CCore on BeagleBone Black

Sergey Strukov 2014, version 1.00

1. Introduction.

Since the version 1.08 **CCore** includes the target for the popular board BeagleBone Black. You can use this target to build applications running on this board. See the site http://beagleboard.org/black for a comprehensive source of information about this hardware. The board is a small card with low power consumption (it may run on USB-supplied power), but with rich hardware capabilities. In fact, it may be used as a fully-featured PC, been connected with the monitor, keyboard and mouse.

To start working with the board unpack it and simply connect to the USB port on the host PC. The board is shipped with the pre-installed Linux distribution. Once connected the Linux kernel is started and the board is recognized by PC as the flash drive. The on-board flash has two partitions. The first one is formatted as a FAT drive and contains two-stage bootloader. This partition is seen over USB, so you can change its files from PC. The bootloader consists of the three files: **MLO**, **u-boot.img** and **uEnv.txt** files. The first file is the first stage bootloader. It starts first after the power on. Then it boots the second-stage bootloader **u-boot.img**, which is the u-Boot bootloader (see the product site http://www.denx.de/wiki/U-Boot). The second-stage bootloader finally boots the Linux kernel. But you can alter its behavior by changing the file **uEnv.txt**. This file contains a script, executed by the bootloader. You may change it to instruct the bootloader to download and run an image using the TFTP protocol from the host PC. Here is the script to do so:

```
uenvcmd=setenv ipaddr 192.168.99.10;setenv serverip 192.168.99.1;tftpboot 81000000 bootfile;go 81000000
```

The first IP address is the address of the board itself. The second is the IP address of the TFTP server. **bootfile** is the name of the file from the server to be downloaded. You must connect the board with PC by the Ethernet cable (directly or using switch). Configure the PC Ethernet card with the proper IP address manually (you may choose any other suitable address). You must run the TFTP server on the host PC. Then reapply a power to the board. It will download and run the image from the host PC using the TFTP protocol.

You may also boot the board from a standalone SD memory card. All you need is a memory card, this card must contain an active FAT partition with these three files in the root directory. Insert the card into the card pocket on the device and turn the power on.

The most convenient way to run a raw image from an MMC card is to use the following script:

```
uenvcmd=fatload mmc ${mmcdev}:1 81000000 /bootfile;go 81000000
```

This script loads the file **bootfile** in the root directory from the first partition of the boot MMC device to the address 81000000h, then starts execution from the address 81000000h.

2. Building a CCore application.

To build a **CCore** application you need a **cygwin** (or **cygwin64**), a toolset and a **CCore** installation. You can build the toolset yourself. Read the **CCore** supplied documentation about the building process and environment.

Here is the target files layout:

```
CCore
  ...
obj
  ...
sysroot
```

```
  lib
    ...
  obj
    ...
  src
    ...
  usr
    ...
  Makefile
test
  ...
test-obj
  ...
tools
  Boot
    ...
  BootServer
    ...
  ELFtoUboot
    ...
  HowToBuild
    ...
```

Two important directories are: **sysroot** and **tools**. The first is the **sysroot** part of the **gcc** cross-compiler. It is required to build the cross-compiler and to link applications. Once you have the cross-compiler you must build libraries **libc.a** and **libm.a** and the startup file **crt0.o**. The **Makefile** in this directory is doing it. You may need to alter it to give proper paths to the cross-compiler and tools. The **tools** contains tree important tools. You must build them to use **CCore**. Use the **Makefile** in this directory to build. **Boot** is the boot utility, it can boot ELF image over an Ethernet link to the board once the board runs **BootServer**. **ELFtoUboot** is a converter, it takes an ELF image and creates a raw file, this file can be run on the board using a TFTP server.

The following is the step-by-step description of the **CCore** deployment on **Windows64** and **cygwin64**.

First you need the following hardware: host PC, BeagleBone Black board, Ethernet link between the host PC and the board. Configure the Ethernet card on PC, connected with the board to the IP address 192.168.99.1 and the IP subnet mask 255.255.255.0. Then plug the board to the USB connector on the PC. Wait until the Linux is running (you will see flashing board lights). PC will detect the new USB memory storage device. Go to this drive, find the file **uEnv.txt** and edit it as shown above. Reapply power to the board. It will try to download and run an image from the host PC using the TFTP protocol. If you wish to start the Linux again, you have to download and run the image with the single command

```
    mov pc, lr
```

You may prepare such image using assembler, linker and **ELFtoUboot** converter. Just remember, that the image base and the entry point is `8100'0000h`.

The next step is to configure and run a TFTP server on the host PC. You may use the

following one http://tftpd32.jounin.net . Once it is done, you is able to download and run **CCore** images. The board requests the file `bootfile` from the server. Such files are created from ELF images by the utility **ELFtoUboot**.

The next step is the installation of the **CCore** itself. But first you need to install **cygwin64**. Go to the product site https://cygwin.com for the software. You must include **make**, **bash** and **gcc** in your installation. You also need **libgmp-devel** component. Once you have the **cygwin** installed, go to the home directory and create the subdirectory **bin**. Include it to the **cygwin** search path. This place is used to store host utilities. Finally, obtain and unpack the **CCore** release. Do it under your home directory, avoid file and directory names with the space. Go to the **CCore** root directory. Change the host target name to the **WIN64**:

```
CCORE_TARGET = WIN32
  change to
CCORE_TARGET = WIN64
```

You are ready to build the host **CCore** library and tools. Just issue the command **make** in the **CCore** root directory from the **cygwin64** console (not the **Windows** console).

The next step is to build the cross-toolchain. The simplest way is to get it from the **github** repository https://github.com/SergeyStrukov/CCoreOpt-Beagle. Or you can build it yourself, but the process is tedious. You will need the subdirectory **sysroot** during the build process. You may also find a build brief in the subdirectory **tools/HowToBuild**.

Once you have the cross-toolchain ready, you can build the target **CCore** library and tools. Go to the target directory **CCORE_ROOT/Target/BeagleBoneBlack**. Check the tool paths in the file **Makefile.tools**. To build the **CCore** just issue the command **make**. To build tools go to the subdirectory **tools** and issue the command **make** in each tool subdirectory. You are done.

When everything is ready you can build and run **CCore** applications on the board. There are two ways to run a built application. The result of the linker stage is an ELF file. You can convert it to a raw image and download and run using a TFTP server as described above. The utility **ELFtoUboot** should be called as following:

```
ELFtoUboot.exe <ELF-file> <boot-file>
```

Or you can run the PTP boot server on the board. This server is built in the directory **tools/BootServer**. You can run it using the first method. Then you can use the boot client (**tools/Boot**) to boot an ELF file:

```
BeagleBoot.exe r 192.168.99.10 <ELF-file>
```

Both **CCore-ELFtoUboot.exe** and **CCore-BeagleBoot.exe** are created in your **HOME/bin** directory. The IP address is hardcoded in the boot server, so if you wish to use another one, change **tools/BootServer/main.cpp** and rebuild the boot server.

This target starts the user code not from the function **main()**, as usual, but from the function **before_main()**. You may use this function to setup a proper hardware environment and then call the function **main()**. See the file **test/main.cpp** for an example.

3. Hardware resource utilization.

A **CCore** application is running in a privilege non-secure mode, so you have a full access to any board hardware. Briefly after start it turns on the MMU and caches, so you have a maximum CPU performance. It sets also the CPU clock to 600 MHz. It is possible to increase the clock up to

1 GHz, but it requires the reprogramming the power supply device to increase the CPU core voltage and is not implemented in this version. MMU is programmed to map a virtual address to the physical but such a way, that the physical address is always equal to the virtual. It also specifies memory cache attributes to various memory regions.

The board has 512 MByte DDR memory, this memory is visible at the address range `0x8000'0000 – 0xA000'0000`. A **CCore** application image is loaded and started from the address `0x8100'0000`. The memory is divided on several memory regions with different usage purpose. The first region `0x8000'0000–0x8080'0000` is reserved for the video-memory. It has no-cache attributes. The next region `0x8080'0000–0x8100'0000` is reserved for the shared memory heap. It also has no-cache attributes. This memory is used to exchange data with peripheral devices without a manual memory cache control. For example, Ethernet packet descriptors and data buffers are resides in this memory region. Right after this region an application image begins. It's assumed to be several MBytes length. After the image (which contains the code and static variables) the free memory zone begins. This zone is used to allocate a space for the prime task stack (1 MByte), the interrupt stack (1 MByte), the interrupt heap (1 MByte), the system log (16 MBytes) and the heap (204 MBytes). All these memory regions have the full cache attributes. If the application image length is less than 16 MBytes, only the first 256 MBytes half of the memory is used.

The following on-chip devices are reserved for the system usage and should not be touched by any other software entities: **GPIO1**, **DMTimer2**, **DMTimer4**, **WDTimer**, **INTC** (interrupt controller). You may use only functions, provided by the target to work with these devices.

The following on-chip devices must be used under the **ControlMutex** protection: **PRMC** and **Control Module**. The mutex is declared in the file **dev/DevControlMutex.h**.

```
namespace CCore {
namespace Dev {

/* global ControlMutex */

extern Mutex ControlMutex;

} // namespace Dev
} // namespace CCore
```

4. Hardware support classes.

The target contains several classes to work with peripheral devices.

4.1 RegRW – read/write registers (dev/DevRW.h).

The class **RegRW** should be used to read/write registers. It can be used along or with **Regen**-generated bar-classes.

```
namespace CCore {
namespace Dev {
```

```
/* types */

using AddressType = uint32 ;

/* classes */

class RegRW;

/* class RegRW */

class RegRW
 {
  public:

   using AddressType = uint32 ;

  private:

   AddressType base_address;

  public:

   explicit RegRW(AddressType base_address);

   template <class UInt>
   UInt get(AddressType address);

   template <class UInt>
   void set(AddressType address,UInt value);
 };

} // namespace Dev
} // namespace CCore
```

Constructor takes a base address of a register bar as the argument. Then you can use methods **set()** and **get()** to read or write registers. The template parameter must be one of **uint8**, **uint16**, or **uint32**. The class should be used to avoid register operations reordering.

4.2 Interrupt handling (dev/DevIntHandle.h).

The following family of functions can be used to setup an interrupt handler for a particular source.

```
namespace CCore {
namespace Dev {
```

```
/* enum IntSource */

enum IntSource
 {
  Int_LCDCINT = 36,
  Int_I2C0INT = 70,
  Int_I2C1INT = 71,
  Int_I2C2INT = 30,
  Int_TINT4   = 92,

  Int_3PGSWRXTHR0 = 40,
  Int_3PGSWRXINT0 = 41,
  Int_3PGSWTXINT0 = 42,
  Int_3PGSWMISC0  = 43,

  Int_GPIO1_0 = 98,
  Int_GPIO1_1 = 99,

  Int_TableLen = 128
 };

/* functions */

void SetupIntHandler(IntSource int_source,
                     Function<void (void)> handle_int,
                     unsigned priority=0); // priority 0-63

void CleanupIntHandler(IntSource int_source);

void EnableInt(IntSource int_source);

void DisableInt(IntSource int_source);

} // namespace Dev
} // namespace CCore
```

SetupIntHandler() connects the given interrupt handler with the given interrupt source. The third argument is the priority, if several interrupts are active simultaneously, then the more prioritized handler calls first. The highest priority is 0. After call of this function interrupts from the source are enabled.

CleanupIntHandler() disconnects the interrupt handler from the given source.

EnableInt() enables the interrupt from the given source.

DisableInt() disables the interrupt from the given source.

IntSource is an enumeration which designates interrupt sources. It is not the full list of all available interrupts on this CPU. For the full list read the chip documentation.

An interrupt handler is a normal function. This function is called in the interrupt context. So there are limitations on what the function can do. In particular, it must be quick and cannot call any blocking functions. It cannot use the default heap or print to the console, but can use the interrupt heap. For synchronization objects it must call only methods, allowed in the interrupt context, usually such methods have names, ended with the "**_int**" suffix.

4.3 Lights control (dev/DevLight.h).

The are four lights on the board. You can control them by the following set of functions.

```
namespace CCore {
namespace Dev {

/* functions */

void LightSet(unsigned mask);    // 4 lsb

void LightOn(unsigned mask);     // 4 lsb

void LightOff(unsigned mask);    // 4 lsb

void LightToggle(unsigned mask); // 4 lsb

} // namespace Dev
} // namespace CCore
```

LightSet() sets the lights according the given mask.

LightOn() turns on the lights according the given mask. I.e. if some bit is not set, the correspondent light remains unchanged. Other are lighted.

LightOff() turns off the lights according the given mask. I.e. if some bit is not set, the correspondent light remains unchanged. Other are darken.

LightToggle() toggles the lights according the given mask. I.e. if some bit is set, the correspondent light changes, otherwise it remains unchanged.

A mask has 4 lowest bits significant.

4.4 Performance counters (dev/DevPerfCount.h).

There is a special device to perform various performance counting. It consists of the block of hardware counters, these counters can be connected to different events. It also counts CPU clocks.

```
namespace CCore {
```

```
namespace Dev {

/* enum PerfEvent */

enum PerfEvent
 {
  PerfEvent_DataRead       = 0x06,
  PerfEvent_DataWrite      = 0x07,
  PerfEvent_CmdExec        = 0x08,
  PerfEvent_ExceptionTaken = 0x09,
  PerfEvent_L2Cache        = 0x43,
  PerfEvent_L2CacheMiss    = 0x44,
  PerfEvent_AXIRead        = 0x45,
  PerfEvent_AXIWrite       = 0x46
 };

/* classes */

class PerfCount;

/* class PerfCount */

class PerfCount : InstanceLock<PerfCount> , public Funchor
 {
  public:

   PerfCount();

   ~PerfCount();

   // methods

   unsigned getNumberOfCounters() const;

   void prepare(unsigned index,PerfEvent event);

   void start_mask(bool clock,uint32 mask);

   void stop_mask(bool clock,uint32 mask);

   template <class ... TT>
   void start(bool clock,TT ... indexes);

   template <class ... TT>
   void stop(bool clock,TT ... indexes);
```

```
   uint64 get(unsigned index);

   uint64 get_clock();
 };

} // namespace Dev
} // namespace CCore
```

To use this device you create the instance of the class **PerfCount**. You may create only one object of this type.

getNumberOfCounters() returns the number of available event counters.

prepare() connects the specified event with the counter with the given index. The counter is stopped and erased by this operation.

start_mask() starts the event counting on the given set of counters and on the CPU clock counter. The CPU clock is started, if the argument **clock** is **true**. The set of counters is given by the **mask**. The bit of number **i** in the **mask** corresponds to the counter of the index **i**.

stop_mask() stops the event counting on the given set of counters and on the CPU clock counter.

start() starts the event counting on the given set of counters and on the CPU clock counter. The set is given by the index list.

stop() stops the event counting on the given set of counters and on the CPU clock counter. The set is given by the index list.

get() returns the value of the counter with the given index.

get_clock() returns the CPU clock counter value.

The enumeration **PerfEvent** designates different events to be count.

DataRead – read data by the CPU core.

DataWrite – write data by the CPU core.

CmdExec – command is executed by the CPU core.

ExceptionTaken – exception is taken by the CPU core.

L2Cache – L2 cache access.

L2CacheMiss – L2 cache miss.

AXIRead – AXI data read.

AXIWrite – AXI data write.

4.5 I2C controller (I2CDevice.h).

The chip have three I2C controllers. The first is connected with some on-board devices and used to control them. Other may be connected to expansion devices using the expansion connectors.

The class **I2CDevice** can be used to perform read or write transactions on I2C bus.

```
namespace CCore {
```

```
/* class I2CDevice */

class I2CDevice : public ObjBase
 {
   ....

  public:

   explicit I2CDevice(Dev::I2CInstance instance);

   virtual ~I2CDevice();

   // methods , may throw exception

   void read(uint8 slave_address,PtrLen<uint8> buf);

   void write(uint8 slave_address,PtrLen<const uint8> buf);

   void exchange(uint8 slave_address,PtrLen<const uint8> out_buf,
                                     PtrLen<uint8> in_buf);
 };

} // namespace CCore
```

Constructor takes the argument of type **Dev::I2CInstance**, which is the enumeration:

```
enum I2CInstance
 {
  I2C_0,
  I2C_1,
  I2C_2,

  I2C_InstanceCount
 };
```

You may create only one object per one hardware instance.

The method **read()** reads a byte range from the given slave address.

The method **write()** writes the given byte range to the given slave address.

The method **exchange()** performs write with the following read operation. It is often used to read a register from a device with addressable registers on I2C bus.

The address is 7-bit and the byte count must not exceed 64 KBytes.

These methods are “slow”, they may block for a long period of time until the bus transaction is finished. They are also multi-task safe, that means the execution is serialized using the internal mutex. They may throw an exception in case of error.

Remember, that instances 1 and 2 need to be connected to the proper pads using the **Control Module** subsystem. The correspondent code in the file **DevI2C.cpp** is left blank. If you want to use these instances with some expansion hardware add the required code to this file in the method **DevI2C::enable()**.

4.6 Video device (video/VideoControl.h).

You may stream a video to the monitor using the class **VideoControl**.

```
namespace CCore {
namespace Video {

class VideoControl : public ObjBase , public VideoDevice
 {
  ....

  public:

   VideoControl(StrLen i2c_dev_name,
                TaskPriority priority=DefaultTaskPriority,
                ulen stack_len=DefaultStackLen);

   virtual ~VideoControl();

   void stopOnExit() { stop_on_exit=true; }

   ....
 };

} // namespace Video
} // namespace CCore
```

Constructors arguments are: I2C device name, task priority and task stack length. If you call the method **stopOnExit()**, the streaming will be stopped by the destructor. Otherwise it is retained. The class implements the interface **Video::VideoDevice**. The board video subsystem can stream a video up to the 1024x768 pixels in 32, 24 and 16 bpp formats from the video-memory. You may create only one object of this type.

Here is an example how to setup a console output:

```
I2CDevice i2c(Dev::I2C_0);
          // create I2C device on I2C-0 instance

ObjMaster i2c_master(i2c,"i2c[0]");
          // register I2C device under the name “i2c[0]”

Video::VideoControl vctrl("i2c[0]");
```

```
                    // create VideoControl device

ObjMaster vctrl_master(vctrl,"video");
          // register VideoControl device under the name “video”

Video::VideoConsole vcon("video");
                    // create VideoConsole device on “video”

vcon.waitOpen();
// wait for monitor plug

SingleMaster<Video::VideoConsole> vcon_master(Sys::GetConHost(),
                                              "!VideoConsoleMaster",
                                              vcon);
                                  // direct the console output to vcon
```

4.7 Ethernet controller (dev/DevEth.h).

To use the on-chip Ethernet controller you may use the class **EthDevice**.

```
namespace CCore {
namespace Dev {

....

class EthDevice : public ObjBase , public Net::EthDevice
 {
  ....

  public:

   EthDevice(ulen rx_count=50,ulen tx_count=50);

   virtual ~EthDevice();

   ....

   // start/stop

   void startTask(TaskPriority priority,ulen stack_len=DefaultStackLen);

   void stopTask();

   class StartStop : NoCopy
    {
```

```
      EthDevice &obj;

     public:

      template <class ... TT>
      StartStop(EthDevice &obj,TT ... tt);

      ~StartStop();
    };
 };

} // namespace Dev
} // namespace CCore
```

This class implements the **Net::EthDevice** interface and can be used with other **CCore** network classes. The constructor arguments specifies the number of Rx and Tx buffers. You may create only one object of this type.

Here is an example how to setup a network support:

```
Dev::EthDevice eth;
               // create Ethernet device

ObjMaster eth_master(eth,"eth");
          // register Ethernet device under the name “eth”

Net::NanoIPDevice ip("eth",Net::IPAddress(192,168,99,10),
                           Net::IPAddress(255,255,255,0));
                 // create IP device, assign IP address and network mask

Dev::EthDevice::StartStop start_stop(eth,TaskPriority(5000));
                          // start Ethernet task

ObjMaster ip_master(ip,"ip");
          // register IP device under the name “ip”

Net::NanoUDPEndpointDevice udp_ptp("ip",Net::PTPClientUDPort,
                                        true,
                                        Net::UDPoint(192,168,99,1,
                                        Net::PTPServerUDPort));
                           // create UDP endpoint device on “ip”
                           // on PTP ports to work with host

ObjMaster udp_ptp_master(udp_ptp,"udp_ptp");
          // register UDP device under the name “udp_ptp”
```

```
Net::PTP::ClientDevice ptp("udp_ptp");
                       // create PTP client device on “udp_ptp”

ptp.support_guarded(10_sec);
// initial exchange with the server

ObjMaster ptp_master(ptp,"ptp");
          // register PTP client device under the name “ptp”

Net::HFS::ClientDevice hfs("ptp");
                       // create HFS device on “ptp”

ObjMaster hfs_master(hfs,"hfs");
          // register HFS device under the name “hfs”

Net::HFS::FileSystemDevice host("hfs");
                           // create HFS FileSystem device on “hfs”

ObjMaster host_master(host,"host");
          // register HFS FileSystem device under the name “host”

Net::PTPCon::ClientDevice ptpcon("ptp");
                          // create PTP Console device on “ptp”

ObjMaster ptpcon_master(ptpcon,"ptpcon");
          // register PTP Console device under the name “ptpcon”

Net::PTPCon::Cfg cfg(Net::PTPCon::TriggerAll);
                 // immediate input keys readiness

RedirectPTPCon redirect("ptpcon","Beagle",cfg);
               // open and redirect PTP console
```