Demo code:
In the MSCC environment

for release 0.1
1. Februar 2017

Table of Contents

1 Adding module to the MSCC application

This document is meant as a guide to implementing a module in the MSCC application environment. The module we shall implement is called *demo*, and the purpose it to learn how to hook a new module, here demo, into MSCC application.

1.1 Preparation

This module shall be placed under `vtss_appl`, i.e., `vtss_appl/demo/`

Goto the `build` directory. If you have not already setup a link `config.mk`, then run `make` which will provide you with a list of posibilities. Choose one, e.g. `ln -s configs/ce_switch_caracal1_l10_ref.mk config.mk`.

I suggest you build the MSCC application before doing any changed. You will in that case have files that you can practise on in the two section below, where it is explaind howto do a ramload and howto use JTAG/GDB.

To make life easy, generate tags, i.e., `make TAGS`. Then you have tags for the emacs editor. That will be usefull in the following so that you do not have to find files explicit, but instead can search for a specific name that will get you to the right location.

1.2 Ramload

In order to test the code that we are going to build in the following, it is recommended to use ramload. This means that the code is only uploaded to the ram and therefore does not make changed to the device.

When the code has been compiled, the result is in `build/obj` as either a `.dat` file for eCos or a `.mfi` for linux. This file is ramloaded to the device in the following way (in case of eCos)

```
# platform debug allow
# debug firmware ramload tftp://a.b.c.d/some/path/xxx.dat
```

When the target reboot press `Ctrl-C` in redboot and follow the instruction that redboot print. That usually means you have to give the command `ramload` and then wait for the target to boot.

1.3 JTAG and GDB

For eCos an `.elf`, which is located in the same directory as `.dat`, can be loaded with GDB. The figure below illustrate the setup we will use:

```
         JTAG/RS232          USB         Network/loop
           |                  |              |
+--------+ V  +-------------+ V  +---------+ V   +---------------------+
| Target |----| Flyswatter2 |--+-| OpenOCD |--+--| Machine running GDB |
+--------+    +-------------+  | +---------+  |  +---------------------+
                               |              |
                          +---------+         |  +------------------------+
                          | Minicom |         +--| Machine running telnet |
                          +---------+            +------------------------+
```

1.3.1 Install Flyswatter2

The Flyswatter2[1] is connected to the target with a ARM20MIPS14 cable, and to the PC running OpenOCD with an USB cable. On a linux PC the following two devices turn up

```
/dev/ttyUSB0
/dev/ttyUSB1
```

The first device it the one that controls JTAG, and the second is the one controlling the RS232 of the Flyswatter2.

You can connect to the `ttyUSB1` with `Minicom`. The OpenOCD is going to connect to `ttyUSB0`.

1.3.2 Install OpenOCD

On fedora you can install OpenOCD by giving the command

```
$ sudo dnf install openocd
```

which as of writing will install 0.9.0. Also you need configuration files for running agains the different evaluation boards. Do

```
$ cd
$ git clone git://github.com/vtss/openocd-config-vtss vtss
```

which will create the directory `vtss` in your home directory. If you are in your home directory, the OpenOCD is started with

```
$ openocd -f interface/ftdi/flyswatter2.cfg -f vtss/serval1-ref.cfg
```

The `flyswatter2.cfg` is located in `/usr/share/openocd/scripts/interface/ftdi` or something similar. In the 2nd file `serval1-ref.cfg` you can find the socket ports that GDB and telnet can connect to. These are commented out in this config file but should be commented in, i.e.

```
telnet_port 4444
gdb_port 3333
```

So this means, that you can telnet to OpenOCD by telnetting to port 4444 on the machine on which OpenOCD is running. Similar GDB can connect to port 3333.

1.3.3 Run GDB

For eCos the GDB tool to use is `/opt/vtss-cross-ecos-mips32-24kec-v2/bin/mipsel-vtss-elf-gdb`. If you go into the `build/obj` directory where the `.elf` file is, and the start GDB, you get the prompt. Then connect to OpenOCD with `target remote a.b.c.d:3333`. In the OpenOCD console you should be able to see that the connect succeed.

Then you shall load the code `load xxx.elf` and the symbols `file xxx.elf`. Then you can set a break point in e.g. `cyg_user_start()` with `break cyg_user_start`. Then say `continue` or just `C` and enter, and the code start running until it hit the break point.

[1] www.tincantools.com/JTAG/Flyswatter2.html

2 Adding a module

In this chapter we shall add a module to the MSCC application. We will to it in two steps. First a simple module is added, where we restrict ourselves to do what is strictly necessary.

2.1 Build simple application:

The `demo/platform` contain a number of files that we eventally will build. But in order to get started we will only build the `demo_simple.c`. It contain an absolute minimum of infrastructure. So when you have made this work your well on your way.

Go through the following steps and remember to take a couple of minutes to look into the `demo_simple.c` and the `module_demo.in_simple`.

1. Add `VTSS_MODULE_ID_DEMO` to `vtss_appl/include/vtss/appl/vtss_module_id.h`. If you use tags in your editor, then search for `VTSS_MODULE_ID_NONE`. You will have to edit 3 places. See the instructions just above `VTSS_MODULE_ID_NONE` in this file.
2. Hook up the demo module with the MSCC application, i.e., edit `vtss_appl/main/main.c`. Search for `VTSS_SW_OPTION_HTTPS` in that file, and do something similar, where you just change the `HTTPS` with `DEMO`. Three thing must be done:
   - Include the `demo_api.h` if `VTSS_SW_OPTION_DEMO` is defined.
   - Add the `demo_init()` function to the `initfun[]` list, if `VTSS_SW_OPTION_DEMO` is defined.
   - Add the `demo_error_txt()` function to the `error_txt()` function, if `VTSS_SW_OPTION_DEMO` is defined.
3. `cp module_demo.in_simple ../../build/make/module_demo.in`. This add what is needed to the makefile system. Above we have defined `VTSS_MODULE_ID_DEMO` to some number. In `module_demo.in` we must set `MODULE_ID_demo` to the same number explicitly.
4. If the line `Custom/AddModules =` does not exist in `config.mk`, then add the line `Custom/AddModules = demo`. If it does exist, then add `demo` to the end of the space seperated list, e.g., `Custom/AddModules = xxx yyy demo`
5. In the build directory run `make clobber`. This is because we have change the make system. And then run `make -j` or just `make`.
6. Upload to code to the target with e.g. the ramload method.

When application eventually boot, you should be able to see some of the `printf` statements in the `demo_simple.c`.

2.2 Build more advanced application:

In this example, then functionality of `demo_simple.c` is replaced with `demo.c`, which has the same `demo_init()`, but with more relevant code put into the different cases. The makefile for demo is replaced with on that build more object files (step 1 below).

1. `cp module_demo.in_advance ../../build/make/module_demo.in`

   If you did not do the simple example in previous section, then go through step 2-4 in that section now.

2. The `demo_icfg.c` application need some defines in `vtss_appl/icfg/icfg_api.h` in order to compile. Follow the instructions in the `demo_icfg.c` file.
3. In the build directory run `make clobber`. This is because we have change the make system. And then run `make -j` or just `make`.
4. Upload to code to the target. When it eventually boot, you should be able to see some of the printf statements in the `demo.c`.

3 The examples

3.1 ICFG

When a device boot, the ICLI commands listed in the startup-config is run. The configuration can se seen with `show running-config`. When running `show running-config` each module is quired for there configuration to the extend that they have registerd configuration parameters. How this is done for the demo module is show in `demo_icfg.c`.

In this file are instructions to register some attributes for the demo module, that you have to do in order to make the code compile.

The output from the attribute functions are used, as mentioned, when `show running-config` is run; but also when you do save the configuration with `copy running-config startup-config`. The output of ICFG must therefore match the ICLI commands if they are going to have any effect.

3.2 ICLI

This is where you implement commands. Look in the `demo.icli` file.

3.3 Trace

The recommended way to implement debug print is not by means of `printf()` but as descibed in `demo_trace.c`.

In our case the have the DEMO module, with is identified `VTSS_MODULE_ID_DEMO`. Within a module we can divide debug prints into groups. See `demo_trace.h`. In `demo_trace.c` the array `trace_grps[]` configure parameters for each group. If we run

```
debug trace module level demo
```

it should be obvious what the meaning of this array. In our case, we have set the print level the `VTSS_TRACE_LVL_WARNING`. This can be changed runtime with

```
debug trace module level demo msg debug
```

After having run this command a print statement in `demo_msg.c` like `T_DG(...)` will show up.

3.4 MSG (demo_msg.c)

The function `demo_msg_tx()` will send a message `DEMO_MESSAGE_SOMETHING` to the module `VTSS_MODULE_ID_DEMO`. This function is called if the ICLI command `demo msg <0-17>` is given. See `demo.icli`.

The `demo_register_msg()`, which is called from `demo.c` when the module is initialized, do register the funciton `demo_msg_rx()`, which is called, when a message is sent to the `VTSS_MODULE_ID_DEMO`.

If you search for `TRACE_GRP_MSG` in `demo_msg.c`, you will see all the places where somehing can be printed to the console. The level on which these mesages are printed can be seen with `debug trace module level demo msg`. The default level is `warning`. In order to see `T_DG()` you'll have to lower the level to altleast `debug`, i.e., `debug trace module levet demo msg debug`. If you do that, and run the ICLI command `demo msg 0` you'll see output on the console that illustrate the functionality.

3.5 Frames to/from the CPU

3.5.1 Receiving a frame

When the CPU shall receive a frame, then there are basically 3 steps as described in the sections below. In a generel setup we can have a stack of switches. One switch is the master and all the others are slaves. The MSCC application run on the master, and all the slaves has to goto a slave state where they do "slave stuff".

If we have only one switch, i.e. no stack, then that device is master. The reason for pointing this out is, that when a frame is received on a port in a stack, and that frame is for the CPU, i.e. the CPU on the master, then the CPU which is on the switch on which the receiving port exist must configure the switch chip so that the frame is sent to the local CPU, regardless on whether the switch is master or slave. This described below in 2.5.1.

When the local CPU get the frame, it must be told what to do with it. That is described in 2.5.2. A function is registers and called when a frames match the criteria configured. This callback function shall make sure, that the frames is forwarded to the master CPU.

And finally the frame arive at the master CPU in 2.5.3.

3.5.1.1 Forward packet to the local CPU

In `demo_forward.c` the 3 methods are shown that will have a frames forwarded to the CPU. Note, that this will only get the frames to the CPU - which sounds obvious, but it does not get the frame to the process that eventually wants it. How this is done is illustrated in a subsequent section.

In a stack solution there is one master switch and all other switchs are slave. The MSCC application run on the master switch/CPU. Therefore when a frame is received on a slave, like a L2CP frame, then that frame first have to be sent to the CPU of the slave on which it was received, and that CPU has to make sure that it is forwarded to the master CPU.

The methods below show how a frame get to the CPU of the switch on which it is received, i.e., master or slave.

1. `forward_frame_to_cpu_method1()` shows how we on a port basis can be configured which L2CP frames shall be sent to the CPU. L2CP frames are frames with destination MAC address `01:80:C2:00:00:XY` where `X=0,2` and `X=0,1,2,...,F`.
2. `forward_frame_to_cpu_method2()` configure the MAC forwarding table. In this case you specify a destination MAC address and some other parameters like VLAN that shall apply to the frames.
3. `forward_frame_to_cpu_method3()`. In this case a ACL rule is configured. The show an api and an appl method.

3.5.1.2 Forward packet from local CPU to master CPU

In `demo_packet.c` a filter and a callback function is configured and registered in `demo_register_packet()`. When a packet is received by the CPU it is match against this filter, and if it match, then callback function `demo_packet_rx()` is called. All this happens on the switch on which the packet is received, i.e., master or slave CPU.

In the callback function `demo_packet_rx()` the packet is forwarded to the master CPU via the l2proto module.

Next section show how to received the packet on the master CPU.

3.5.1.3 Handling of frame at the master CPU

In demo_l2proto.c we register, that packets to the demo module, which is identified with VTSS_MODULE_ID_DEMO (see section 1.2), shall be sent to the callback function demo_l2_rx(). This function copy the frame and some other information to the demo_rx_buffer and then signals the thread that shall process the frame.

The processing thread is demo_thread() in demo.c

3.5.2 Transmit a frame

In demo_tx_frame.c it is show how a frame ca be sent. The function demo_tx_frame() is called by doing the ICLI commands

```
# configure terminal
(config)# interface gi 1/1  //e.g.
(config-if)# demotx [vtss-os-tx | packet-tx] 00:01:c1:11:22:33 [switch]
```

In case of vtss-os-tx the switch parameter does not have any effect, and can be left out. This will call the demo_os_tx_method_1(port_no,dmac) function. The 00:01:c1:11:22:33 is the dmac. The frame will be sent on port port_no and apply to a stack.

In case of packet-txx the switch parameter do apply. In this case the demo_packet_tx_method_2(port_no,dmac,switch). If switch=false then the frame will be sent on port port_no. This function only work on the local switch in case of a stack. If swtch=true, then the port_no does not apply and the frames is switched the normal way.

3.5.3 Test

Have two switch that are connected. On one switch enable trace messages, i.e.

```
# platform debug allow
# debug trace module level demo packet debug
```

This enable debug info from demo_packet.c when a frame is received that match this filter, which is frames sent to the CPU with destination MAC address 00:01:c1:11:22:33. This filter has been installed then the demo module was initialized. See in demo.c; serach for function demo_register_packet().

In order for instruct frames that will match this filter to be sent to the CPU, run

```
# configure terminal
(config)# demo forward mac
```

which will run method 2 in demo_forward.c. Now the receiver is ready.

On the other switch do

```
# configure terminal
(config)# int gi 1/1
(config-if)# demotx packet-tx 00:01:c1:11:22:33 switch
```

This will run method 2 in demo_tx_frame.c, and the remoten end show say that a frames has been received.

If the switch option was not given, then the frame would only be sent on port gi 1/1 in which case the remote end would only receive the frame if it happened to be on the port.

The behavoiur is similar if packet-tx without switch is replaced with vtss-os-tx. The only difference is that the second case will work in a stack. If a stack does not apply, then the packet-tx method is prefered.

3.5.4 Sockets

In the files demo_socket_[client|server].c an example is show on how to use sockets. The struct is similar to how things are done in UNIX. The example show a server which wait for a client to connect, and then send back a message and the stop the server again. The server is started with

```
#configure terminal
(config)# demo socket server
```

On the other device the client is run with

```
#configure terminal
(config)# demo socket client a.b.c.d
```

where a.b.c.d is the IP address of the server device, which is found with

```
# show ip int b
```

3.6 Web

The demo_web.c contain a simple example of web code. In this file it is explained how to hook up a web menu in the system. This is done by editing the vtss_appl/web/menu_default.cxx. In order to have a web menu, some html code must be implemented. In this example this is located under demo/platform/html/demo.htm. Look in the module_demo.in_advance and search for WEB_CONTENT. If the ICLI command debug trace module level demo web debug is given, then output can be seen in the console when changes are performed in the web.