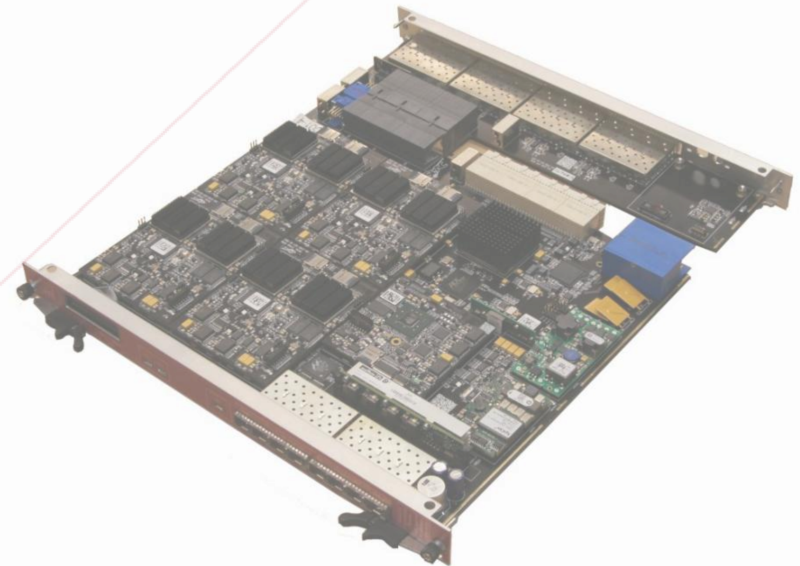
Hello World Exercise

This exercise will introduce the SDK tools and the programmers' development cycle on the RCE. A simple program will be created, loaded and run

Objectives:

- Get started with the SDK
- Connect to an RCE via telnet
- Create and load a simple shared library
- Create and run a Task
- Parameterize the .so and .exe with a SVT



SLAC NATIONAL ACCELERATOR LABORATORY PPA Particle Physics & Astrophysics RPT

Part One: Getting started

Your software/system administrator will have loaded the SDKs to a common group area. Let us call that area $RPT_ROOT. For the CERN workshop, this area is: `/home/workshop/V0.7.1-WS`. The exercise presenter (or your local contact) should have given you an RCE address to use. Remember, this address is of the form shelf/slot/bay/element, i.e. "`snowbird/1/0/0`". For this exercise, this is the RCE address we'll use as an example. Your network administrator should also tell you what interface the RCE is visible on. For the CERN workshop, this is `eth0`.

1 Set up the SDK environment by sourcing your environment scripts in your favorite shell. For this exercise, we'll use /bin/bash.

```
bash> source /home/workshop/envs.sh
```

If you are a tcsh user only:

```
tcsh> source /home/workshop/envs.tcsh
```

Comments

The setup scripts will add the SDK bin directories to your PATH.

2 Dump your RCE's IP information.

```
bash> atca_dump snowbird/1/0/0 --ifname p1k1
```

Output

3 Connect to your RCE via telnet

```
bash> telnet $(atca_ip snowbird/1/0/0 \
        --ifname p1k1)
```

Note

You should see the RTEMS prompt: [/]

4 List the contents of /, mount your home directory via NFS and then exit.

```
[/] ls /
[/] mkdir /test
[/] mount -t nfs 192.168.210.9:/home/<user> /test
[/] ls /test
[/] exit
```

Part Two: The Hello World Shared Library

The Hello World example starts with creating a shared library, loading it into memory, and observing the output in the system log. The shared library is then hooked into a simple task

2 Copy `workshop_examples.tgz` from `/home/workshop` and expand it.

Change directory to `workshop_examples/hello_example` and open the `hello_so_1.c` source file using, for example, the emacs editor.

```
bash> cd workshop_examples
bash> emacs hello_so_1.c
```

3 Sample code:

```
#include <stdio.h>
#include "debug/print.h"
#define PRINT dbg_printv
int hello(void) {
  PRINT("Hi! I'm a .so!\n");  return 0;
}
int goodbye(void) {
  PRINT("Goodbye .so!\n");  return 0;
}
int lnk_prelude(void* arg,
                void* elf) {
  PRINT("Hello prelude!\n");
  hello();
  goodbye();
  PRINT("Goodbye prelude!\n");
  return 0;
}
```

Notes and Comments

- We are using the `dbg_printv` function to send output to the system log. `printf` sends to `stdout`, which is the console.
- `lnk_prelude` is called when the library loads

SLAC NATIONAL ACCELERATOR LABORATORY PPA Particle Physics & Astrophysics RPT

4 Take a look at build.sh, which compiles and links this whole example. Your shared library is output as hello_1.so.

build.sh uses the rtems-gcc and rtems-ld scripts discussed in the software development (SD) talk to compile and link the example.

Notes

- rtems-gcc and rtems-ld are wrappers around the actual cross compilers (currently installed in /opt/rtems-4.11).
- Observe the "examples:" string in the link statement. This is the namespace discussed in the SD talk.
- Observe also the -l:rtems.so fragment which allows resolution of the dbg_printv symbol.

5 telnet back to your RCE, as in part 1, and see that your new shared library is in your directory. Also look in the "compiled" subdir of workshop_examples and see that the output appears there as well.

```
[/] ls /test/workshop_examples/hello_example
[/] ls /test/workshop_examples/compiled
```

6 Create the "examples" namespace that points to your "compiled" subdirectory. Check that the path is fine using ns_map.

```
[/] ns_assign examples
/test/workshop_examples/compiled
[/] ns_map examples:hello_1.so
```

Notes

- You should see the proper path as a result of the ns_map command.

SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

7

Load the library. Observe that the output is in the system log.

```
[/] load examples:hello_1.so
[/] syslog
```

Output

Here, we elide over the timestamp.

```
… Hello prelude!
… Hi! I'm a .so!
… Goodbye .so!
… Goodbye prelude!
```

7

Now, we'll link `hello_1.so` with a Task. (Tasks were explained in the SD talk.) First, open the `hello_task.c` code in your editor.

```
#include <stdio.h>
#include "debug/print.h"
#include "task/Task.h"
#define PRINT dbg_printv
// Functions from hello.so
extern int hello(void);
extern int goodbye(void);

void Task_Start(int argc,
                const char** argv) {
  PRINT("Hello from Task!\n");
  hello();
  PRINT("Return from Start.\n");
  return;
}
void Task_Rundown() {
  goodbye();
  PRINT("Goodbye from Task!\n");
  return;
}
```

Notes

- Note the new include of `Task.h`. This include defines the Task semantics. (See the SD talk)
- All tasks must have a `Task_Start` and a `Task_Rundown`. These are the entry and exit points.

8

Look again at `build.sh` for the linking of `hello_1.so` with `hello_task.o`. The script uses `rtems-task` to perform the linkage.

Notes

- The `rtems-task` statement references both `hello_task.o` and `hello_1.so`

SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

9	Now, run the new task.	**Output**
	`[/] run examples:hello_1.exe` Hello prelude! Hi! I'm a .so! Goodbye .so! Goodbye prelude! Hello from Task! Hi! I'm a .so! Return from Start. Goodbye .so! Goodbye from Task!	• Observe that the prelude loads and its output appears **before** the task.

SLAC NATIONAL ACCELERATOR LABORATORY PPA Particle Physics & Astrophysics RPT

Part Three: Parameterization

Hello world continues by adding a parameterization to the task via the SVT mechanism discussed in the Software Development talk.

1

Open `hello_svt.c` in your editor.

```
char const HELLO_MESSAGE[]= \
  "Hello from svt!";
char const GOODBYE_MESSAGE[]= \
  "Goodbye from svt!";
```

Notes

- Notice that `hello_svt.c` only defines two symbols. That's what an SVT (Symbol Value Table) is for!
- You can put anything you want in a symbol, be it an array, struct or instance of a C++ class.

2

Now look at `hello_so_2.c` in your editor.

```
#include <stdio.h>
#include "svt/Svt.h"
#include "debug/print.h"

#define PRINT dbg_printv
#define NUM 15
#define TABLE (1<<NUM)

int hello(void) {
  PRINT("Hi! I'm a .so!\n");
  const char* hm =
     Svt_Translate("HELLO_MESSAGE",TABLE);
  if(hm) PRINT("%s\n",hm);
  return 0;
}
int goodbye(void) {
  const char* gm =
     Svt_Translate("GOODBYE_MESSAGE",TABLE);
  if(gm)  PRINT("%s\n",gm);
  PRINT("Goodbye .so!\n");
  return 0;
}
```

Notes

- Since `hello_so_2.c` needs to deal with SVTs, include the relevant header.
- We're going to create our own table, let's choose number 15. We also need it as a bitmap.
- `Svt_Translate` is the lookup of the symbol. If the lookup fails, 0 is returned.

SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

```
int lnk_prelude(void* arg,
                void* elf) {
  PRINT("Hello prelude!\n");
  hello();
/* install the hello SVT */
  Svt_Install(NUM, "examples:hello.svt");

  goodbye();
  PRINT("Goodbye prelude!\n");
  return 0;
}
```

- In `lnk_prelude`, we install the SVT into its table location, referencing the SVT by **namespace**.
- Since the SVT is installed, it may not be uninstalled without extensive dependency tracking. Even Linux doesn't do this. Once it is installed, it stays until the next reboot.

3

Again, examine build.sh for the compiling and linkage of `hello_svt.c` to `hello.svt`. The output should already be in your "`compiled`" directory.

The task object `hello_task.o` is linked with `hello_2.so` into `hello_2.exe`

Notes

- Compiling an SVT is exactly like compiling a regular C or C++ file.
- Linking an SVT requires use of the `rtems-svt` wrapper script, discussed in the SD talk.
- Using an SVT requires nothing in the link statement. The linkage is done programmatically via `Svt_Install()`.

4

Now, we run the `hello_2.exe` on the RCE.

```
[/] run examples:hello_2.exe
```

Hello prelude!
Hi! I'm a .so!
Goodbye from SVT!
Goodbye .so!
Goodbye prelude!
Hello from Task!
Hi! I'm a .so!
Hello from the svt world!
Return from Start.

Notes

- When `hello.exe` loads `hello_2.so`, the SVT is not loaded until **after** trying the `hello()` function in the .so. Therefore, the lookup of `HELLO_MESSAGE` from of the SVT returns null.
- However, GOODBYE_MESSAGE **is** found, as its lookup is after the SVT load.
- When the Task runs `hello()`, the SVT is loaded so the lookup of `HELLO_MESSAGE` works fine.

SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT

Goodbye from the svt world!
Goodbye .so!
Goodbye from Task!

5

Edit the SVT and change the messages to whatever you want.

Recompile with `build.sh`, then reset your RCE from Linux:

```
bash> cob_rce_reset snowbird/1/0/0
```

Wait ~30 seconds until your RCE boots. Then telnet back in and remount the NFS drive as in part 1, step 3 and 4.

Reassign your namespace as in part 2, step 5. Run `hello2.exe` as in step 4.

Notes

- We reset the COB (or reboot it) as an SVT is installable exactly **once** per boot.
- After resetting (and without relinking your task), the message will have changed.

SLAC NATIONAL ACCELERATOR LABORATORY
PPA Particle Physics & Astrophysics
RPT