RCE/COB Gen3 MiniWorkshop

Software Development

Jim Panetta (panetta@slac.stanford.edu)

Overview: What you should get from this talk

- Software Development Kits (SDK) are provided
- Host and target: two different things (usually)
- Software consists of a set of shared libraries
- RCEs are distributed with software already in flash
- Updating RCE software is easy
- Development and production level code are supported

Software Development Kits

Software is distributed in two forms

- Primary: Binary files on µSD card which will boot an RCE
- Secondary: Software development kits

SDKs are monolithic and self contained

- They may be placed anywhere
- As many as needed may be installed
- They are **not** a build system – they provide **tools** for building the software

Installed SDKs may be updated with new releases

Embedded System Development

Target	The ultimate destination of software
Host	Where software is written
SDK	Software Development Kit. Tools used to develop software on a host for a target

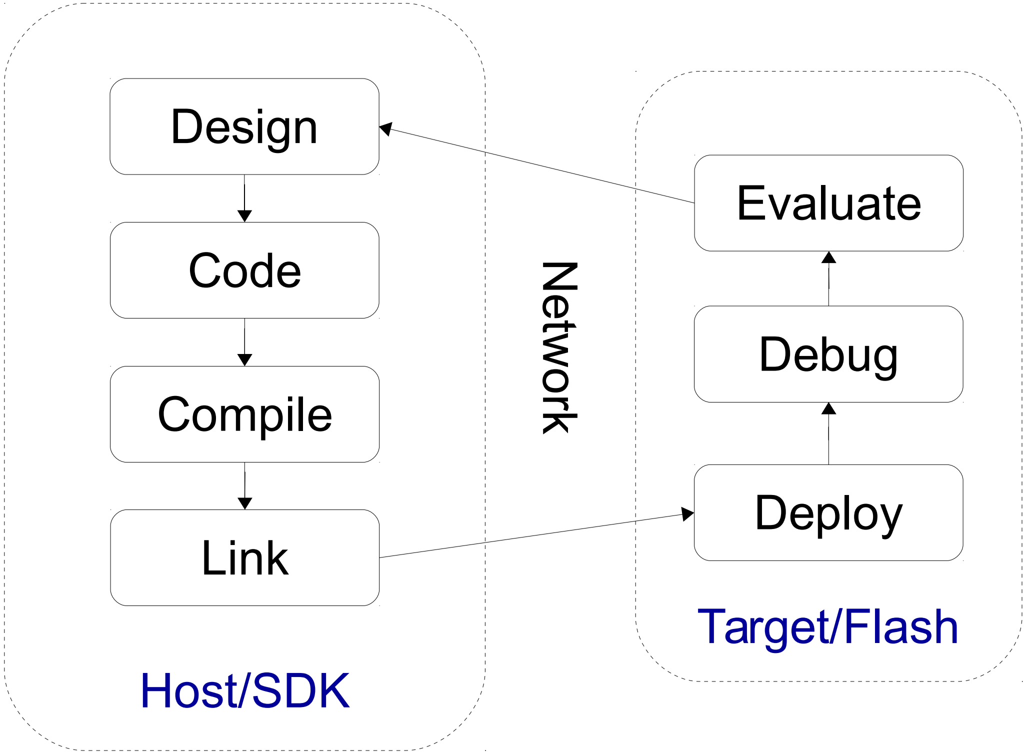
Embedded system development is different than developing for Linux

- Access to the target is limited
- Target working environment is limited
- Target is less powerful than host

Development model

Cross development is a split model

- The majority of work is done on the host, compiling code which can run on the target
- Linking is done on the host against a **copy** of the target's software
- The final debugging and evaluation is done on the target
- This is the circle of code



Programming/Execution Environments

DAT provides SDKs for several computing environments:

- RTEMS on the RCE (ARM Cortex-A9)
- Host Linux
  - RHEL 5/6 and Scientific Linux 5/6 (i86/32, x86/64 in dev)
- Embedded Linux
  - ArchLinux (ARM Cortex-A9)
  - In development: Not currently supported

RTEMS SDK Contents

Bootable Image

(RTEMS, C++, SD Driver,
ShLib Support, MemMGT)

Bootstrap Items

(Bootstrap loader, FPGA image)

Host Tools

(Compiler wrappers, shared libraries,
scripts, ATCA probes)

RCE Utilities

(Network Drivers, console, telnet,
NFS, shell, DSL)

Configuration DB

Include Files

Example Code

Items in blue are pre-loaded on the SD cards

Getting the RTEMS SDK

- Fetch core scripts via **git** using a tag provided by DAQ:

```
git clone -q --branch rtems-V0.0.0 \
  http://www.slac.stanford.edu/projects/CTK/SDK/rtems/common.git \
     <install_location>
```

- Install the cross compilation tools (if needed):

```
sudo <install_location>/tools/install-devtools.sh
```

- Finalize the install by fetching libraries, includes and compiling the template code:

```
<install_location>/tools/install-sdk.sh
```

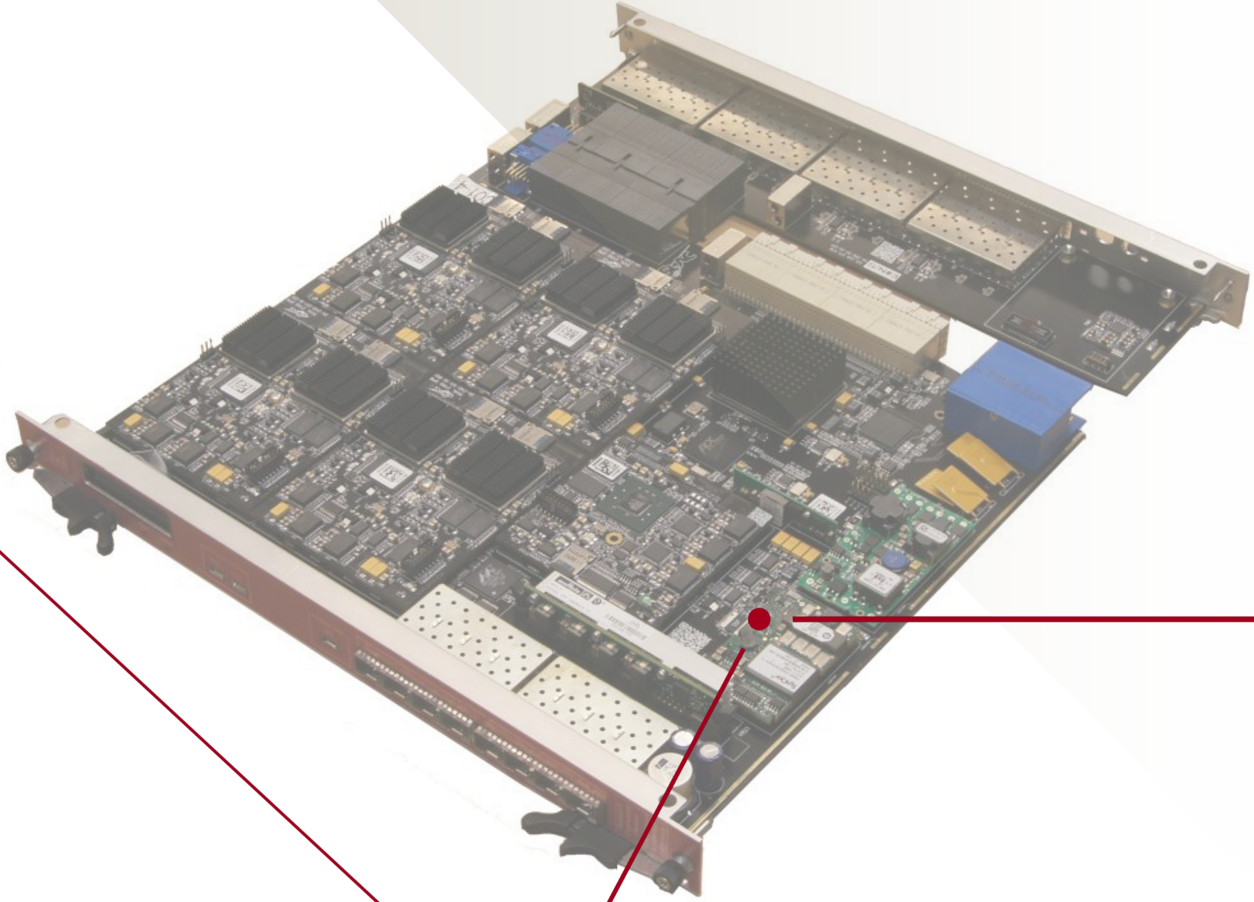
- Last step is to add the DAT environment to yours:

```
source <install_location>/tools/envs.{csh,sh}
```

Micro-SD Flash Filesystem

- Six Partitions on 32 GB uSD card
  - BOOT – Bootstrap, FPGA bit file (invis on RTEMS)
  - SCRATCH – User writable partition (r/w)
    (largest partition ~ 16 GB)
  - RTEMSAPP – Application and configuration files (r/w)
  - RTEMS – DAT RTEMS installation (r/o)
  - ArchLinux – System files for ARM Linux (invis on RTEMS)
  - LinuxKernel – Linux kernel files (invis on RTEMS)

RTEMS



RTEMS Real Time Kernel

Think of RTEMS as analogous to to the Linux Kernel

- Hard Real-Time: Fully deterministic
- POSIX 1003.b API (incl. pthreads)
- Multithreading, Interrupts, Semaphores, IPC, etc.
- Networking stack (TCP, UDP, DHCP, NTP)
- File Systems (DOSFS, TFTP, NFS)
- Utilities: telnet, simple shell
- Floating point & SMP (in development)

Documentation:

- http://rtems.org/onlinedocs/doc-current/share/rtems/html/

RTEMS – DAT's Added Value

DAT provides added services on top of RTEMS

- Dynamic linker / Shared library support
- Task Management
- Symbol-Value Abstraction (SVT)
- Filesystem Abstraction (Namespaces)
- Lightweight Distributed Client-Server model (DSL)
- Console, telnet and shell support
- C++ support

Note: Currently, the DAT system is not based on an RTEMS release – 4.10 does not have ARM support and 4.11 is not released yet. When 4.11 is released, we will use that.

Shared or Dynamic Libraries

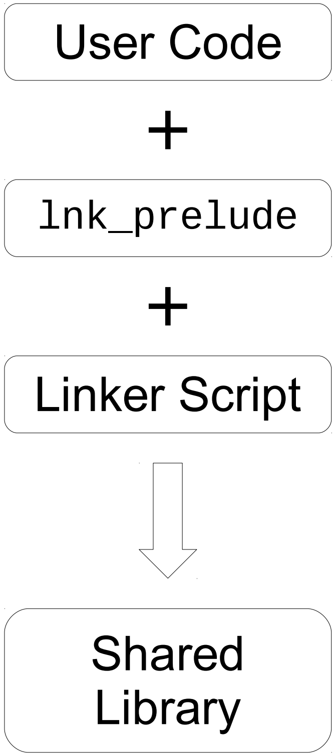
RTEMS downloaded from rtems.org is linked statically

- Problem: Constantly recompiling/relinking against core kernel
- Problem: Removing unneeded code for prod. systems is hard

DAT provides a shared library system which allows modularity

- Mix and match for production and development systems
- Allows much simpler designs to interfaces
- Evolution of core is independent of user software
- No need to re-link on non-major changes

Shared or Dynamic Libraries (cont.)



A shared library is simply user code and an optional prototype, all glued together with a linker script from the SDK.

On load, the core relocates the library and prepares it for execution.

The optional prototype (lnk_prelude) executes **once**, when the library is loaded, and executes in the context of the Task which started it.

Code is distributed as shared libraries on SD.

Example: hello.so – A Simple Shared Library

```
#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;
}
```

On the host, this is compiled:

```
rtems-gcc --arm hello.c \
   -I $install/include/core \
   -I $install/include/rtems \
   -o hello.o
```

and then linked:

```
rtems-ld --arm -L$install/lib\
   -l:rtems.so \
   -Wl,-soname:examples:hello.so\
   -o hello.so
```

then copied to the target for use later. Note the use of “`examples`”. This is an example of a namespace.

Namespaces

Namespaces are used by shared libraries to abstract paths and mount points to allow a software system to advance without requiring a re-link. Namespaces may not contain a colon (`:`)

They may be created in the RTEMS shell:

```
$ ns_assign examples /mnt/rtemsapp/examples
```

They also may be queried, renamed and removed:

```
$ ns_map examples:hello.exe
Path=/mnt/rtemsapp/examples/hello.exe

$ ns_rename examples murgatroyd

$ ns_remove examples
Error: Ldr_Remove(example) returned 0

$ ns_remove murgatroyd
```

All this can be done programmatically, too. See appendix for more info.

Why Namespaces? Why not just use the path?

Namespaces allow design and organization to go forward without having to recompile all code.

- Do you really want to hard-code the file path?
- … in eleventy-dozen places?
- What happens when the file system changes?

A namespace means simply:

All things in this namespace are in the same place

They allow mixing, matching and shuffling of code around during the development cycle, without creating dependencies into the filesystem.

Tasks

Tasks are like Linux executables

- They have a well defined entry point
- They depend on (link against) other shared libraries
- They execute in their own context (they have their own resources such as a stack)

In this system, a task is implemented as a shared library with a well defined entry point, as well as a defined cleanup function.

These entry and cleanup functions are `Task_Start` and `Task_Rundown`.

Example: hello.exe – A simple Task

```
#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;
}
```

The user implements `Task_Start` and `Task_Rundown`

Then compile and link:

```
rtems-gcc --arm hello.c \
   -I $install/include/core \
   -I $install/include/rtems \
   -o hello.o

rtems-task --arm hello.o \
   -L$install/lib -l:rtems.so \
   -l:hello.so
   -Wl,-soname,examples:hello.exe\
   -o hello.exe
```

Running a Task

Once a task and its associated shared libraries are created and copied to the target (more on that later), it's time to run them!

```
$ ns_assign examples /mnt/rtemsapp/examples

$ run examples:hello.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!
```

Notice that the prelude executes when the .so is loaded

The task can use the functions in the .so.

When `Task_Start` returns, `Task_Rundown` is automatically run.

Symbol Value Tables: Configuration Data

All this is fine for development, but what about production?

(Nobody wants to have to type at a shell on 2000 embedded systems.)

The Symbol Value Table (SVT) is a shared library construct which matches strings to data structures in code. SVTs may be replaced without recompiling other .sos or .exes.

Example: startup services

You have a set of services that need to run at startup. However, these will change as a function of time.

Solution: Define a list in an SVT and modify it as needed.

```
const char*
INIT_STARTUP_SERVICES[] = {
  "system:nfs.so",
  "system:shellx.so",
  "system:telnet.so",
  "system:console.exe",
  "config:appinit.so",
  "system:dsld.exe",
  NULL
  };
```

Symbol Value Tables

Effectively, the SVT provides an answer to the question:

Would you rather have your parametrization buried in code, or would you rather it be external?

- SVTs can contain any data structure.
- After boot, SVTs are **read-only**.
- There are 32 possible SVTs.
- SVT 31 is the System SVT. (network, default OS settings)
- SVT 30 is the Application SVT. This is where the we put application startup info and user controls.
- It is expected that users will modify the App SVT. But you don't have to since...
- Users may install an SVT of their own.
- Example SVT contents:
  - Namespace definitions
  - Defaults of any kind
  - Shared data between tasks

Example: hello.svt – Parametrize hello.so

hello_svt.c:

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

hello_so.c:

```
#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;
}
// continued next slide
```

`hello_svt.c` is very simple: two lines of code. The two strings `HELLO_MESSAGE` and `GOODBYE_MESSAGE` will be available to the SVT interface, and reference the two char arrays.

Since `hello_so.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.

Here's the lookup. If the lookup fails, 0 is returned.

Example: hello.svt – Parametrize hello.so (cont)

hello_so.c (cont.):

```
int goodbye(void) {
  const char* gm = Svt_Translate
      ("GOODBYE_MESSAGE",TABLE);
  if(gm)
    PRINT("%s\n",gm);
  PRINT("Goodbye .so!\n");
  return 0;
}

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;
}
```

Here's the lookup for the other symbol.

And then in `lnk_prelude`, we install the newly created SVT.

Compile exactly as above:

```
rtems-gcc --arm hello_svt.c \
   -I$install/include/core \
   -o hello_svt.o
```

Linking uses its own script, analogous to `rtems-task` & `rtems-so`:

```
rtems-svt --arm hello_svt.o \
   -L$install/lib -l:rtems.so \
   -Wl,-soname,examples:hello.svt \
   -o hello.svt
```

Example: hello.svt – Parametrize hello.so (cont)

The new svt, exe and so are then copied to flash and run exactly as above:

```
[/] # run examples:hello.exe
Hello prelude!
Hi! I'm a .so!
Goodbye from the svt world!
Goodbye .so!
Goodbye prelude!
Hello from Task!
Hi! I'm a .so!
Hello from the svt world!
Return from Start.
Goodbye from the svt world!
Goodbye .so!
Goodbye from Task!
```

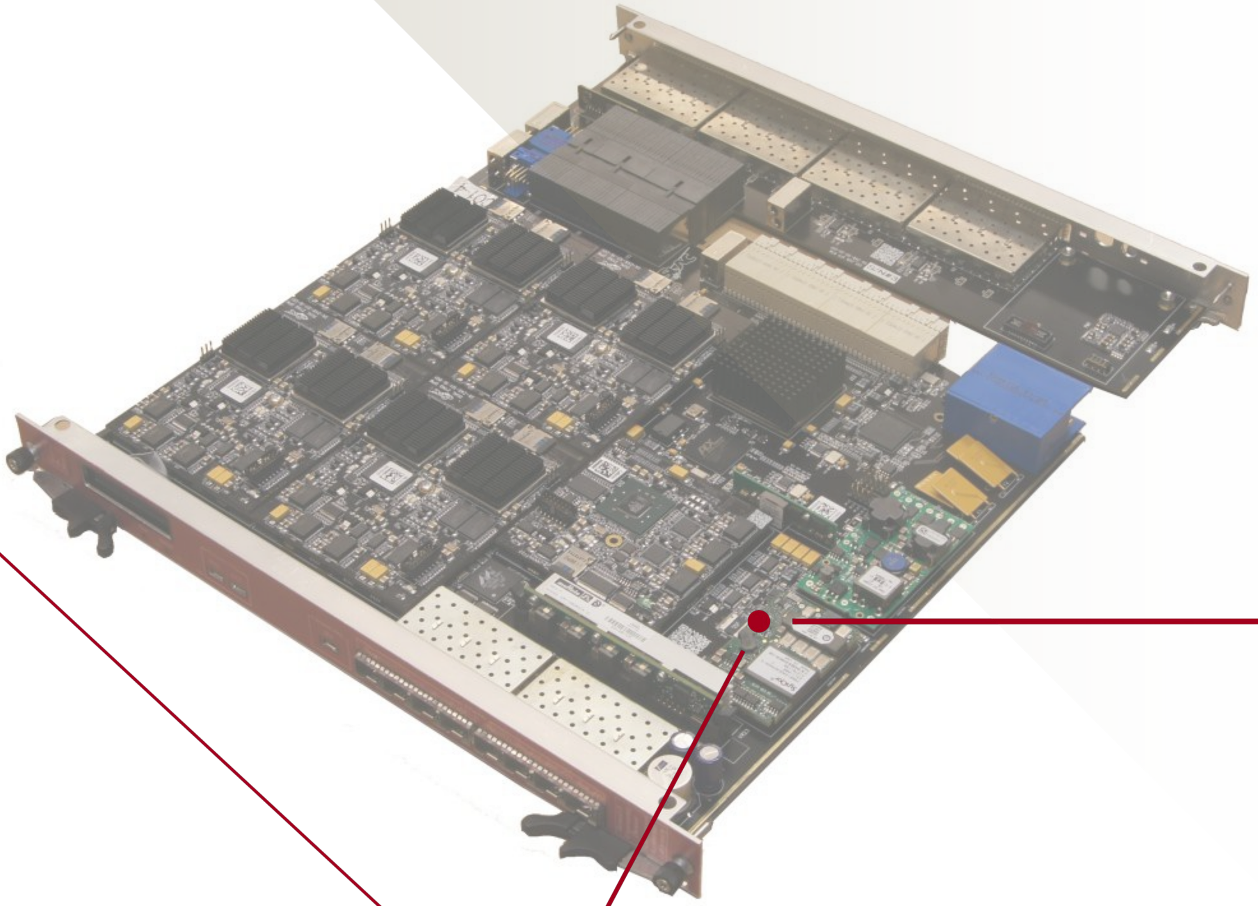
Notice that when `hello.exe` loads `hello.so`, the SVT is **not** loaded until after trying the `hello()` function in the .so. Therefore, the lookup of `HELLO_MESSAGE` from of the SVT returns null.
However, GOODBYE_MSSAGE is found, as it's lookup is after the SVT load.

Copying from Host to Target

Transferring shared libraries to the target in V0.0.0 of the code is *slightly* complicated:

- `telnet` to the RCE using the IP address from `atca_ip`
- `reboot -t linux` to switch to ArchLinux
- Wait until the Linux side boots (< 30 seconds)
- `scp` your shared libraries to the directory you want (they're the same on both Linux and RTEMS). u/p == root/root
  ```
  scp <image> root@<IP>:/mnt/wherever
  ```
- `ssh` to the RCE and log in as `root`
- `reboot_rtems` and wait for RTEMS (< 30 seconds)
- You are now back where you started

Other Operating Systems



Linux – Host and Target

Host:

- Red Hat Enterprise Linux 5/6 (i86-32)
- Scientific Linux 5/6 (i86-32)
- x86-64 compatible libraries in development
- Compiled under RHEL5 (for now for forward compatibility)

Target:

- Arch Linux (ARM Cortex-A9 on the RCE)
- **Not** real-time
- In development: future support

Linux SDK Contents

The Linux SDK contains fewer constituents than the RCE SDK

- Host Libraries
- Include Files
- Host Tools
  (Compiler wrappers, scripts, ATCA probes)
- Example Code

The example code contains the code for the ATCA host tools

Getting the Linux SDK

- Fetch core scripts via **git** using a tag provided by DAQ:

```
git clone -q --branch linux-V0.0.0 \
  http://www.slac.stanford.edu/projects/CTK/SDK/linux/common.git \
     <install_location>
```

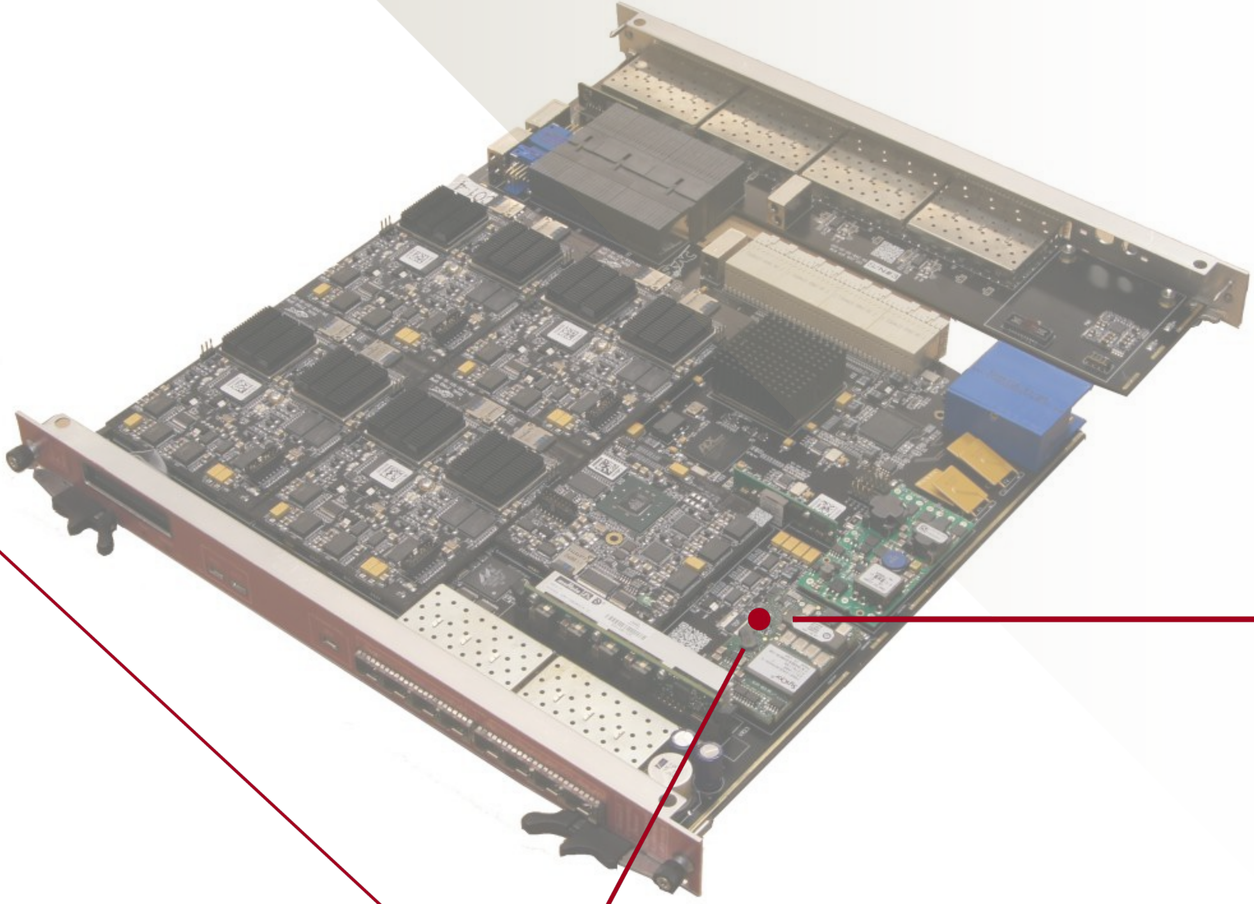
- Finalize the install by fetching libraries, includes and compiling the template code. Architecture is `i86-linux-32` or `arm-linux-rceCA9`

```
<install_location>/tools/install-sdk.sh <architecture>
```

- Last step is to add the DAT environment to yours:

```
source <install_location>/tools/envs.{csh,sh}
```

Appendix



Useful RTEMS Shell Commands

Namespace related commands:

- `ns_assign <namespace> <path>`
- `ns_map <namespace>:<image>`
- `ns_remove <namespace>`
- `ns_rename <namespace> <path>`

Task related commands:

- `run <namespace>:<image> <image arguments>`
- `task  (Lists tasks by ID and name)`
- `stop [id/name]`
- `suspend [id/name]`
- `load <namespace>:<image>`

More Useful RTEMS Shell Commands

Informational commands:

- `ifconfig`
- `syslog [-c]     (dump syslog)`
- `sysinfo         (print system info)`

Other commands:

- `reboot [-t <rtems|linux|ramdisk>]`

Shared Library API

- The API is primarily documented in the include files:
  - SVT: include/core/svt/Svt.h
  - Task: include/core/task/Task.h
  - Loader: include/core/ldr/Ldr.h