available tools can be found in the following directory:

$HOME/beyond/ba-elf-*VersionNr*.i686-*Version/bin*

When displaying information either by using *man, help or H* switches, it often happens that the output is longer than one screen. To read the information "screen by screen", the "more" pipe can be used. For instance:

manba-elf-gcc |more

To scroll to the next screen, press the 'space' key. To leave the information stream, press 'q'.

More information about GCC and GNU binutils can be found at the following websites:

http://gcc.gnu.org/

http://www.gnu.org/s/binutils/

5. BASIM - Architectural Simulator

The Architectural Simulator enables the user to monitor the execution of the written SW including interactive data transmission to the designed system.

Syntax:

ba-elf-sim.ba2<options><binary_file>

Most common options used with*ba-elf-sim.ba2*:

-i enable interactive command prompt

-f load script file [sim.cfg]

Sample usage:

ba-elf-sim.ba2 -i -f sim.cfg hello-uart.ba2

To display all possible commands available in the interactive mode, the “help” command should be used in the interactive mode. Some of the commands enable displaying the BA22 registers, memory, execution statistic or history while the others allow setting breakpoints or registers.

The most common commands used in the interactive mode are listed below:

-	t	execute next instruction
-	set {$r[register], *[address]} = [value]	set register or memory value
-	r	display contents of all General Purpose Registers
-	de address	disassembly memory; e.g. de 0x100
-	spr<name or value>	display contents of Special Purpose Register; e.g. spr SPR_SR or spr 17. This command can be also used to set value of SPR like spr 17 = 0x100.
-	x address	display memory at “address”
-	run <nr of instructions>	executes N consecutive instructions

In case of having a UART implemented in the system, the output of simulation will be shown on the screen and also stored in the “uart0.tx” file located in the current working directory. There is also a way to send data to the BA22 environment using the “uart0.rx” – also present in the current working directory.

For instance typing in a separate shell:

echo “test” >>uart.rx

would result in sending the “test” stream via UART.

Sometimes the source code includes a SW loop, like waiting for the incoming data or ever-lasting loop - while (1). To quit such loops, <CTRL + c> should be used.

Note that the breakpoints functionality is not implemented in BASIM. The breakpoints can be executed using ba-elf-gdb debugger or Eclipse IDE. For more details please contact CAST.

6. Project Files

Below is a list of files which make up the included sample project. Note that the sample project can be easily expanded by a user at the development phase.

- hello.c C project
- uart.h UART's registers definitions and functional macros
- spr_defs.h BA architecture specific special purpose registers definitions
- mc.h definitions used for Memory Controller simulation; it's used by Architectural Simulator
- board.h description of the simulation board (chip level)
- Makefile sample development script
- sim.cfg Architectural Simulator configuration script
- ram.ld linker script
- head.S sample exception handlers
- reset.S sample startfile

The sample project can be found in:

Cygwin: $HOME/beyond/example/hello-uart-ba2/

Linux: $HOME/opt/example/hello-uart-ba2/

6.1 makefile

The makefile included with the sample project automates the entire process of compilation, linking and copying the object to a binary file. Additionally, it creates a system map file (System.map) which shows all the symbols included in the object file together with their addresses.

A complete description of the Make utility can be found here:
http://www.gnu.org/software/make/manual/make.html

There are many more concise tutorials which quickly go over the Makefile syntax, e.g.:

http://www.opussoftware.com/tutorial/TutMakefile.htm

http://mrbook.org/tutorials/make/

6.2 sim.cfg

This file is part of the BASIM - BA Architectural Simulator included in the BA22 SDK. It contains the default configuration and help about configuring the simulator. The user can switch between configurations at startup by specifying the required configuration file with the -f <filename.cfg> option. If no configuration file is specified, the BASIM searches for the default configuration file `./sim.cfg'.

BASIM configuration file is split into the following sections (some sections optional and/or may not apply to your purchased product version, but are listed here for the sake of completeness):

- MEMORY : specifies how the memory is generated and the blocks it consist of
- IMMU : configures the Instruction Memory Management Unit
- DMMU :configures the Data Memory Management Unit
- IC : configures the Instruction Cache
- DC : configures the Data Cache
- PIC : specifies behavior of the Programmable Interrupt Controller
- TICK TIMER : configures Tick Timer
- SIM : specifies the BASIM's behavior
- CPU : specifies various CPU parameters
- PM : describes Power Management parameters
- DEBUG : defines the debug unit's behavior
- DBG_SOFT :configures the 'soft' debugger
- MC : configures the memory controller
- MC2 : specifies the memory controller v2
- UART : configures UARTs
- rs232_file : file endpoint for the rs232 bus
- rs232_fd : file descriptor endpoint for the rs232 bus
- rs232_tcp : TCP endpoint for the rs232 bus
- rs232_xterm : XTERM endpoint for the rs232 bus
- rs232_tty : TTY endpoint for the rs232 bus
- rs232_fd_log : rs232 traffic logger
- VGA : configures the VGA/CLD controller
- FB : specifies the frame buffer (VGA/LCD)
- KBD : configures the PS/2 compatible keyboard

Each section is described in detail in the sim.cfg file located in:

Cygwin: $HOME/beyond/example/hello-uart-ba2/

Linux: $HOME/opt/example/hello-uart-ba2/

6.3 ram.ld

The main purpose of the linker script ram.ldis to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file.

The main commands used in the ram.ld script are briefly described below..

6.3.1 MEMORY

The linker's default configuration permits allocation of all available memory. This can be overridden by using the MEMORY command.

The MEMORY command describes the location and size of blocks of memory in the target. It can be used to describe which memory regions may be used by the linker, and which memory regions it must avoid. Then, the sections have to be assigned to particular memory regions. The linker will set section addresses based on the memory regions, and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

A linker script may contain at most one use of the MEMORY command.

The syntax is as follows:

MEMORY

{

name [(attr)] : ORIGIN = origin, LENGTH = len

...

}

- The *name* is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script.
- The *attr* string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script.
- The *origin* is an numerical expression for the start address of the memory region
- The *len* is an expression for the size in bytes of the memory region

More information about the MEMORY command can be found at:

http://www.kpitgnutools.com/manuals/ld.html#SEC5

6.3.2 ENTRY

The first instruction to execute in a program is called the entry point. The ENTRY command in the linker script can be used set the entry point.

The argument is a symbol name: ENTRY(symbol)

More information about the ENTRY command can be found at:

http://www.kpitgnutools.com/manuals/ld.html#SEC10

6.3.3 SECTIONS

The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory.

The format of the SECTIONS command is: SECTIONS

{

sections-command

sections-command

...

}

Assuming that a program consists only of code, initialized data, and uninitialized data, the following sections have to be defined: `.text', `.data', and `.bss'.

For simplicity, these are the only sections which appear in input files.

In the example below, the code should be loaded at address 0x10000, and that the data should start at address 0x8000000. Here is a linker script which will do that:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

The first line inside the `SECTIONS' command of the above example sets the value of the special symbol `.', which is the location counter. If the address is not specified, the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the `SECTIONS' command, the location counter has the value `0'. In the example above, the location counter is set at 0x10000.

The second line defines an output section, `.text'. The colon is required syntax which may be ignored for now. Within the curly braces after the output section name, are listed the names of the input sections which should be placed into this output section. The `*' is a wildcard which matches any file name. The expression `*(.text)' means all `.text' input sections in all input files.

Since the location counter is `0x10000' when the output section `.text' is defined, the linker will set the address of the `.text' section in the output file to be `0x10000'.

The remaining lines define the `.data' and `.bss' sections in the output file. The linker will place the `.data' output section at address `0x8000000'. After the linker places the `.data' output section, the value of the location counter will be `0x8000000' plus the size of the `.data' output section. The effect is that the linker will place the `.bss' output section immediately after the `.data' output section in memory.

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary. In this example, the specified addresses for the `.text' and `.data' sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the `.data' and `.bss' sections.

An excerpt from the ram.ld linker script included in SDK:

```
MEMORY
{
        vectors : ORIGIN = 0x00000000, LENGTH = 0x00002000
        ….
}
SECTIONS
{
        .vectors :
        {
            *(.vectors)
        } > vectors

        …..
        …...
}
```

As it is shown above, the output section ‘.vectors’ will be placed after the location ‘vectors’ specified inside the MEMORY command.

More information about the SECTIONS command can be found at:

http://www.kpitgnutools.com/manuals/ld.html#SEC20

7. Sample development flow

The typical steps which could be taken with the SW development are described below.

All the used files are listed in Project Files section and can be found in $HOME/beyond/example/hello-uart-ba2/ directory.

1) Source files compilation:

   ba-elf-gcc –g –c –Os–march=ba2 –o hello.o hello.c

   Output:

   hello.o – object file

2) Startfile compilation:

   ba-elf-gcc –g –c –Os –o reset.o reset.S

   Output:

   reset.o – object file

3) Linking

   ba-elf-ld –T ram.ld –o hello-uart.ba2 reset.o hello.o –m ba2_elf

   Output:

   hello-uart.ba2 – linked object file with embedded debugging information

4) Retrieving the size info

   ba-elf-size hello-uart.ba2

5) Creating the disassembly file

   ba-elf-objdump –d hello-uart.ba2 > hello-uart.d

   Output:

   hello-uart.d – disassembly file

6) Copying the object file to binary

   ba-elf-objcopy –O binary hello-uart.ba2 hello-uart.bin

   Output:

   hello-uart.bin – binary file which can be instantiated into the uP memory module

7) Simulating in BASIM

ba-elf-sim.ba2 –f sim.cfg hello-uart.ba2

Note that the steps 1-6 (except #4) are automated by using the included makefile.

To execute the makefile the make “m*ake ba2”*command should be used. It is assumed that the make utility is installed in the Cygwin implementation; it’s available in Linux by default.

8. Support

Every effort has been made to ensure that this core functions correctly. If a problem is encountered, contact:

CAST, Inc.

11 Stonewall Court

Woodcliff Lake, New Jersey 07677 USA

Technical Support Hotline: +1-201-391-8300 ext. 2

Fax: +1-201-833-2682

E-mail: support@cast-inc.com e

URL: www.cast-inc.com