

# AndESLive™ Modeling Training

## SID-based Component Modeling



Jonson Chen

# Outline



- ❖ Virtual (Evaluation) Platform Introduction
- ❖ Andes Virtual Evaluation Platform (VEP)
- ❖ SID: Simulation Backbone of AndESLive
- ❖ SID Component Modeling
- ❖ Integrating SID Components in AndESLive
  
- ❖ A little touch on SystemC Modeling
- ❖ How VEP can be used in SoC Development Cycle

# Virtual Platform Introduction

From physical to virtual and vice versa



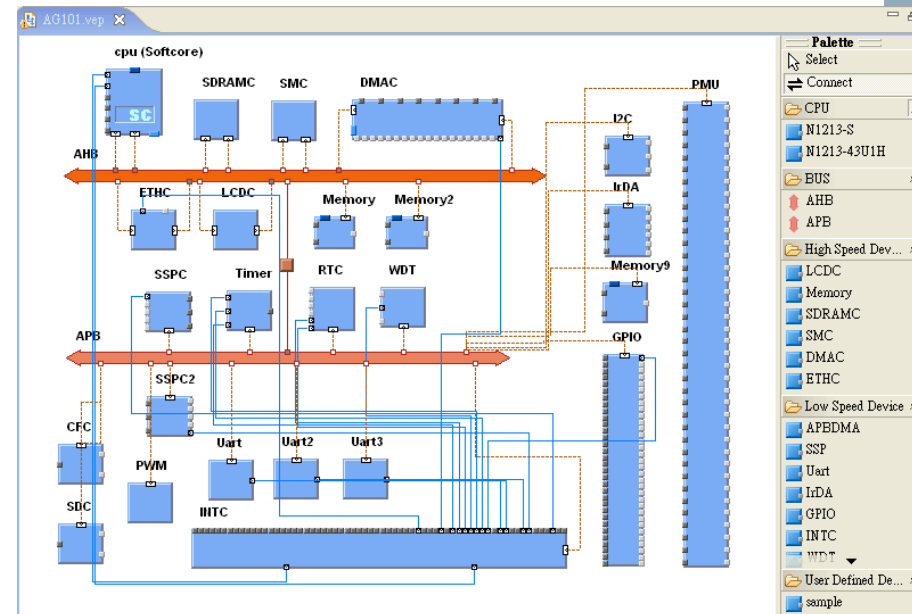
# What is Virtual Platform?



“It is a system-level **simulation model** that characterizes real system behavior. It operates at the level of *processor instructions, function calls, memory accesses and data packet transfers*, etc, as opposed to the bit-accurate, nanosecond-accurate logic transitions of a register transfer level (RTL) model.”\*



Andes Development Platform



Andes Virtual Platform

\*from the book

*ESL Design and Verification: A Prescription for Electronic System Level Design Methodology.* B. Bailey, G. Martin and A. Piziali. Elsevier Morgan Kaufmann, 2007

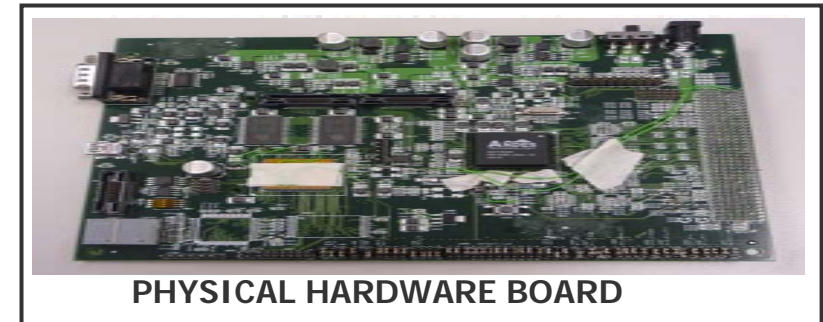
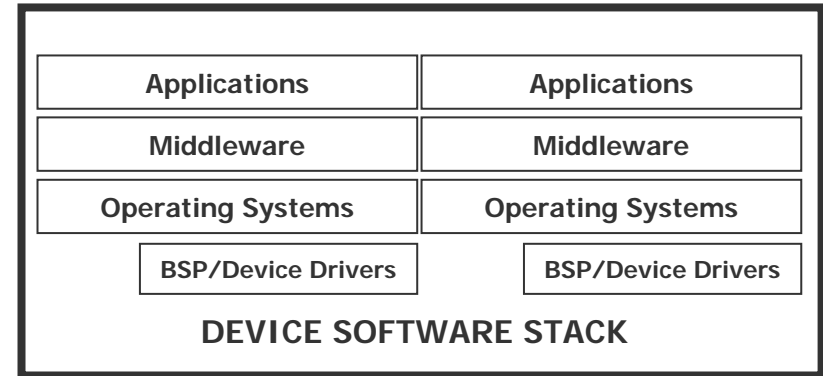
# S/W Development with Physical EVB



## SW Developer Desktop



## Target Hardware

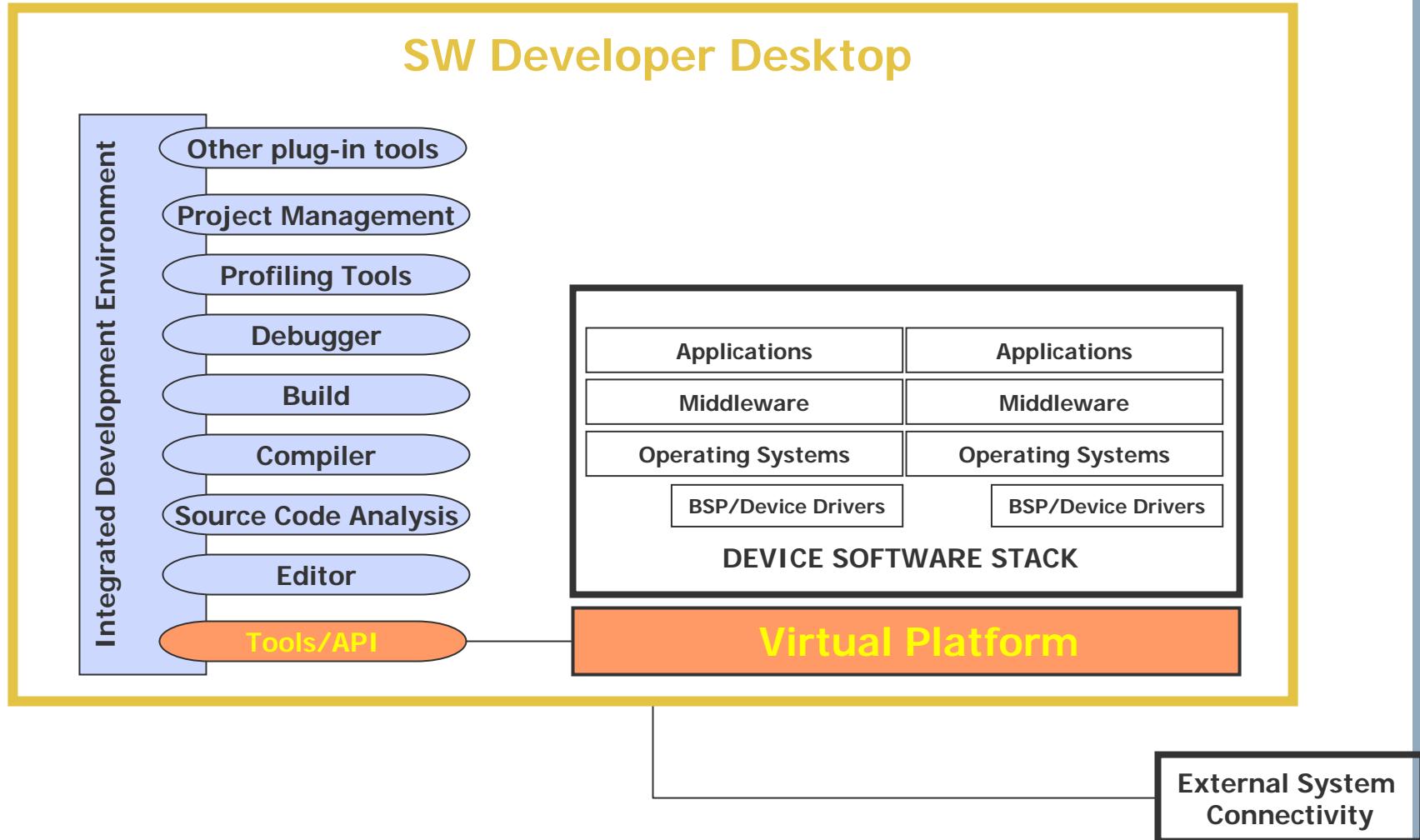


## Physical Target Connection

On-Chip-Debug, Ethernet, USB, ...

External System Connectivity

# S/W Development with Virtual Platform

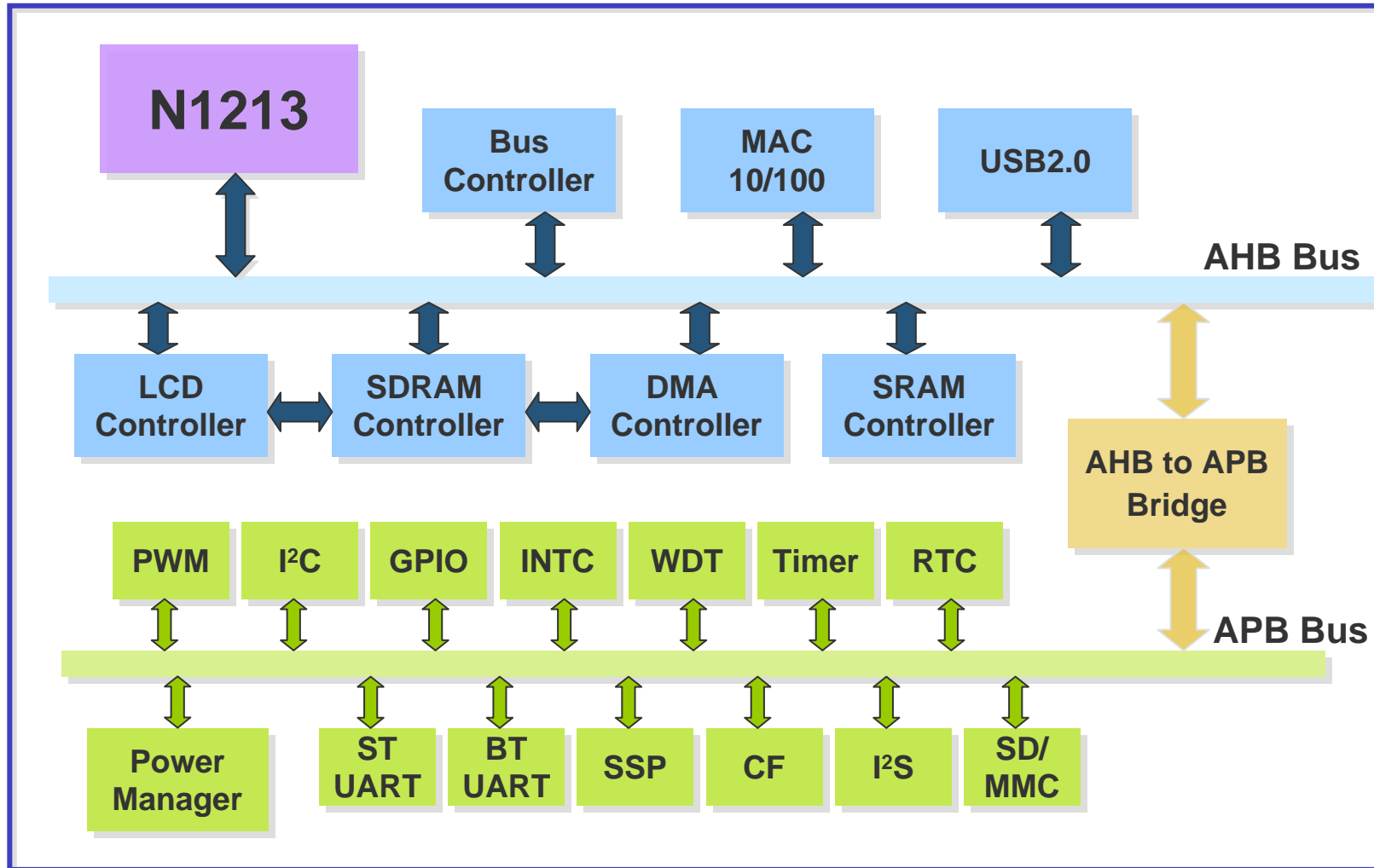


# Andes **V**irtual **E**valuation **P**latform

ADP-AG101

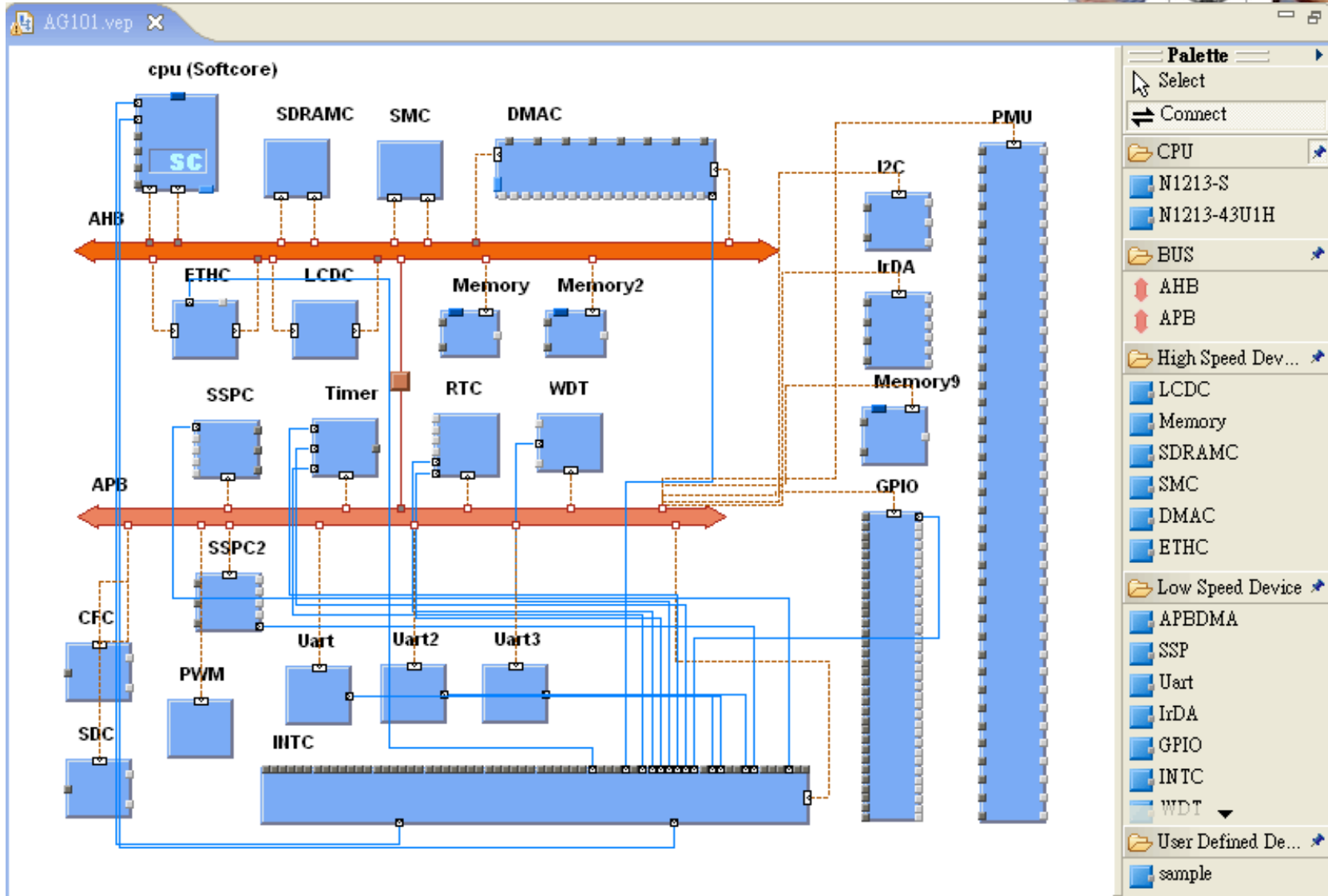


# AndeShape™ Platform SoC: AG101

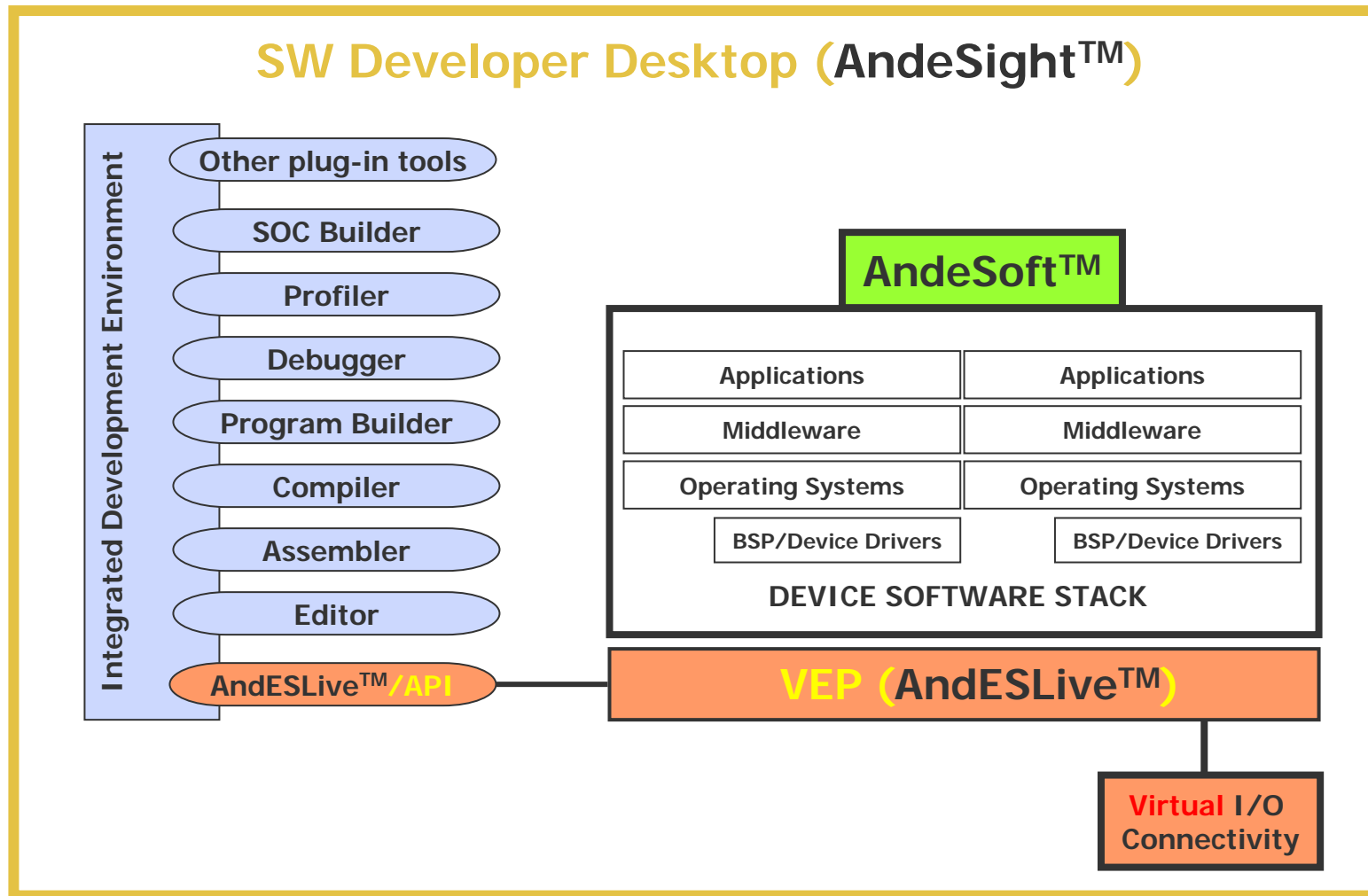




# AG101 Virtual Evaluation Platform (VEP)



# S/W Development with Andes Tools



# AndESLive™ VEP Environment

The screenshot displays the AndESLive VEP Environment interface, which includes several key components:

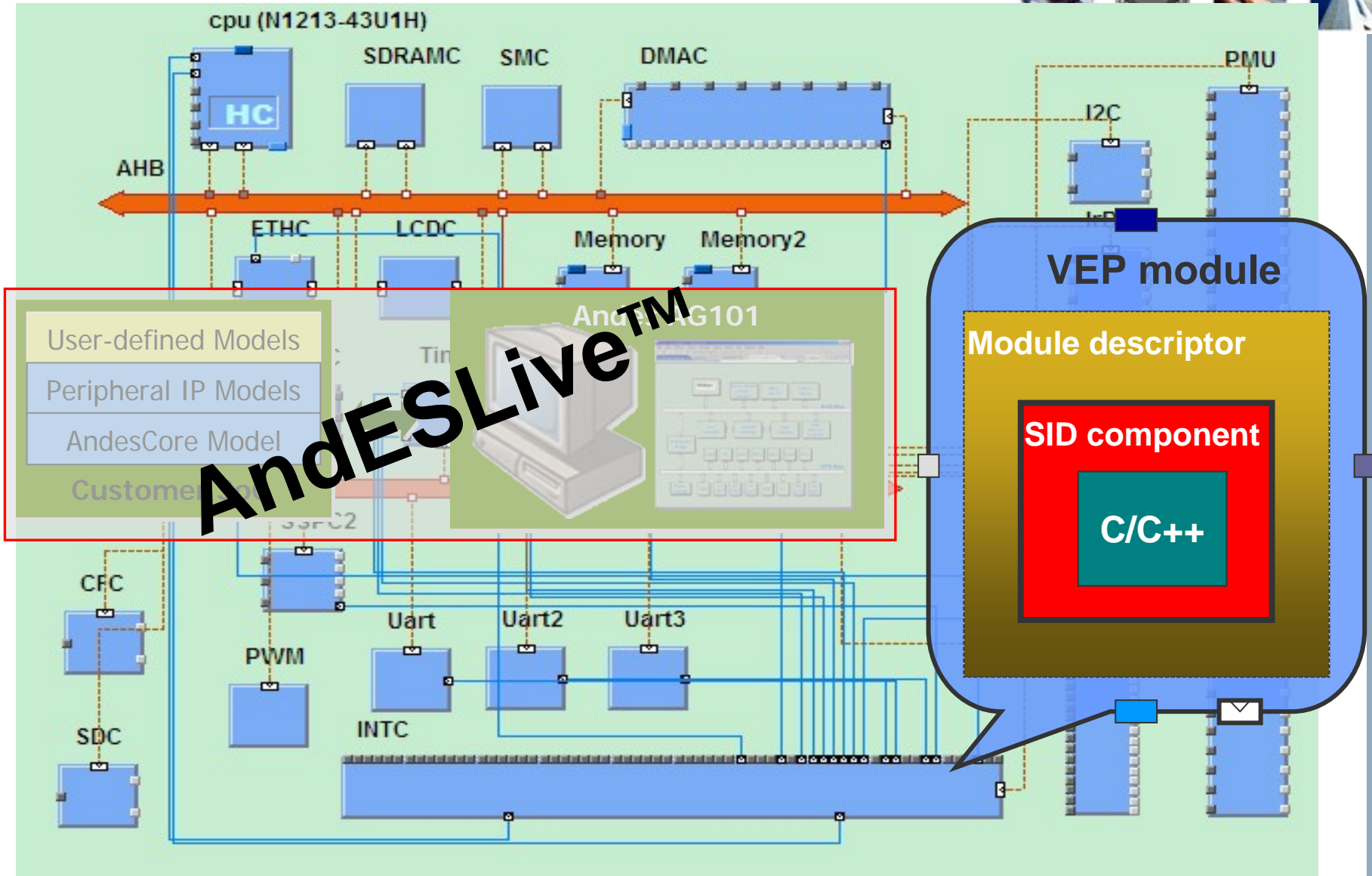
- Hardware Simulation (Main Window):** Shows a system architecture with a CPU (N1213-43U1H) connected to various peripherals via an AHB bus. Components include SDRAMC, SMC, DMAC, I2C, IrDA, Memory, Memory2, ETHC, LCDC, SSSP, Timer, RTC, and WDT.
- GPIO Window:** Provides a grid of pins for configuration.
- SDC (SDC) - AndESLive Window:** Manages SD card simulation. It shows options to load a default image (an empty 16MB SD card) or a custom image. The selected image file is `D:\ANDEST-1\ANDESTI-1\vep\image\SD\SD016.sdc`. It also displays card insertion status (Card remove selected) and information about FAT-16 formatting.
- Console (Uart3) - AndESLive Window:** Shows the system boot log and command-line output. The log includes messages for module loading, network registration, and file system mounting. The command `ls` is executed, showing the directory structure:
 

```

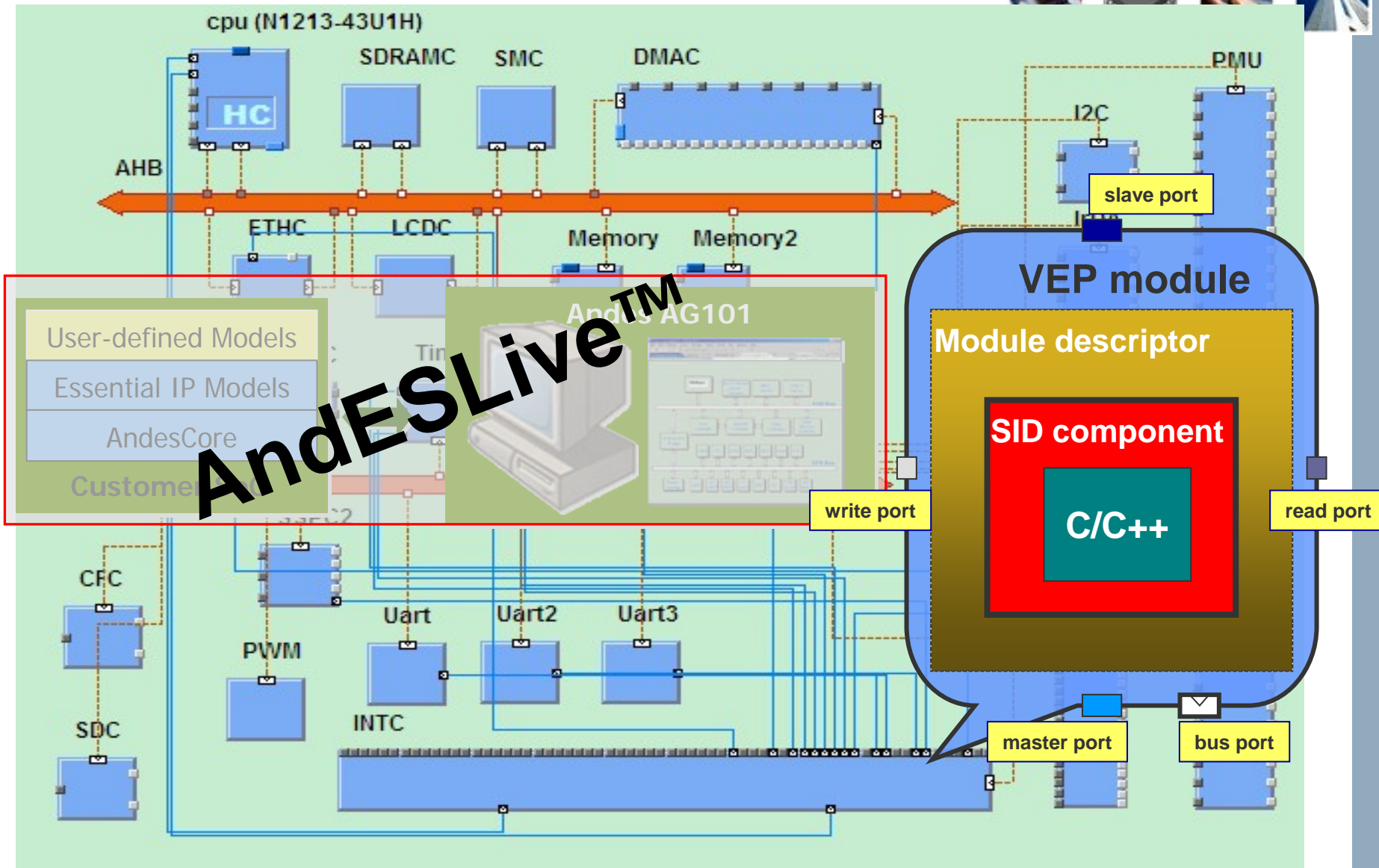
      bin      home      linuxrc   proc      tmp
      dev      lib       lost+found root      usr
      etc      lib64    mnt       sbin     var
      / $
      
```
- Control Registers:** A table of system registers with their current values:
 

Control registers	Intr state registers	Other registers
RTC enable: 0	rtc_sec: 0	RTC record: 0
intr sec: 0	rtc_min: 0	RTC divider En: 0
intr min: 0	rtc_hour: 0	RTC divider cycle: 32768
intr hour: 0	rtc_day: 0	RTC revision: 0
intr day: 0	rtc_alarm: 0	
intr alarm: 0		
counter load: 0		
Alarm values	Counter write port	
alr sec: 63	sec: 0	
alr min: 63	min: 0	
alr hour: 31	hour: 0	
	day: 0	

# AndESLive™ VEP Environment



# VEP Module



# Andes VEP (a quick summary)



- ❖ **SID**, an open-source framework for building simulated Embedded Systems, has been integrated into **AndESLive** as backbone simulator
- ❖ Simulated component, or a *SID component*, can be written in **C/C++**, or Tcl to which the **SID API** is bound
- ❖ *VEP module* is a SID component *wrapped* with **XML**-based **Module Descriptor** in which the parameters and attributes are described (and can be changed)
- ❖ Andes provides sample code (C++-based) and SID example for modeling **target** (*bus slave*) and **initiator** (*bus master*) components that run on **AndESLive**
- ❖ Depending on the requirements from customers, Andes can provide **Modeling Training** and **Services** as well

# SID Simulator

AndESLive™ simulation backbone



# SID Overview



- ❖ The **SID simulator** consists of an engine that *loads* and *connects* simulated components, based on a simulator configuration file, and runs simulation sessions.
- ❖ SID comes with a number of simulated components, or **SID components**, each of which can be modified, configured, or connected to any other independently.
- ❖ The SID simulator **configuration file** (or command file) is a text file that configures a SID simulation run. The configuration file defines the simulation *contents* (which components are used?), *connections* (how the components connect together?), and *initial states* (properties).
- ❖ Adding *new* components is straightforward and does not require any modifications to SID. More info on **SID Component Library**, refer to the *SID Simulator Component Developer's Guide*.
- ❖ While running a simulation, SID can interface with standard I/O, such as a Tk-based visual simulation monitor, the `gdb` debugger, and the `gprof` profiler.





# SID Architecture (1/2)



- ❖ SID is a simulation framework for supporting embedded systems software development.
- ❖ SID features a modular architecture of loosely-coupled software components that interact with each other to simulate the behavior of physical hardware parts.
- ❖ SID components share a fixed low-level **API**, which defines all possible *inter-component* communication mechanisms.
- ❖ SID is packaged as a standalone command-line program that reads and executes a configuration file (ie, to run a simulation.)
- ❖ A typical session with SID begins with compiling or assembling code for the simulated system to run (using standard cross-development toolchains), and proceeds through loading the target binary into the simulation environment.

# SID Architecture (2/2)



- ❖ Four Component types are supported in SID
  - Hardware model (**hw**-xxx)
  - Software model (**sw**-xxx)
  - Bridge (Tcl/Tk bridge)
  - Special function (event scheduler, host network interface, etc)
  
- ❖ Communication mechanisms between Components:
  - **SID API** is used to model these mechanisms:
    - pin and bus
    - attribute and relation
  
- ❖ The SID API can be thought as the *socket* on a circuit board. The SID Component is like the IC that plugs into these sockets and the SID simulator configuration file is like the *circuit wiring* that connects the sockets to each other



# SID API (1/2)



## ❖ `sid::component` Interface

### ■ Attributes

- `vector<string> attribute_names(); // return attributes for a component`
- `string attribute_value (const string& name); // query`
- `string set_attribute_value(const string& name, const string& value);`

### ■ Pins

- `vector<string> pin_names(); // return pins for a component`
- `pin* find_pin (const string& name); // input pins`
- `status connect_pin (const string& name, pin *pin); // out -> in`
- `Status disconnect_pin (const string& name, pin *pin);`

### ■ Buses

- `vector<string> bus_names(); //return buses for a component`
- `vector<string> accessor_names();`
- `bus* find_bus (const string& name); // bus slave`
- `status connect_accessor (const string& name, bus *bus); // bus master`
- `Status disconnect_accessor (const string& name, bus *bus);`

### ■ Relationships

- `vector<string> relationship_names ();`
- `status relate (const string& name, component *comp);`
- `status unrelate (const string& name, component *comp);`

# SID API (2/2)



## ❖ `sid::bus` Interface

- All bus transactions (read or write) are initiated by a bus master, and respond with a status code object
- Read (byte addressable, 1, 2, 4, 8, little and big order)
  - `status read (host_int_4 addr, little_int_1& data);`
- Write (byte addressable, 1, 2, 4, 8, little and big order)
  - `status write (host_int_4 addr, little_in_1 data);`
- Status codes
  - `return status (); // status = ok`
  - `return status (unmapped);`
  - `Return status (unpermitted);`
  - `return status (ok, 5); // latency = 5`
  - `return status (not_found);`
  - `return status (misaligned);`

## ❖ `sid::pin` Interface

- Only the API for input pins is provided
  - `void driven(host_int_4 value);`



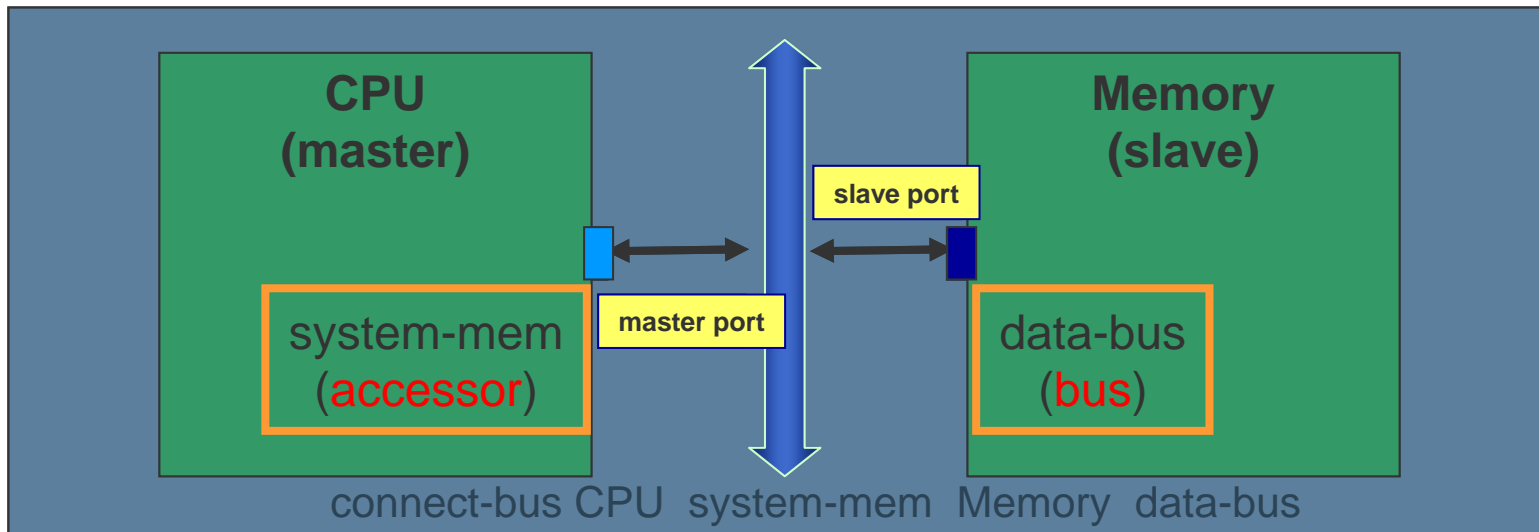
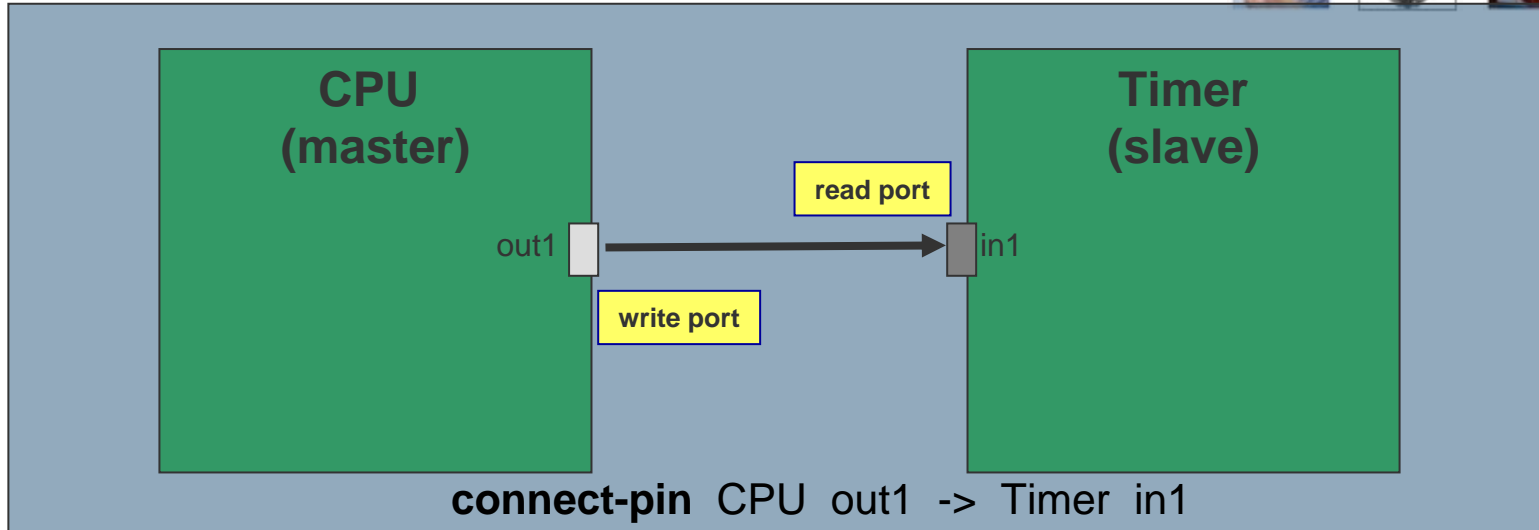
# SID Configuration File



- ❖ The configuration file is composed of commands which map to a small number of SID **API** calls
- ❖ The configuration file consists of three major sections:
  - A listing of component libraries to be loaded (dynamically loaded libraries)
    - load
  - A command to instantiate components
    - new
  - A set of commands that connect and configure the components
    - set
    - connect-pin, disconnect-pin (point-to-point)
    - connect-bus, disconnect-bus (broadcast)
    - relate, unrelate



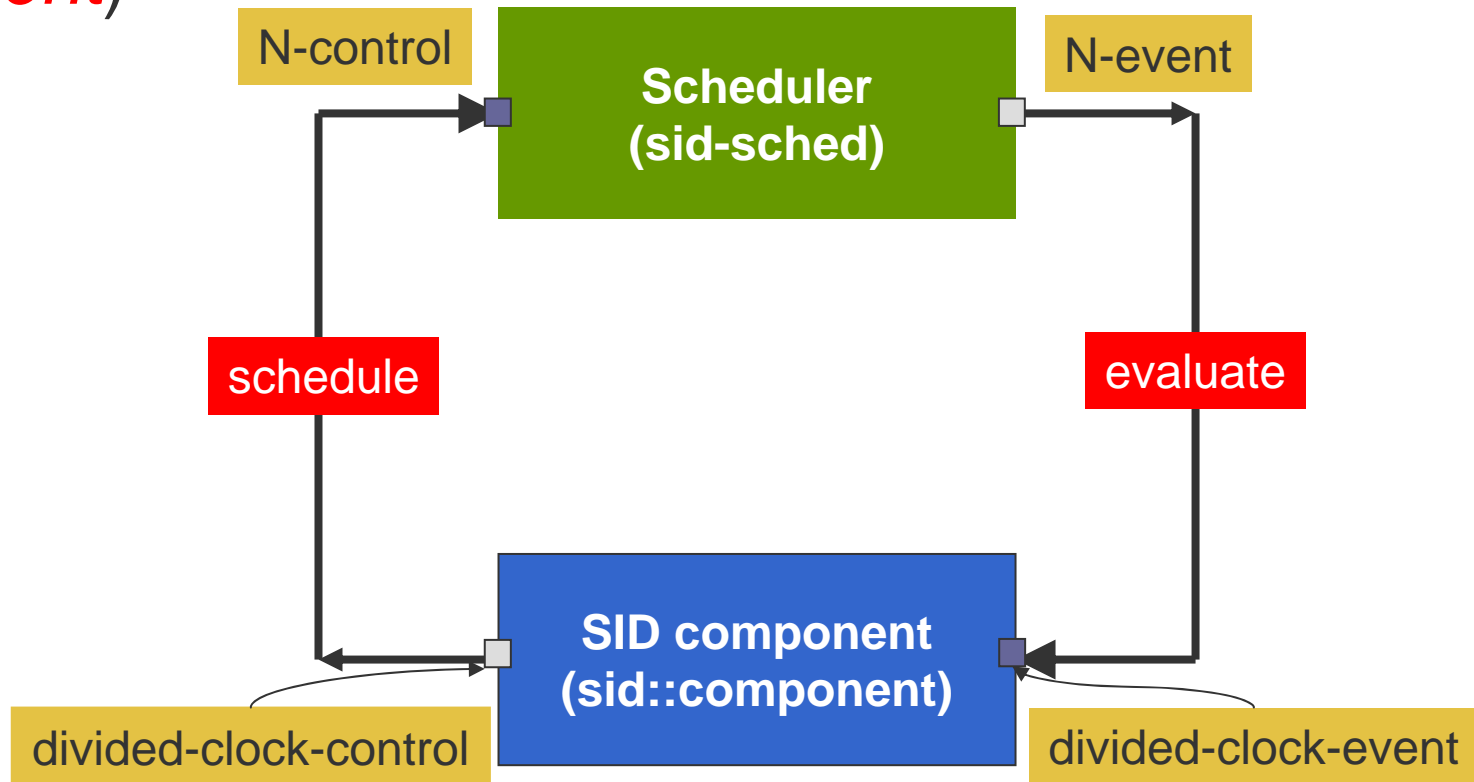
# Component Connection in SID



# Event Scheduling in SID



- ❖ Events in a SID component are scheduled by communicating to the **scheduler** using special pin interface (*divided-clock-control* and *divided-clock-event*)



# SID Component Modeling

User-defined Components





# Basic Component Outline (Class Declaration)



```
#include <sidcomp.h>
#include <sidtypes.h>
#include <iostream>
```

SID and C++ header files

```
using namespace sid;
using namespace sidutil;
```

Namespace

```
class sp_timer : public virtual component
{
public: sp_timer();
      ~sp_timer() throw() {}
```

Declare this class as SID component and use other predefined utilities

```
protected:
    input_pin a, b, c;
    output_pin d, e, f;
```

Data I/O

```
private:
    in_out_handler();
};
```

Inaccessible data and function

# Component Declaration (1/3)

## common header files & utility



### ❖ Header files

`sidattrutil.h`

`sidbusutil.h`

`sidcomp.h`

`sidcomputil.h`

`sidmiscutil.h`

`sidpinattrutil.h`

`sidpinutil.h`

`sidscheutil.h`

`sidtypes.h`

`sidwatchutil.h`

### ❖ Tens of utilities

#### ❖ For simplicity issue

```
using namespace sid;
```

```
using namespace sidutil;
```

# Component Declaration (2/3)

## class inheritance



```
class sp_timer
:public virtual component
    //each component class need to inherit from this
, public fixed_attribute_map_component
    //if the component provide configure attribute such as “verbose?” else use
    //“no_attribute_map_component”
, public fixed_pin_map_component
    //if the component provide input/output pin else use “no_pin_map_component”
, public fixed_bus_component
    //if the component provide bus access else use “no_bus_map_component”
, public no_relation_component
    //no relation utility requirement
, public no_accessor_component
    //not a bus master
```

# Component Declaration (3/3)

## Data I/O



```
//input type
input_pin din_pin;

friend class callback_pin<sp_timer>;
callback_pin<sp_timer> rst_pin;

//output type
output_pin intr_pin;

//clock type (connected to scheduler)
friend class scheduler_event_subscription<sp_timer>;
scheduler_event_subscription<sp_timer> clk;
```

# Integrating SID Component in AndESLive

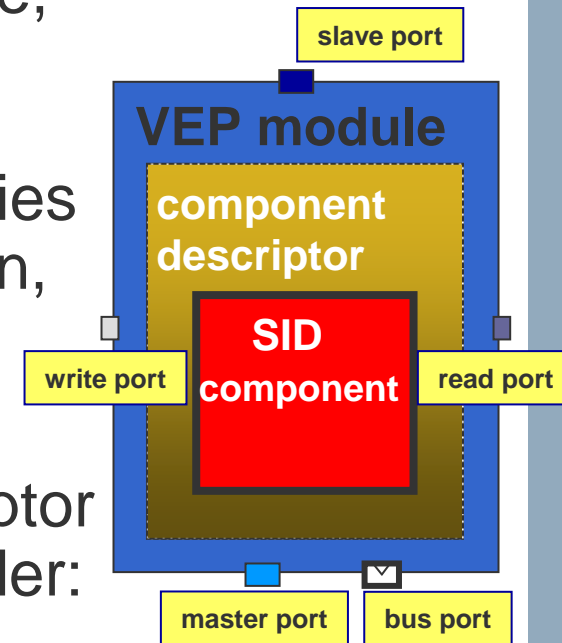
Wrapping SID component with Module Descriptor



# Integrating Component in AndESLive™



- ❖ To integrate SID component in AndESLive, an XML-based **component descriptor** is created for each component. The component descriptor defines the properties to be used in AndESLive (such as bus, pin, and attributes)
- ❖ Once the user-defined component descriptor is completed, save the XML file in the folder:
  - `$ANDESIGHT_ROOT/vep/component/user`
- ❖ Under this directory, a sample component descriptor, `sample.comp.xml`, is provided for reference



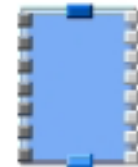
# Sample Component Descriptor (1/2)



## ❖ Component Definition:

- `<defcomponent name="hw-sample" shortname="sample" type="SID">`
- `<sid-lib dlsym="sample_component_library" name="libsampla.lib" />`
- `</defcomponent>`

sample



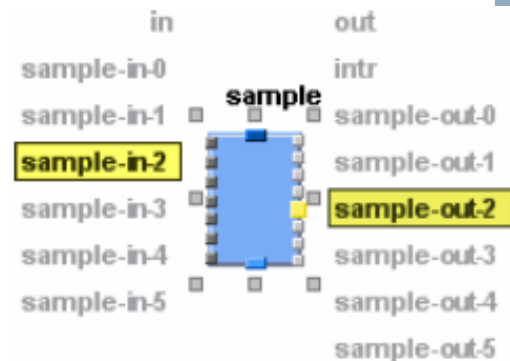
## ❖ Bus Definition:

- `<!--sid bus accessor-- >`
- `<busmaster name="master" type="AHB"/>`
- `<!--sid bus interface-- >`
- `<buslave name="registers" type="AHB"/>`



## ❖ Pin Definition:

- `<!--pin-- >`
- `<defpins name="sample-out-" from="0" to="5" direction="out"/>`
- `<defpins name="sample-in-" from="0" to="5" direction="in"/>`
- `<defpin name="out" direction="out"/>`
- `<defpin name="in" direction="in"/>`
- `<defpin name="intr" direction="out" type="interrupt"/>`

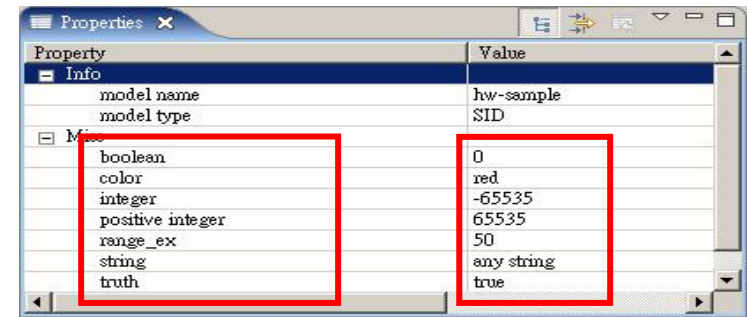


# Sample Component Descriptor (2/2)



## ❖ Attribute Definition:

- `<!--sample attributes-- >`
- `<defattribute category="setting" name="string" type="string" default="any string"/>`
- `<defattribute category="setting" name="boolean" type="{0, 1}" default="0"/>`
- `<defattribute category="setting" name="truth" type="{true, false}" default="true"/>`
- `<defattribute category="setting" name="integer" type="integer" default="-65536"/>`
- `<defattribute category="setting" name="range_ex" type="[0..100]" default="50"/>`



Property	Value
Info	
model name	hw-sample
model type	SID
Misc	
boolean	0
color	red
integer	-65535
positive integer	65535
range_ex	50
string	any string
truth	true

## ❖ Schedule Event and Time:

- Clock definition:
  - `<defclockinput name="clockIn" event="event" control="control" />`
- Time Query definition:
  - `<time-query-client query="time-query" high="time-high" low="time-low" />`

## ❖ Life Cycle Control Pin:

- `<!--to initialize, de-initialize, or reset component-- >`
  - `<control-pin init="init" deinit="deinit" reset="reset" />`



# VEP component (sample)

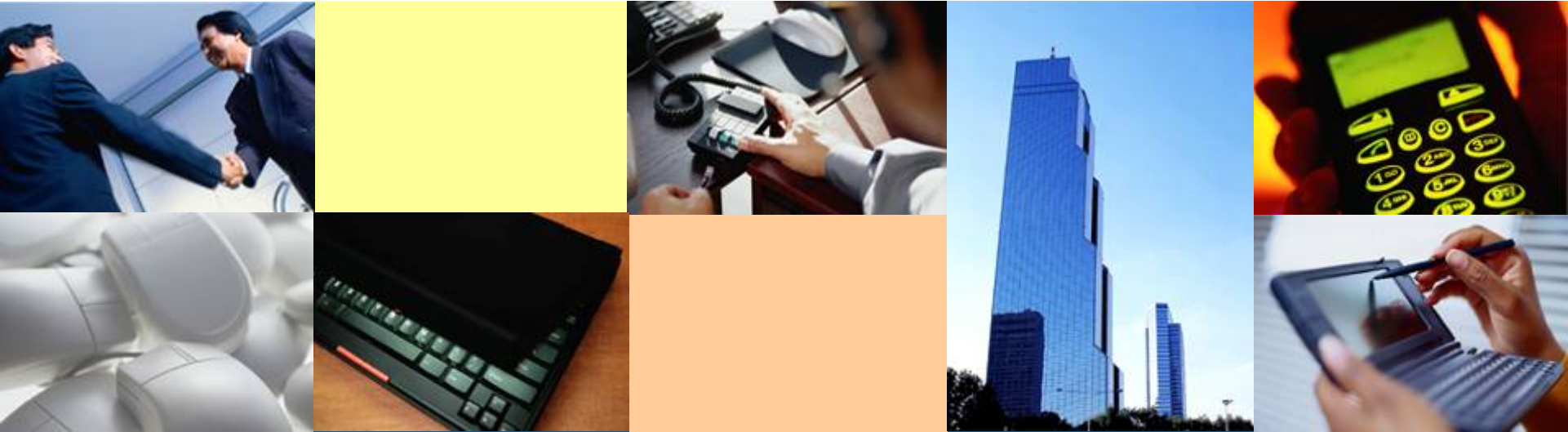


The screenshot displays the ANDES MP3 Decoder software interface. The main workspace shows a schematic diagram with a central orange AHB bus. On the left, a 'cpu (N903A-S)' and 'Memory' are connected to the bus. A 'sample' component is connected to the bus and is highlighted with a red box. The 'Palette' on the right lists various components like CPU, BUS, AHB, and Memory. The 'Outli...' pane shows a tree view of the project components.

Below the workspace, the 'Memory Map' tab is active, showing a table of memory components:

Device Name	Base	Size (Byte)	Range
AHB			
Memory	0x00000000	0x02000000	0x00000000 - 0x01FFFFFF
sample	0x98000000	0x00000100	0x98000000 - 0x980000FF

# Questions/Answers



# Questions



- ❖ We want to know how a component is modeled in AndESLive and what language/description by which the modeling is based upon?
- ❖ Where can we find the document/tutorial for modeling user-defined components and if Andes can provide modeling training/services?
- ❖ Will Andes support SystemC as a modeling constructor and what if we have SystemC models, how can we incorporate them into AndESLive?

# Answers



- ❖ The behavior of user-defined model can be written in C/C++ and the SID API is used to compose a SID component by which a VEP-based module is created that runs on AndESLive.
- ❖ There is a document in User Manual (release 1.3.1) for creating a user-defined model. Tutorial is also being prepared as well as application notes. Andes can provide modeling training and/or services based on customer requests.
- ❖ In the current release (1.3.3), SystemC modeling/import is NOT supported in AndESLive. Andes is now developing SID-SystemC *bridge* which can communicate SystemC interface with SID-based pin and bus. Untimed SystemC models will be supported first.

# SystemC Modeling



# How VEP can be used in Development Cycle

Enable early S/W development



# VEP Use Models



- ❖ VEP as an **Early (or pre-silicon) Software Development and Software Validation Platform**
  - Reduce SW bring-up and system test time
  - Ideally start SW dev. in parallel to HW dev.
  - Leave more time for SW dev. and quality assurance
  
- ❖ VEP as an **Architecture Exploration Platform**
  - Evaluate HW/SW configuration and/or system partitioning
  - Optimize system architecture
  
- ❖ VEP as a **RTL Verification Platform**
  - Golden reference models for functional verification
  - Verify architecture and system validation

# VEP as Early SW Development & Validation



- ❖ VEP can be used for developing & testing SW  
*(only if the VEP can run as fast as HW board does)*
  - Low-level device drivers and kernel
  - OS and middleware porting
  - App. SW development
  
- ❖ One scenario is as follows:
  - **SW team** extends existing device drivers based on updated IP spec
  - At the same time, **Platform team** enhances/creates models (ex. new features added) based on the VEP
  - SW team completely **debugs** and **tests** the driver functionality early on the VEP
  - SW 'bring-up' in a shorter time after the HW (FPGA) is available



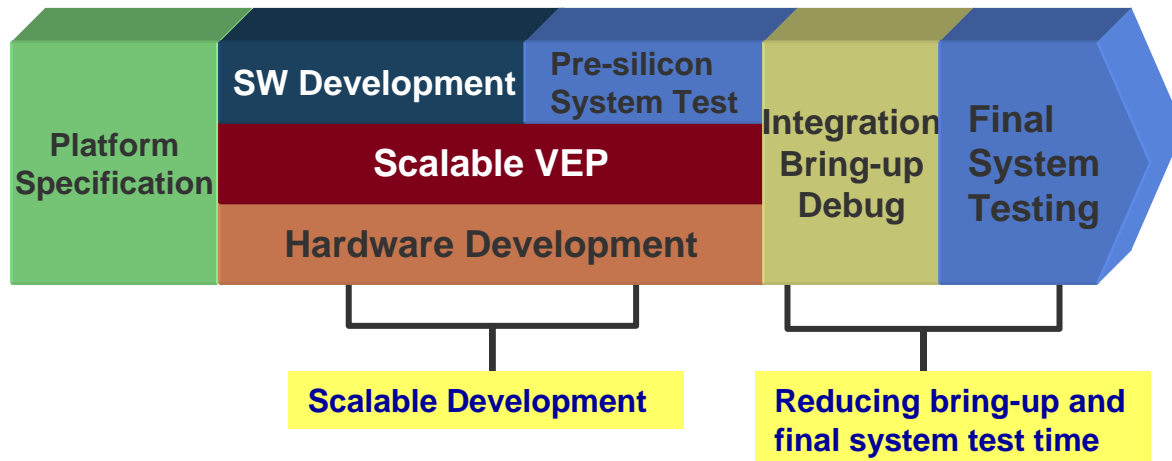
# Development Cycle Impact



## Traditional (or Past) Approach:



## VEP-based Approach:



- *Enable Early SW Development*
- *Enable Scalable Development*
- *Pre-Silicon System Test*
- *Reduce Bring-up time*
- *Reduce Post-Silicon System Test*

# VEP as Architecture Exploration



- ❖ VEP can be used to explore design **alternatives** to determine the *appropriate* architecture (or system)
  
- ❖ Some alternatives to evaluate are:
  - AndesCore configuration
    - Cache, MMU, Local memory, branch prediction, etc
  - SW profiling (in AndESLive)
    - code optimization
  - HW/SW partitioning
    - HW accelerator engine or SW oriented
    - Performance and cost tradeoff
  
  - Bus Matrix or Multi-layer
    - currently NOT supported in AndESLive

# VEP as RTL Verification



- ❖ Currently NOT supported in AndESLive!
  
- ❖ Team-up Partnership possibilities
  - GUC (porting AndesCore ISS with SystemC TLM2.0)
  - EVE (compiling AndesCore on ZeBu)
  - CoWare (porting AndesCore ISS with SystemC/AMBA)
  - Carbon (porting AndesCore ISS with CASI/SystemC)
  - Cadence/Synopsys/Mentor...
  
- ❖ *Speed* is always an important concern and incentive
  - Transaction-based interface is the key
  - HW accelerator/emulator may be too expensive to be justified

# THANK YOU!



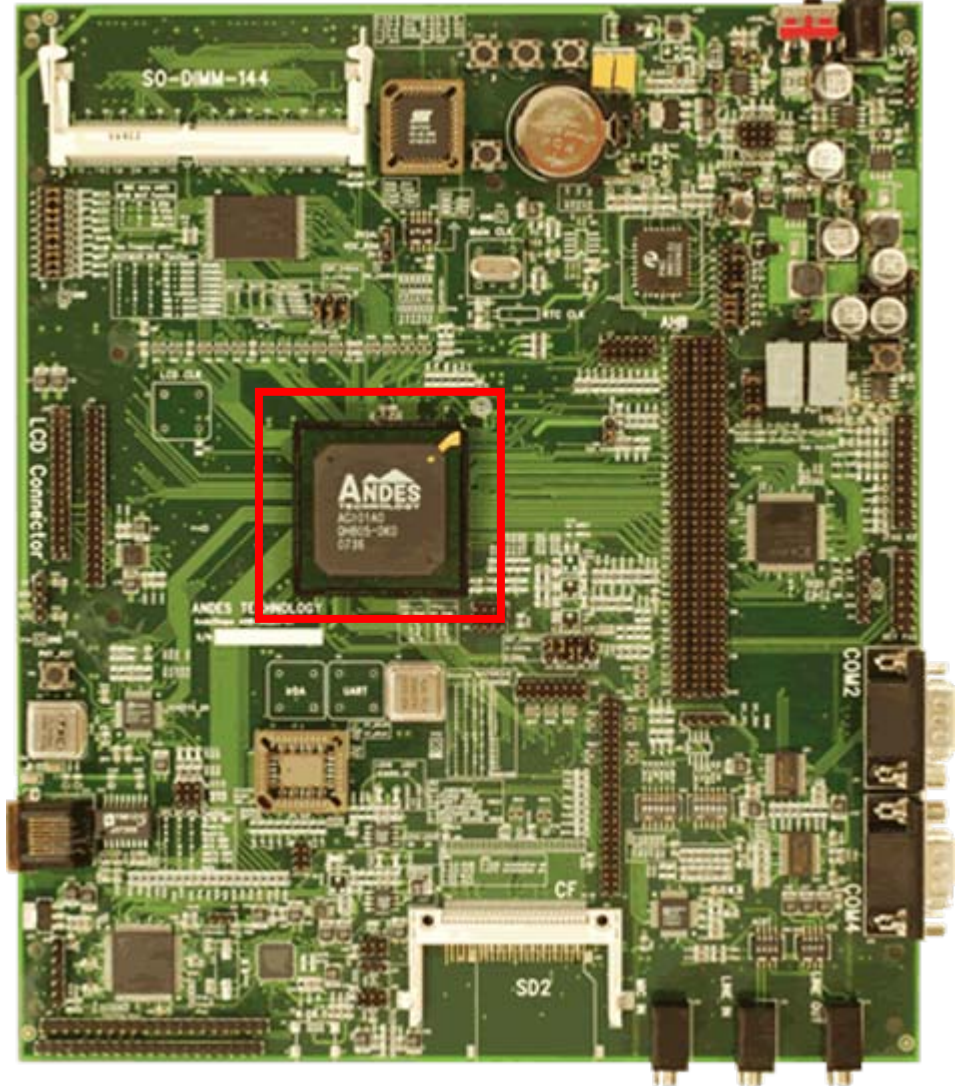
# Supplement Slides



# Andes ADP-AG101 Features



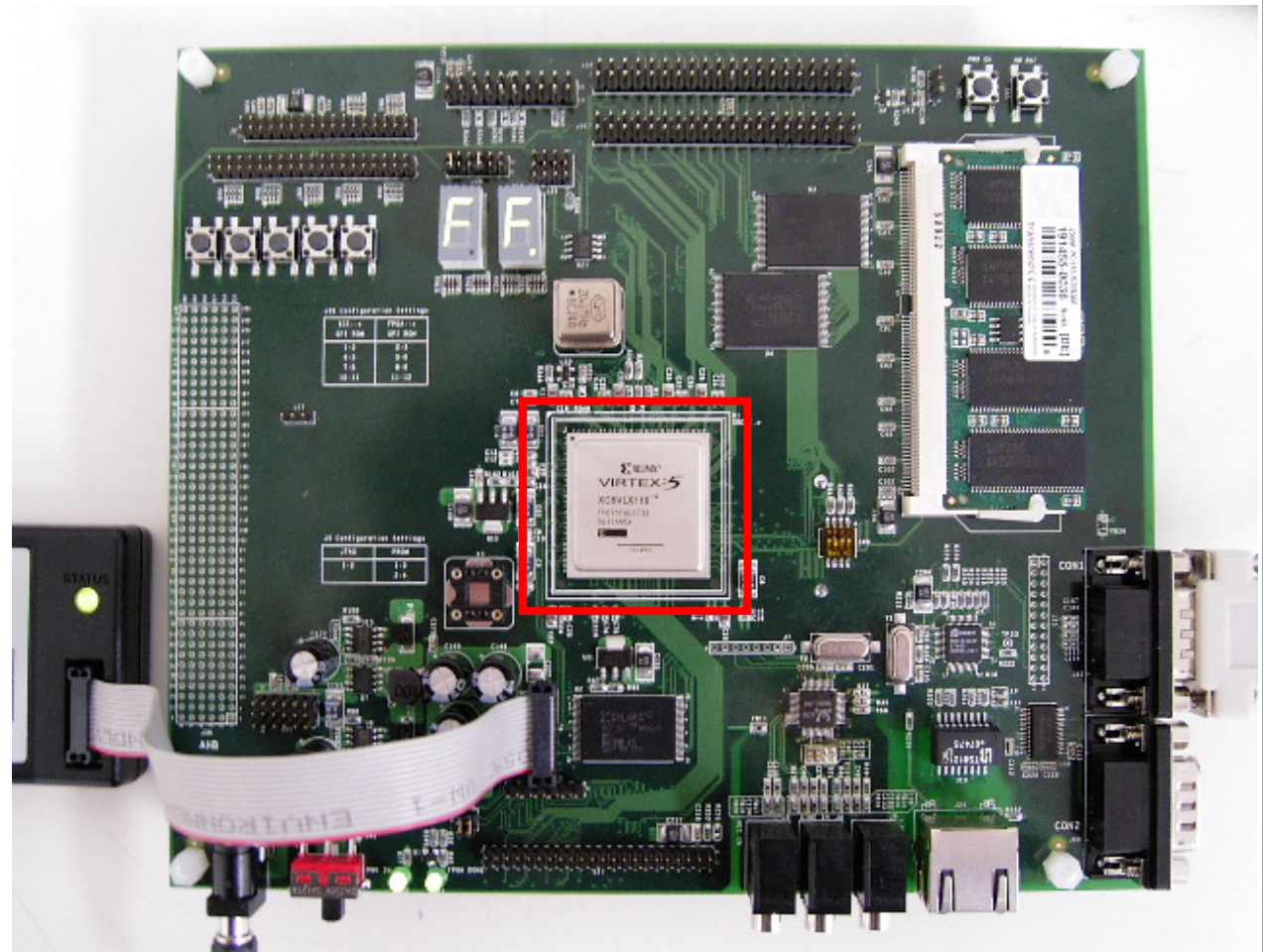
- ❖ AMBA 2.0 based AHB environment for new IP verification
- ❖ High integration flexibility for bus extension
- ❖ On-board 64MB SDRAM with one SODIMM slot
- ❖ On-board 512KB Boot ROM with 32MB flash memory
- ❖ On-board header for LCD plus touch screen module
- ❖ 15 on-board GPIO keypads
- ❖ On-board 10/100 PHY and RJ45 connector
- ❖ On-board T&MT interface connector for USB 2.0 PHY
- ❖ On-board CF & SD slot, providing CF & SD/MMC card access
- ❖ On-board AC97 & I2S codec, selected by switch
- ❖ 2 on-board UART interfaces with level-shifting on board for asynchronous serial data transfer
- ❖ On-board Andes ICE interface for function debugging



# Andes ADP-**XC5** Features



- ❖ Devices on AHB
  - SDRAM controller
  - LCD controller
  - DMA controller
  - MAC controller
  - USB 2.0 controller
  - SRAM controller
  - Flash controller
  - AHB Bus controller
- ❖ Devices on APB
  - AHB-APB bridge
  - PMU
  - PWM
  - I<sup>2</sup>C
  - GPIO
  - Interrupt controller
  - Watch dog timer
  - Timer
  - RTC
  - UART
  - SSP
  - CF controller
  - I<sup>2</sup>S/AC97
  - SD/MMC



# Virtual Platforms Authoring Tools



## ESL Model and Virtual Platform Creation

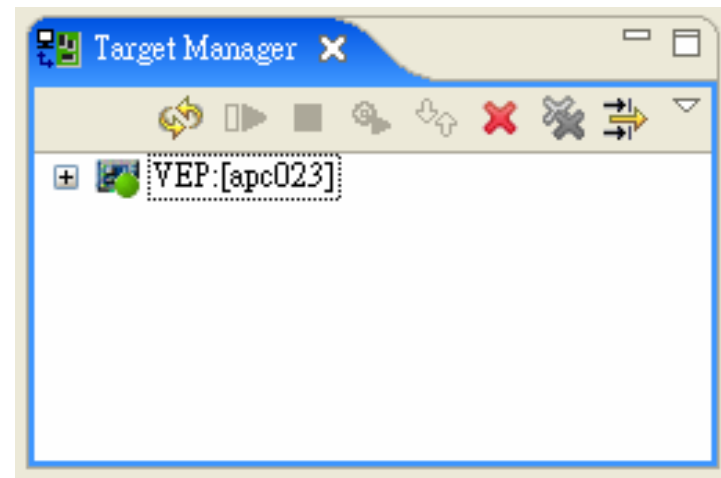
	SystemC support	Pre-assembled platform	Tool-chain	Targeted CPU
IBM ChipBench SLD	Y	Y	GNU-based	PPC 405
ARM System Generator	Y	Y	native	ARM9,11, Cortex
MIPS ICS	Y	N	native	MIPS
Tensilica TurboXim	Y	N	native	Diamond
ARC VTOC & IP eXchange	Y	N	GNU-based	ARC 600
<b>Andes AndESLive</b>	Y (partial)	Y	GNU-based	AndesCore
Synopsys Innovator	Y	Y	GNU-based	ARM, etc
Mentor Graphics Visual Elite	Y	Y	GNU-based	ARM, etc
Carbon SOC Designer	Y	Y	GNU-based	ARM, etc
CoWare PA	Y	Y	GNU-based	many
Virtutech Simics	Y	Y	GNU-based	many



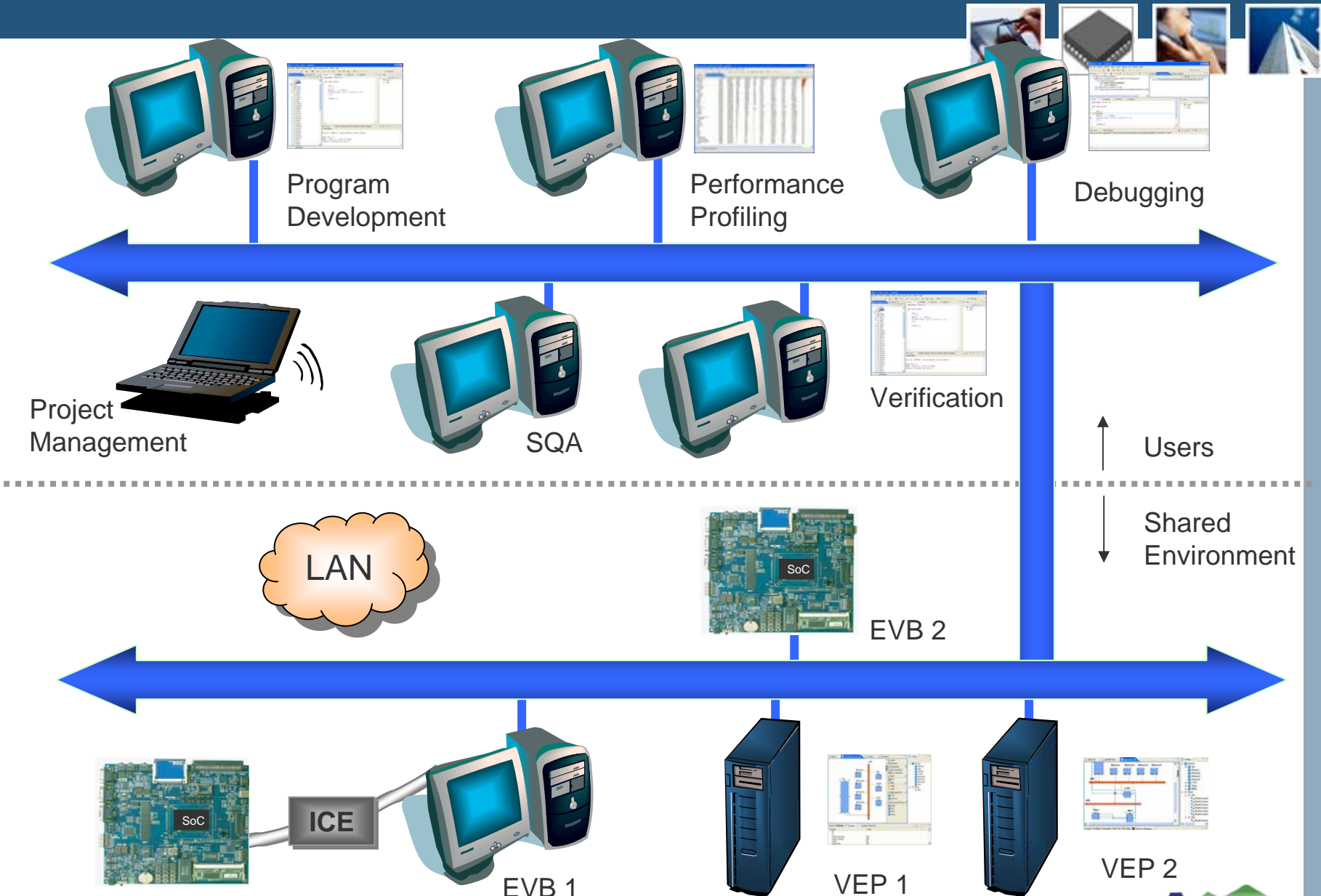
# Target Manager



- ❖ Allow application executables to run on a simulator or a real board (Target)
- ❖ All Targets are available on different places but within a LAN environment to allow sharing among all users easily
- ❖ All Targets are queried automatically to save the set up efforts



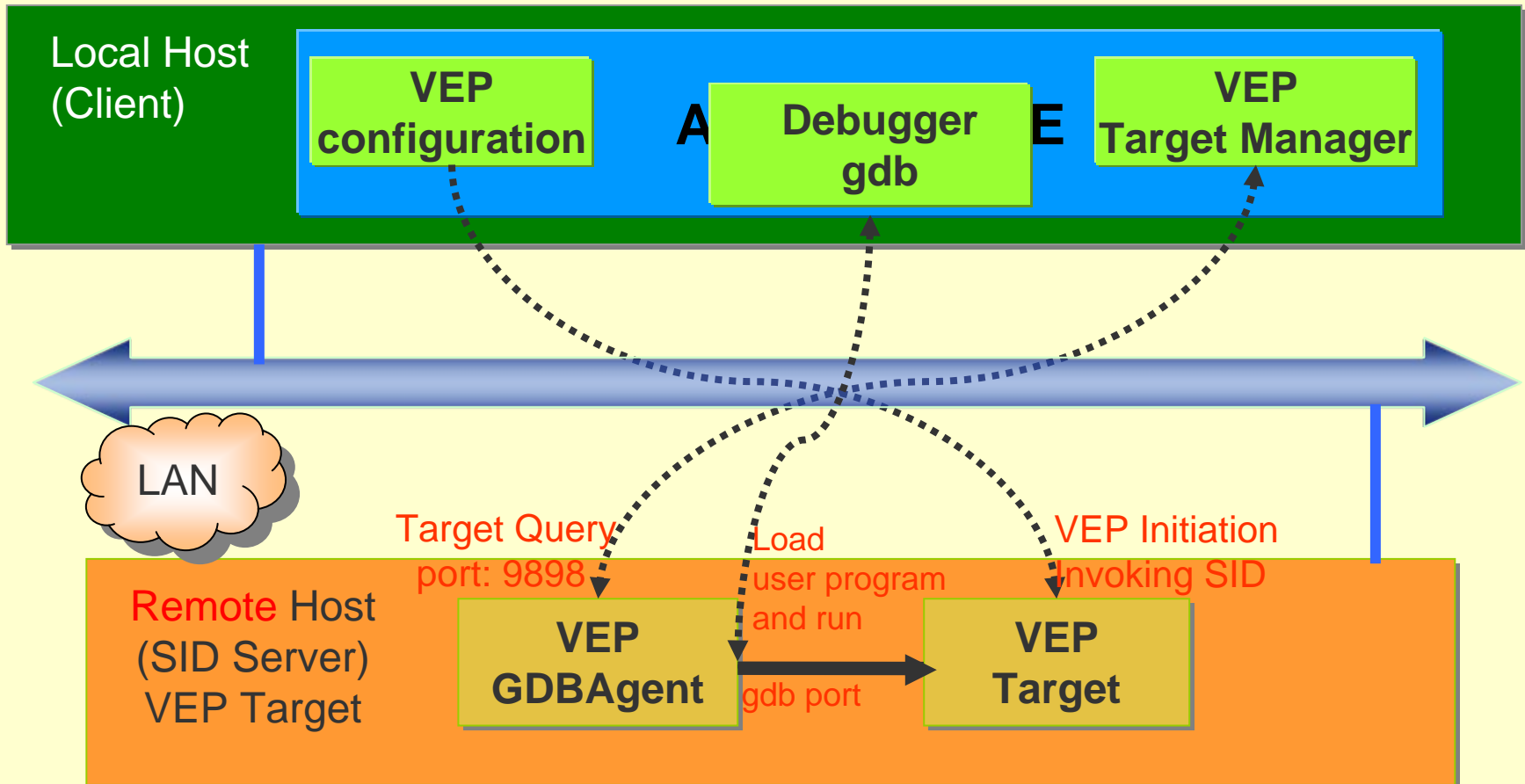
# Targets on networking



# (VEP) Target Manager



user  
↓



# Basic VEP Creation



- ❖ Create a basic VEP configuration file (vep file) for Project “Hello”
  - Right-click on Project “Hello”>New>New VEP Config>enter file name Hello.vep>Basic platform for N903>Finish → a basic VEP configuration (N903, Memory, and AHB) is created
  
- ❖ Invoke GDBAgent (can be on local or remote system)
  - `GDBAgent.exe -v none` (on Windows)
  - `GDBAgent -v none -d` (on Linux)
  
- ❖ Query VEP Target
  - Target>Query Target → available VEP target(s) are shown in Target Manager View
  
- ❖ Initiate VEP Target with the Hello VEP configuration
  - Expand Project “Hello”>click, drag, and drop “Hello.vep” onto an available VEP target
  - → VEP target is connected through the GDB agent
  - → Project “Hello” is now ready to run, debug, and profile

# Running Program through Command-line Mode



- ❖ Click the Hello.vep on Editor View
  - File>Export SID Config>type “Hello” for the output file->a SID configuration with filename Hello.conf is created
- ❖ On a Cygwin terminal (SID server)
  - `$sid.exe Hello.conf`
  - Look for the gdb port showing on the last line (server at 0.0.0.0:port)
- ❖ On another Cygwin terminal
  - `nds32le-elf-gdb.exe Hello.adx`
  - `(gdb)target remote :port*`
  - `(gdb)load`
  - `(gdb)list`
  - `(gdb)c`
  
  - `*target remote hostname:port (if on another machine)`

# Command-line Mode



```
File 另存 /cygdrive/d/Andestech/AndeSight132/toolchains/nds32le-elf-n903a-s-32gpr/bin 903a-s-
00976 hello! world!
00977 hello! world!
- as 00978 hello! world!
- as 00979 hello! world!
- re 00980 hello! world!
- st 00981 hello! world!
as 00982 hello! world!
j as 00983 hello! world!
$ as 00984 hello! world!
as 00985 hello! world!
j as 00986 hello! world!
$ as 00987 hello! world!
t as 00988 hello! world!
d as 00989 hello! world!
d RT 00990 hello! world!
- as 00991 hello! world!
- as 00992 hello! world!
- as 00993 hello! world!
d as 00994 hello! world!
d as 00995 hello! world!
- au 00996 hello! world!
- pi 00997 hello! world!
- so 00998 hello! world!
so 00999 hello! world!
j GD
$ Program exited normally.
(gdb)
```