

# Embedded System Design

## Modeling, Synthesis, Verification

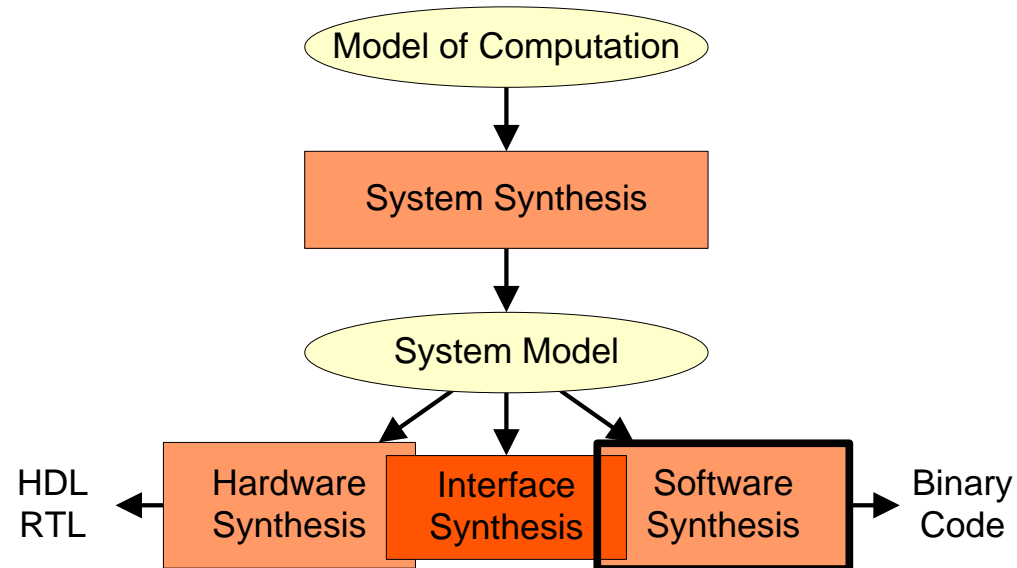
Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, Gunar Schirner



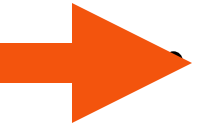
## Chapter 5: Software Synthesis

# SW Synthesis within Overall Design Flow

- **MoC**
  - Capture application
- **System Synthesis**
  - Define system wide decisions
- **System model**
  - Estimate / analyze system wide decisions
- **Component synthesis**
  - Generate component implementation
  - Software Synthesis  
Produce SW binary for exec on processor



# Outline



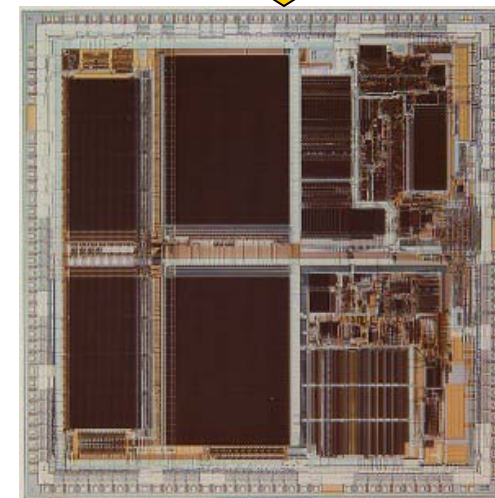
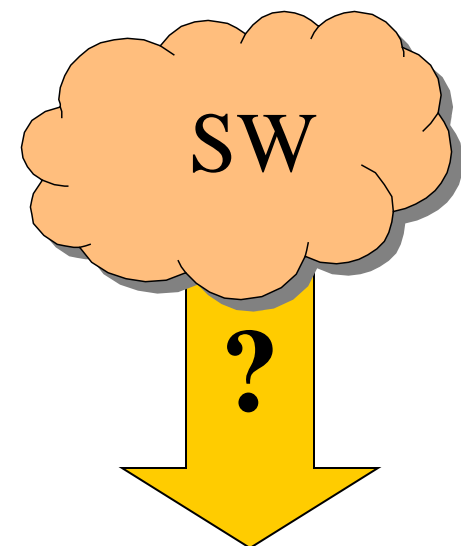
## **Introduction**

- Preliminaries
- Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



# Motivation

- **Increasing complexity of Multi-Processor System-on-Chip (MPSoCs)**
  - Feature demands
  - Production capabilities + Implementation freedom
- **Increasing software content**
  - Flexible solution
  - Addresses complexity
- **How to create SW for MPSoC?**
  - Avoid break in ESL flow:
    - Synthesize SW from abstract models

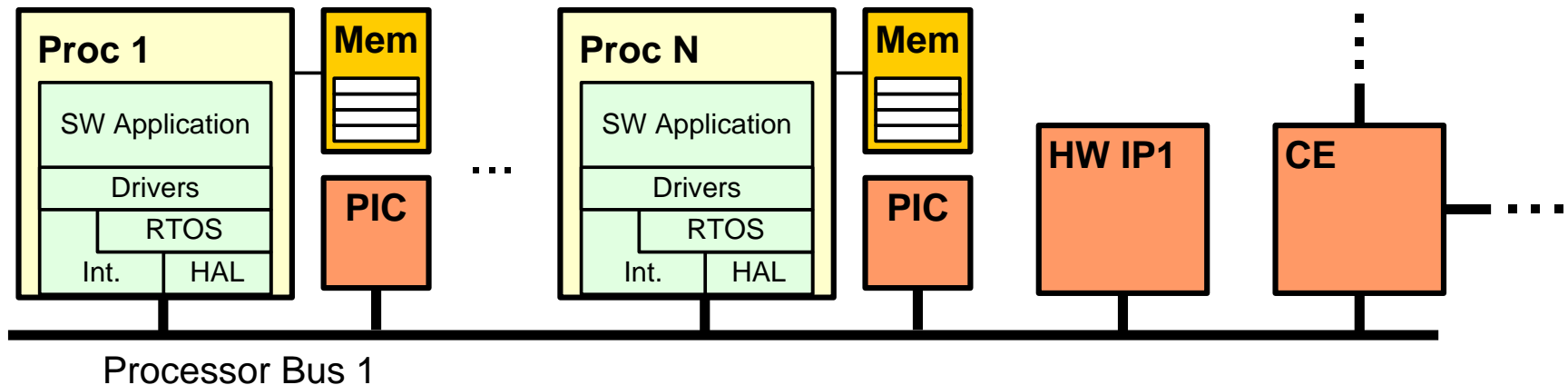


Source: [simh.trailing-edge.com](http://simh.trailing-edge.com)




# Goals

- **Generate SW binaries for MPSoC from abstract specification**
  - Eliminate tedious, error prone SW coding
  - Rapid exploration
  - High-level application development
  - Support wide range of system sizes
    - with RTOS / without RTOS (i.e. interrupt-driven)



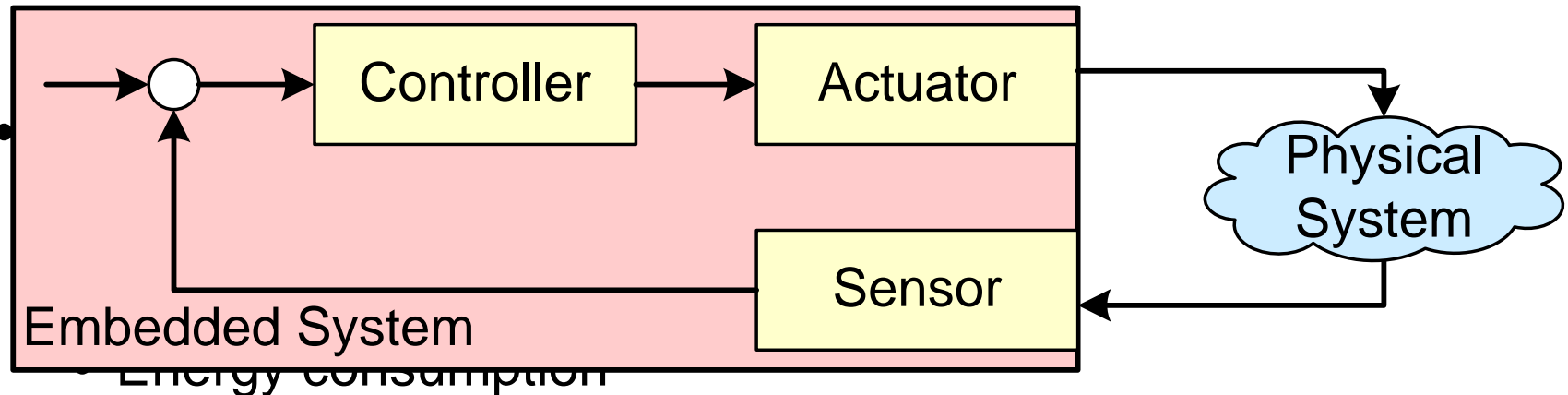
# Outline

- 
- **Introduction**
    - Preliminaries
    - Software Synthesis Overview
  - **Code Generation**
  - **Hardware-dependent Software**
    - Multi-Task Synthesis
    - Internal Communication
    - External Communication
    - Startup Code Generation
    - Binary Image Generation
  - **Execution**
  - **Summary**



# Embedded Software Challenges

- **SW tightly coupled**
  - Underlying HW (accelerator)
  - Part of a physical control loop
- **Time constrained**
  - Correctness depends on time frame
    - Too late functional correct answer still wrong
- **Concurrent**



# Example Target Languages

- **Assembly**
  - Basically a symbolic representation of machine code
    - Fine grained control: Registers, Instructions
- **C**
  - General purpose programming language, including level features
    - Bit operations, direct memory management
  - Low run-time overhead
  - Coding standards for portability
    - E.g. MISRA (automotive)
- **C++**
  - Backward compatible to C
  - Facilitates object oriented programming
- **Java**
  - Portability!
  - Hides memory management, pointer arithmetic
  - Interpreted (JVM), ahead of time and Just in Time (JIT) compiled
  - Java accelerators (e.g. ARM Jazelle), Java processors
  - Real-time extension: Real-time Specification for Java (RTSJ)





# Real-Time Operating System (RTOS)

- **Similar to general purpose operating system**
  - SW layer above HAL
  - Provides services for
    - Concurrent execution
    - Communication
    - Synchronization
    - Resource management
- **Real-Time Operating System**
  - **Facilitates** constructing real-time systems
  - Predictable response time
  - Hard real-time
    - Catastrophic failure on deadline miss
  - Soft real-time
    - Missing some deadlines tolerable



# Real-Time Operating System (RTOS)

- Characteristics of scheduling policies
  - Preemptive / Non-preemptive
  - Static / Dynamic
  - Off-line / On-line
  
- Example scheduling policies
  - Priority-based
  - Earliest Deadline First (EDF)
  - Rate Monotonic (RM)



# Outline

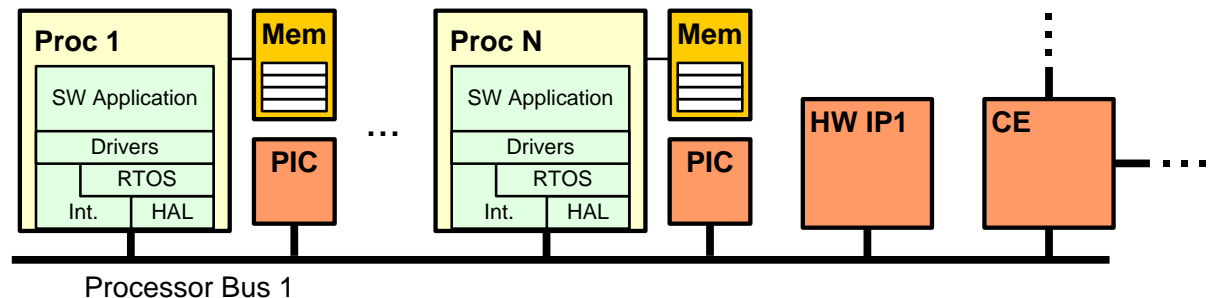
- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



# Software Synthesis Overview

- **Multi-core target platform**

- Processors
  - PIC, Timer, MEM
- HW IP
- Communication Element
  - E.g. Transducer
- Bus hierarchy



- **Generate for each core**

- Application
- Drivers to communicate with
  - Other cores
  - Accelerator IP
  - Communication Elements
  - External Memory



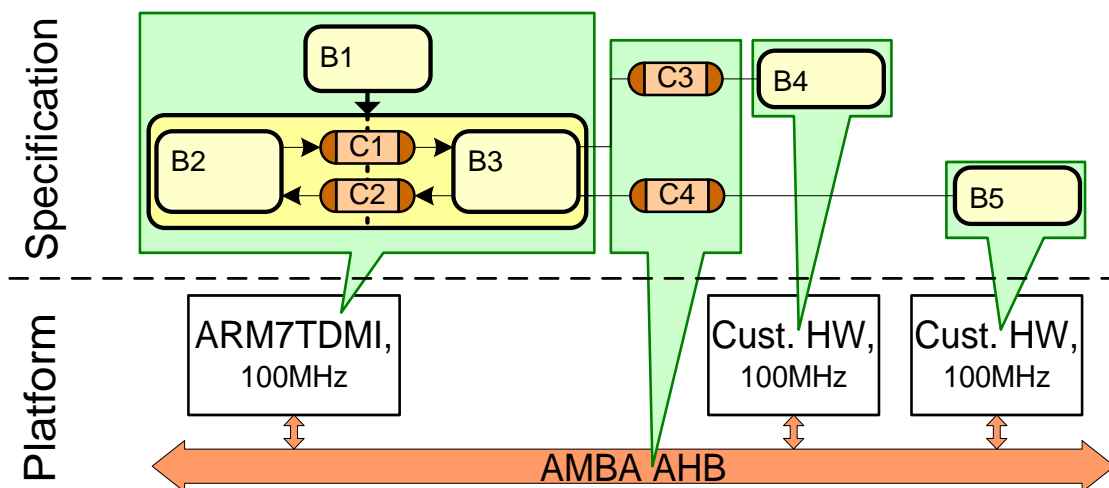
# System Design Flow Overview

## Input specification (review)

- **Capture Application in MoC, then SLDL**
  - Computation
    - Organize code in behaviors (processes)
  - Communication
    - Point-to-point channels, feature-rich
      - » Synchronous / Asynchronous
      - » Blocking / Non-Blocking
      - » Synchronization only (e.g. semaphore, mutex, barrier)

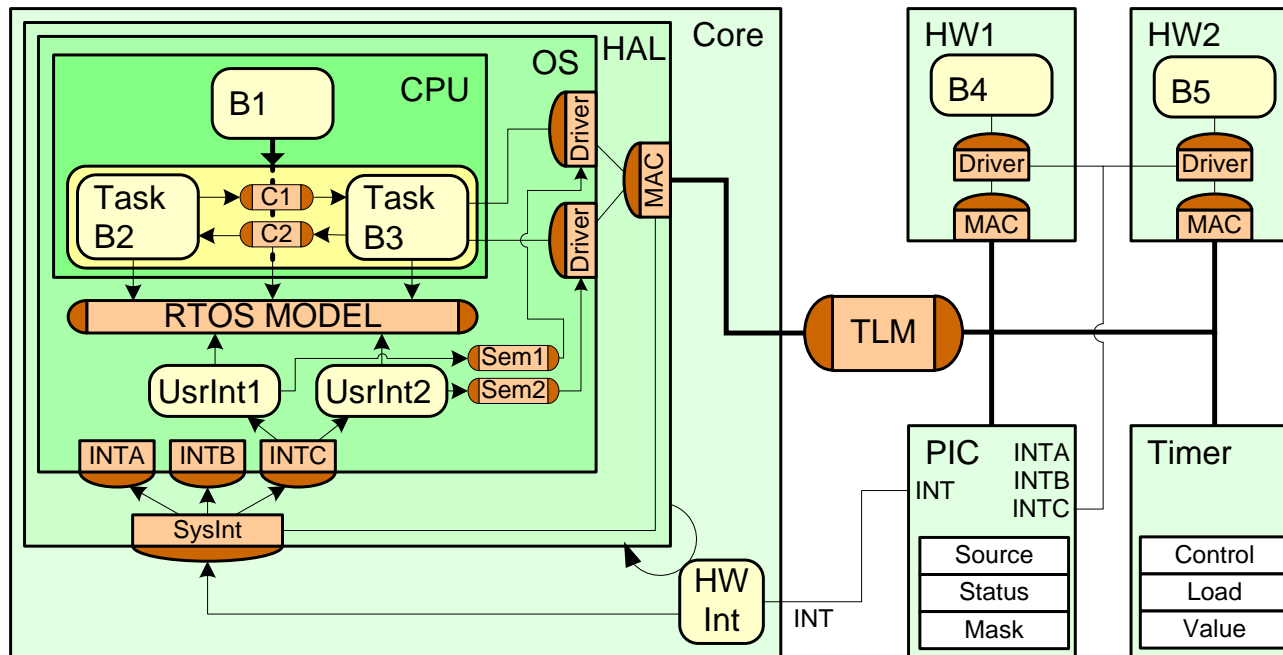
- **Architecture decisions:**

- Processor(s)
- HW component(s)
- Busses
- Mapping
- ...



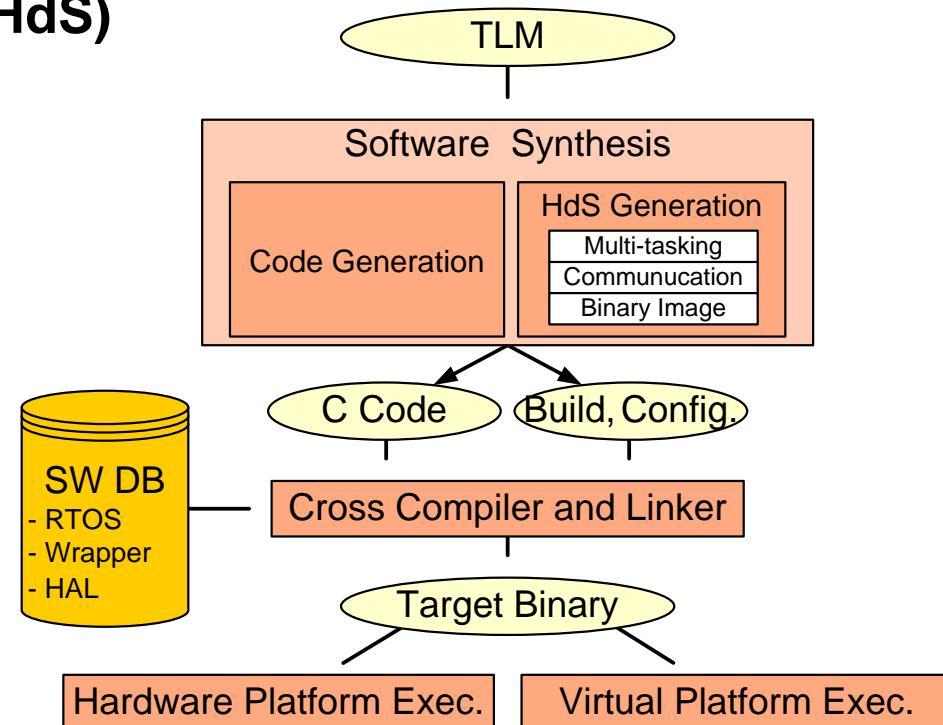
# System Design Flow Overview

- **System synthesis output: system TLM**
  - Functional timed abstract model
  - Reflects system-wide decisions



# Software Synthesis Overview

- **Code Generation**
  - Generate application code inside each task
  - Resolve behavioral hierarchy into flat C code
- **Hardware-dependent Software (HdS) Generation**
  - Multi-task Generation
    - Tasks -> RTOS
  - Communication Synthesis
    - Drivers
  - Binary Image Generation
    - Cross Compile and Link
- **Binary executable on**
  - Actual Hardware Platform
  - Virtual Platform with integrated ISS



# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**





# Code Generation

- **Rational:**
  - TLM in SLDL (e.g. SystemC, SpecC)
  - SLDL system level optimized, not target
    - Generic for any architecture
      - » Hierarchical composition of concurrent modules
      - » Connectivity between modules
      - » Communication encapsulation
      - » Hardware concepts
  - Inefficient direct compilation to target
    - Large simulation kernel on target
  - Need efficient (footprint, memory, speed) target implementation
  - Translate system model to target language
    - SystemC -> ANSI-C



# Code Generation

- **Generate sequential code executing in a task**
  - Remove SLDL specific concepts
  - Resolve hierarchy and connectivity into flat C code
    - Set of C *structs*: Behaviors / channels
      - » Member variables + Port connectivity
    - Global functions + context pointer (\*pThis)

## Rules for C code generation

### 1. **Module resolution**

Behaviors and channels are converted into C *struct*

### 2. **Hierarchy translation**

Child behaviors and channels are instantiated as C *struct* members inside the parent C *struct*

### 3. **Variable translation**

Variables defined inside a behavior or channel are converted into data members of the corresponding C *struct*

### 4. **Port resolution**

Ports of behavior or channel are converted into data members of the corresponding C *struct*

### 5. **Method globalization**

Functions inside a behavior or channel are converted into global functions

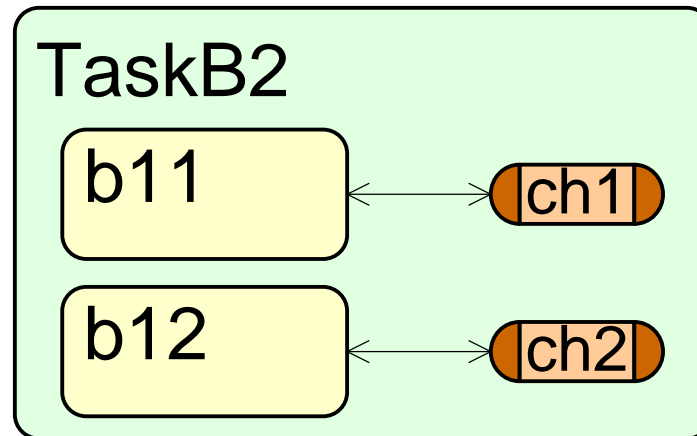
### 6. **Global context creation**

A static *struct* instantiation for each PE is added to allocate/initialize the data used by SW



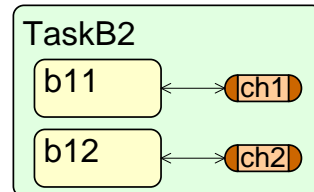
# Code Generation

- **Example**
  - Sequential executing b11, b12
    - Instances of channel class B1
  - Port bound to ch1, ch2
    - Instances of channel class CH1



# Code Generation

## 1) Module resolution



### SystemC task specification

```
1 SC_MODULE(B1){
2     int A;
3     sc_port<iChannel> myCh;
4     SC_CTOR(B1){}
5     void main(void) {
6         A = 1;
7         myCh->chCall(A*2);
8     }
9 };
10
11 SC_MODULE(TaskB2){
12     CH1 ch11, ch12; //channel inst
13     B1 b11, b12;    //module inst
14     SC_CTOR(TaskB2):
15         ch11("ch11"), ch12("ch12"),
16         b11("b11"), b12("b12") {
17         b11.myCh(ch11); // connect ch11
18         b12.myCh(ch12); // connect ch12
19     }
20     void main(void) {
21         b11.main();
22         b12.main();
23     }
24 };
```

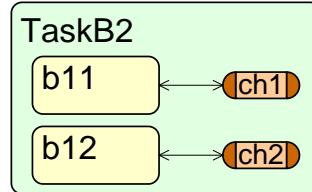
### ANSI-C task code

```
1 struct B1 {
2     struct CH1 *myCh; /* port iChannel*/
3     int A;
4 };
5 struct TaskB2 {
6     struct B1 b11, b12;
7     struct CH1 ch11, ch12;
8 };
9 void B1_main(struct B1 *This) {
10     (This->A) = 1;
11     CH1_chCall(This->myCh, (This->A)*2);
12 }
13 void TaskB2_main(struct TaskB2 *This){
14     B1_main(&(This->b11));
15     B1_main(&(This->b12));
16 }
17 struct TaskB2 taskB2= {
18     {&(taskB2.ch11),0/*A*/}/*b11*/,
19     {&(taskB2.ch12),0/*A*/}/*b12*/,
20     {} /*ch11*/, {} /*ch12*/
21 };
22 void TaskB2() {
23     TaskB2_main( &task1);
24 }
```



# Code Generation

## 2) Hierarchy translation



### SystemC task specification

```
1 SC_MODULE(B1){
2   int A;
3   sc_port<iChannel> myCh;
4   SC_CTOR(B1){}
5   void main(void) {
6     A = 1;
7     myCh->chCall(A*2);
8   }
9 };
10
11 SC_MODULE(TaskB2){
12   CH1 ch11, ch12; //channel inst
13   B1 b11, b12;    //module inst
14   SC_CTOR(TaskB2):
15     ch11("ch11"), ch12("ch12"),
16     b11("b11"), b12("b12") {
17     b11.myCh(ch11); // connect ch11
18     b12.myCh(ch12); // connect ch12
19   }
20   void main(void) {
21     b11.main();
22     b12.main();
23   }
24 };
```

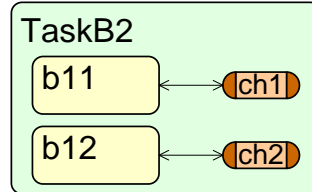
### ANSI-C task code

```
1 struct B1 {
2   struct CH1 *myCh; /* port iChannel*/
3   int A;
4 };
5 struct TaskB2 {
6   struct B1 b11, b12;
7   struct CH1 ch11, ch12;
8 };
9 void B1_main(struct B1 *This) {
10   (This->A) = 1;
11   CH1_chCall(This->myCh, (This->A)*2);
12 }
13 void TaskB2_main(struct TaskB2 *This){
14   B1_main(&(This->b11));
15   B1_main(&(This->b12));
16 }
17 struct TaskB2 taskB2= {
18   {&(taskB2.ch11),0/*A*/}/*b11*/,
19   {&(taskB2.ch12),0/*A*/}/*b12*/,
20   {} /*ch11*/, {} /*ch12*/
21 };
22 void TaskB2() {
23   TaskB2_main( &task1);
24 }
```



# Code Generation

## 3) Variable translation



### SystemC task specification

```
1 SC_MODULE(B1){
2   int A;
3   sc_port<iChannel> myCh;
4   SC_CTOR(B1){}
5   void main(void) {
6     A = 1;
7     myCh->chCall(A*2);
8   }
9 };
10
11 SC_MODULE(TaskB2){
12   CH1 ch11, ch12; //channel inst
13   B1 b11, b12;    //module inst
14   SC_CTOR(TaskB2):
15     ch11("ch11"), ch12("ch12"),
16     b11("b11"), b12("b12") {
17     b11.myCh(ch11); // connect ch11
18     b12.myCh(ch12); // connect ch12
19   }
20   void main(void) {
21     b11.main();
22     b12.main();
23   }
24 };
```

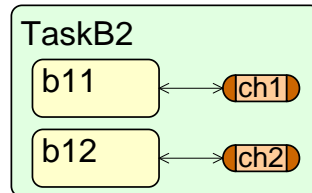
### ANSI-C task code

```
1 struct B1 {
2   struct CH1 *myCh; /* port iChannel*/
3   int A;
4 };
5 struct TaskB2 {
6   struct B1 b11, b12;
7   struct CH1 ch11, ch12;
8 };
9 void B1_main(struct B1 *This) {
10   (This->A) = 1;
11   CH1_chCall(This->myCh, (This->A)*2);
12 }
13 void TaskB2_main(struct TaskB2 *This){
14   B1_main(&(This->b11));
15   B1_main(&(This->b12));
16 }
17 struct TaskB2 taskB2= {
18   {&(taskB2.ch11),0/*A*/}/*b11*/,
19   {&(taskB2.ch12),0/*A*/}/*b12*/,
20   {} /*ch11*/, {} /*ch12*/
21 };
22 void TaskB2() {
23   TaskB2_main( &task1);
24 }
```



# Code Generation

## 4) Port resolution



### SystemC task specification

```
1 SC_MODULE(B1){
2   int A;
3   sc_port<iChannel> myCh;
4   SC_CTOR(B1){}
5   void main(void) {
6     A = 1;
7     myCh->chCall(A*2);
8   }
9 };
10
11 SC_MODULE(TaskB2){
12   CH1 ch11, ch12; //channel inst
13   B1 b11, b12;    //module inst
14   SC_CTOR(TaskB2):
15     ch11("ch11"), ch12("ch12"),
16     b11("b11"), b12("b12") {
17     b11.myCh(ch11); // connect ch11
18     b12.myCh(ch12); // connect ch12
19   }
20   void main(void) {
21     b11.main();
22     b12.main();
23   }
24 };
```

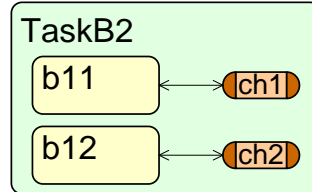
### ANSI-C task code

```
1 struct B1 {
2   struct CH1 *myCh; /* port iChannel*/
3   int A;
4 };
5 struct TaskB2 {
6   struct B1 b11, b12;
7   struct CH1 ch11, ch12;
8 };
9 void B1_main(struct B1 *This) {
10   (This->A) = 1;
11   CH1_chCall(This->myCh, (This->A)*2);
12 }
13 void TaskB2_main(struct TaskB2 *This){
14   B1_main(&(This->b11));
15   B1_main(&(This->b12));
16 }
17 struct TaskB2 taskB2= {
18   {&(taskB2.ch11),0/*A*/}/*b11*/,
19   {&(taskB2.ch12),0/*A*/}/*b12*/,
20   {} /*ch11*/, {} /*ch12*/
21 };
22 void TaskB2() {
23   TaskB2_main( &task1);
24 }
```



# Code Generation

## 5) Method globalization



### SystemC task specification

```
1 SC_MODULE(B1){
2     int A;
3     sc_port<iChannel> myCh;
4     SC_CTOR(B1){}
5     void main(void) {
6         A = 1;
7         myCh->chCall(A*2);
8     }
9 };
10
11 SC_MODULE(TaskB2){
12     CH1 ch11, ch12; //channel inst
13     B1 b11, b12;    //module inst
14     SC_CTOR(TaskB2):
15         ch11("ch11"), ch12("ch12"),
16         b11("b11"), b12("b12") {
17         b11.myCh(ch11); // connect ch11
18         b12.myCh(ch12); // connect ch12
19     }
20     void main(void) {
21         b11.main();
22         b12.main();
23     }
24 };
```

### ANSI-C task code

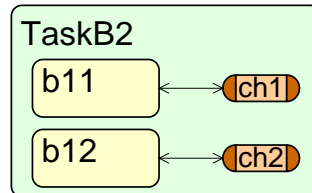
```
1 struct B1 {
2     struct CH1 *myCh; /* port iChannel*/
3     int A;
4 };
5 struct TaskB2 {
6     struct B1 b11, b12;
7     struct CH1 ch11, ch12;
8 };
9 void B1_main(struct B1 *This) {
10     (This->A) = 1;
11     CH1_chCall(This->myCh, (This->A)*2);
12 }
13 void TaskB2_main(struct TaskB2 *This){
14     B1_main(&(This->b11));
15     B1_main(&(This->b12));
16 }
17 struct TaskB2 taskB2= {
18     {&(taskB2.ch11),0/*A*/}/*b11*/,
19     {&(taskB2.ch12),0/*A*/}/*b12*/,
20     {} /*ch11*/, {} /*ch12*/
21 };
22 void TaskB2() {
23     TaskB2_main( &task1);
24 }
```





# Code Generation

## 6) Global context creation



### SystemC task specification

```
1 SC_MODULE(B1){
2   int A;
3   sc_port<iChannel> myCh;
4   SC_CTOR(B1){}
5   void main(void) {
6     A = 1;
7     myCh->chCall(A*2);
8   }
9 };
10
11 SC_MODULE(TaskB2){
12   CH1 ch11, ch12; //channel inst
13   B1 b11, b12;    //module inst
14   SC_CTOR(TaskB2):
15     ch11("ch11"), ch12("ch12"),
16     b11("b11"), b12("b12") {
17     b11.myCh(ch11); // connect ch11
18     b12.myCh(ch12); // connect ch12
19   }
20   void main(void) {
21     b11.main();
22     b12.main();
23   }
24 };
```

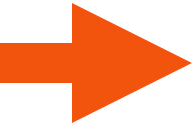
### ANSI-C task code

```
1 struct B1 {
2   struct CH1 *myCh; /* port iChannel*/
3   int A;
4 };
5 struct TaskB2 {
6   struct B1 b11, b12;
7   struct CH1 ch11, ch12;
8 };
9 void B1_main(struct B1 *This) {
10   (This->A) = 1;
11   CH1_chCall(This->myCh, (This->A)*2);
12 }
13 void TaskB2_main(struct TaskB2 *This){
14   B1_main(&(This->b11));
15   B1_main(&(This->b12));
16 }
17 struct TaskB2 taskB2= {
18   {&(taskB2.ch11),0/*A*/}/*b11*/,
19   {&(taskB2.ch12),0/*A*/}/*b12*/,
20   {} /*ch11*/, {} /*ch12*/
21 };
22 void TaskB2() {
23   TaskB2_main( &task1);
24 }
```



# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**

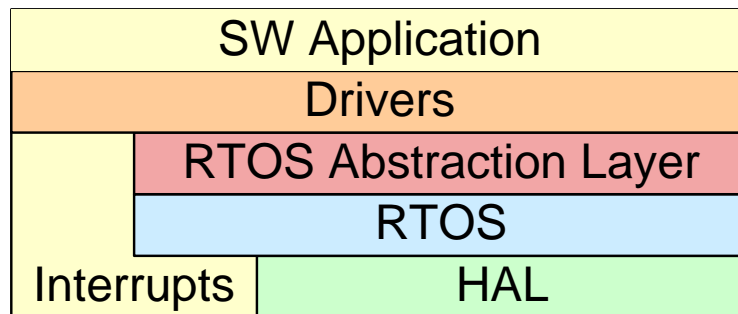


# Multi-Task Synthesis

- **Variants**

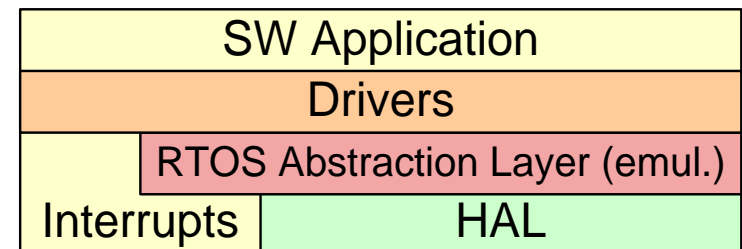
- **RTOS-based**

- General solution
    - Off-the-shelf RTOS
    - Flexible, well tested



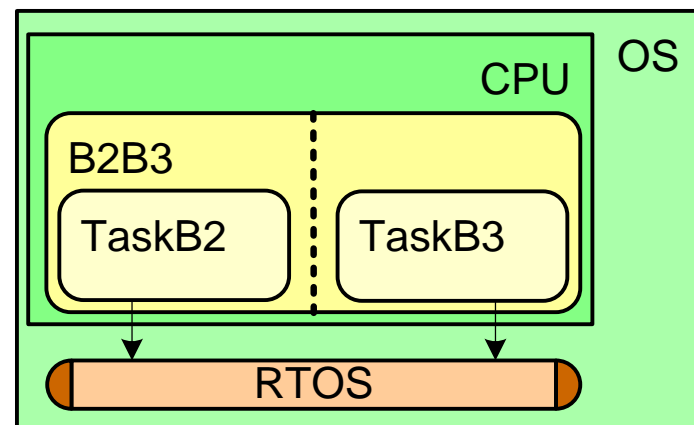
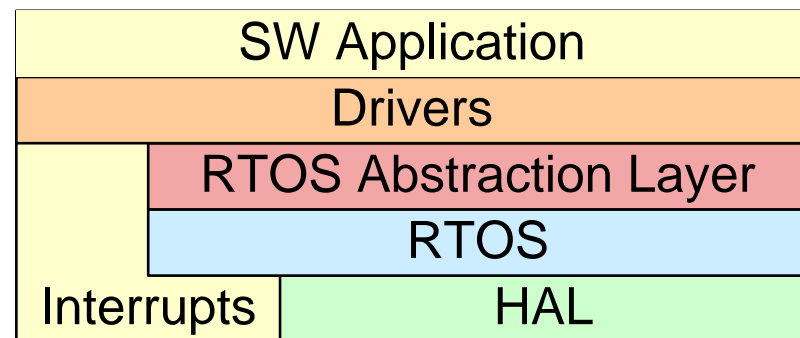
- **Interrupt-based**

- Few reactive tasks
    - Resource constraints inhibit RTOS
    - Unavailability of RTOS
    - E.g. DSP with encoder / decoder application



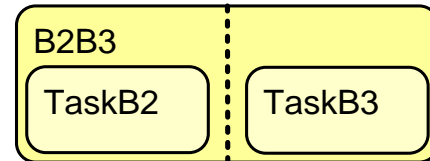
# RTOS-based Multi-Tasking

- **Based on off the shelf RTOS**
  - e.g.  $\mu$ C/OS-II, eCos, vxWorks
- **Uses RTOS Abstraction Layer (RAL)**
  - Canonical interface
    - Limits interdependency RTOS / Synthesis
- **Multi-task synthesis**
  - Generate task management code
  - Adjust internal communication
- **RTOS-based multi-tasking example**
  - Parent *B2B3*
  - Two parallel child tasks:
    - *TaskB2*
    - *TaskB3*



# Multi-Task Synthesis

- RTOS-based multi-tasking example



```
1 SC_MODULE(B2B3) {
2 public:
3   sc_port<iRTOS> rtos;
4   TaskB2 taskB2;
5   TaskB3 taskB3;
6   SC_CTOR(B2B3):
7     taskB2("taskB2", 5, 4096),
8     taskB3("taskB3", 2, 4096) {
9     taskB2.rtos(rtos);
10    taskB3.rtos(rtos);
11  }
12  void main(void) {
13    taskB2.release();
14    taskB3.release();
15    taskB2.join();
16    taskB3.join();
17  }
18 };
```

SystemC

```
1 struct B2B3{
2   struct TaskB2 task_b2;
3   struct TaskB3 task_b3;};
4 void *TaskB2_main(void *arg){
5   struct TaskB2 *this=(struct TaskB2*)arg;
6   /* ...*/
7 }
8 void *TaskB3_main(void *arg){
9   struct TaskB3 *this=(struct TaskB3*)arg;
10  /* ... */
11 }
12 void *B2B3_main(void *arg){
13   struct B2B3 *this= (struct B2B3*)arg;
14   os_task_handle task_b2, task_b3;
15   task_b2 = taskCreate(TaskB2_main,
16                       &this->taskB2, 5, 4096);
17   task_b3 = taskCreate(TaskB3_main,
18                       &this->taskB3, 2, 4096);
19
20   taskJoin(task_b2);
21   taskJoin(task_b3);
22 }
```

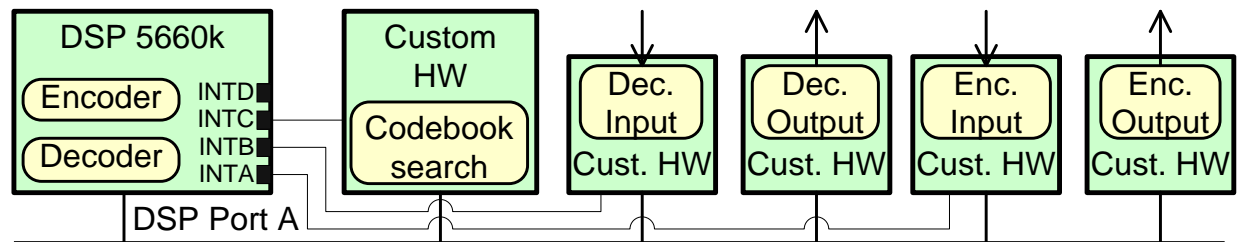
ANSI-C



# Interrupt-based multi-tasking

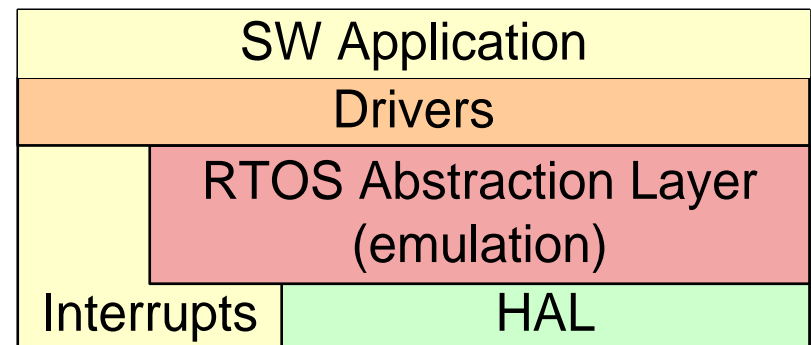
- **Alternative multi-tasking**

- Few tasks
- Resource constraints
  - Memory, footprint, performance
- Unavailability of RTOS port



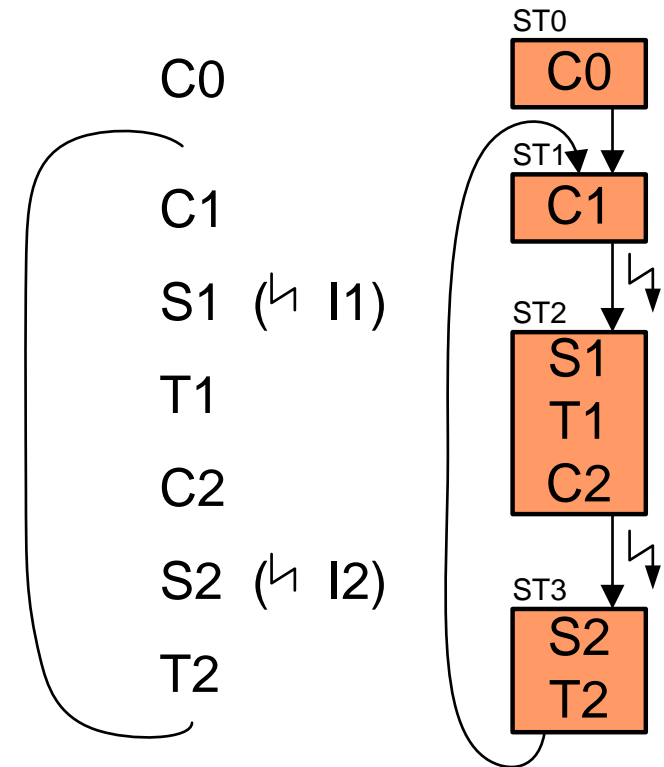
- **Overview**

- Use interrupts for threads of execution
- Emulate minimal RTOS services
  - E.g. suspend, synchronize



# Interrupt-based multi-tasking

- **Procedure overview**
  - Convert tasks to state machine
  - Execute state machine in interrupt handler
  - Execute lowest priority task as main()
- **Assume only interrupt synchronization**
- **Code composed of**
  - Computation  $C_n$
  - Synchronization  $S_n$ ,
  - Data transfer  $T_n$
  - Interrupt sych.  $I_n$
- **Convert task code to state machine**
  - New state for each synchronization
    - E.g. ST2, ST3
  - New state for each conditional
    - E.g. Loop: ST0, ST1
- **Execute state machine in interrupt**



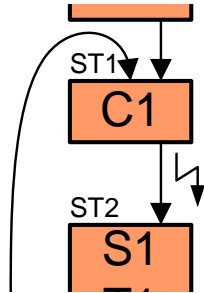
# Multi-Task Synthesis

- **Interrupt-based multi-tasking example**

- **Example execution**

- Assume computation C1 [line 10] just finished
- New state ST2
- Attempt S1
  - » Unavailable
  - » ISR terminates
- Receive I1
- Release S1
- executeTask0()
- Continue ST2
- Attempt S1
  - » Available
- Continue

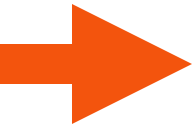
```
1  /* interrupt handler */
2  void intHandler_I1() {
3      release(S1);    /* set S1 ready */
4      executeTask0(); /* task state machine */
5  }
6  /* task state machine */
7  void executeTask0() {
8      do { switch(Task0.State) {
9          /* ... */
10         case ST1: C1(...);
11                 Task0.State = ST2;
12         case ST2: if(attempt(S1))
13                   T1_receive(...);
14                   else break;
15                   C2(...);
16                   Task0.State = ST3;
17         case ST3: /* ... */
18         } } while (Task0.State == ST1);
19 }
```





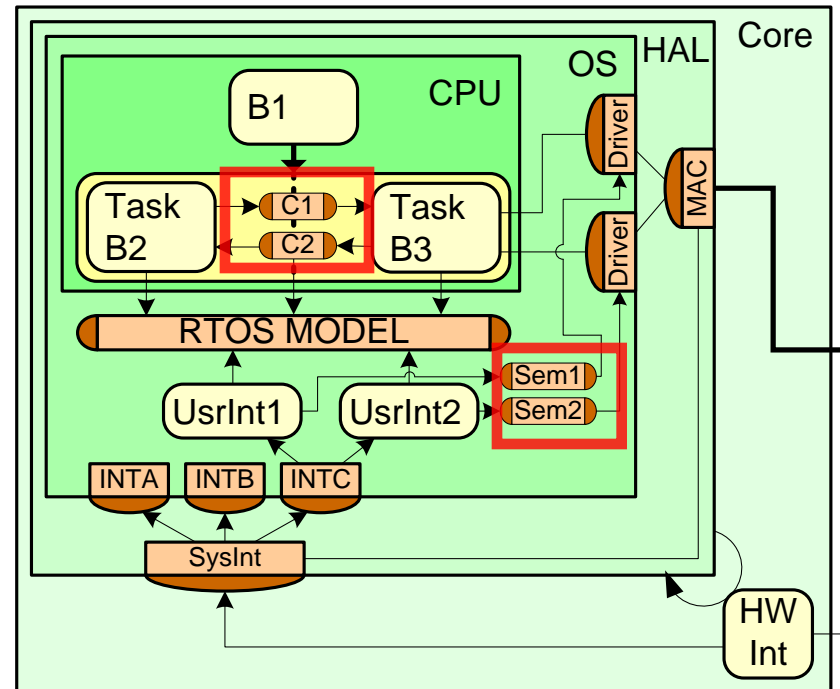
# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



# Internal Communication

- **Communication within processor**
  - Replace with target specific implementation
    - e.g. RTOS semaphore, event, msg. queue



# Internal Communcation

- **Implementation example**

- **Single handshake**

- One way synchronous without data

- **Xilkernel (Xilinx proprietary RTOS) implementation**

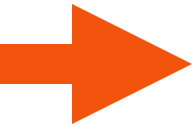
- Semaphore-based

```
1  /** SHS OS-specific struct */
2  typedef struct {
3      sem_t req;  /** os semaphore */
4  } tCh_shs;
5  void Shs__init(tCh_shs *pThis){
6      int retVal = sem_init(&pThis->req, 0, 0);
7      /* ... error handling */
8  }
9  void Shs_send(tCh_shs *pThis){
10     int retVal = sem_post(&pThis->req);
11     /* ... error handling */
12 }
13 void Shs_receive(tCh_shs *pThis){
14     int retVal = sem_wait(&pThis->req);
15     /* ... error handling */
16 }
```



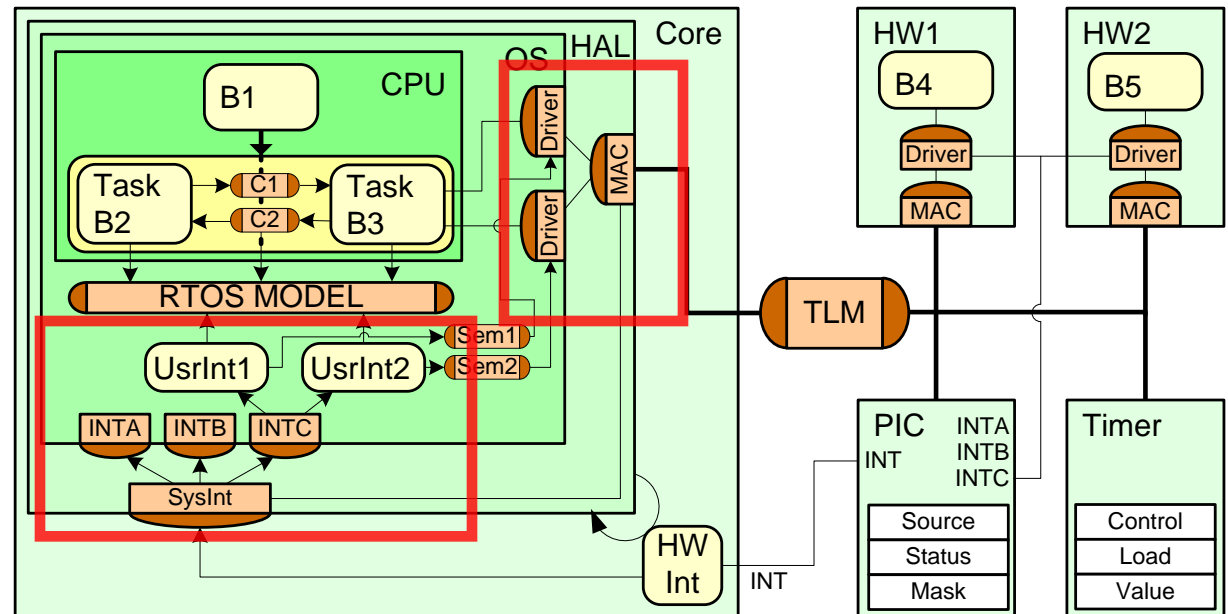
# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



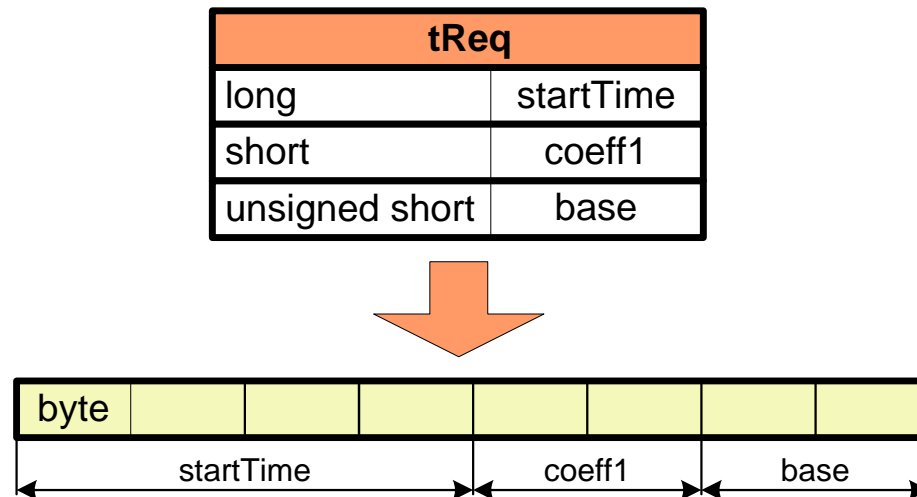
# External Communication

- **Communication outside of processor**
  - E.g. with HW Accelerator
- **ISO / OSI layered to support heterogeneous architectures**
  - Data formatting
  - Packetization
  - Synchronization
  - MAC



# External Communication

- **Data formatting (marshalling)**
  - Problem: different memory representation for same data
    - Endianness (byte order)
    - Packing rules
    - Bit widths for data types
  - Convert typed data into flat untyped byte stream
    - Interpretable by everybody



# External Communication

- **Data formatting example:**
  - Marshalling code uses
    - `hton<datatype>`
    - `ntoh<datatype>`

## User type definition

```
1 typedef struct stReq {
2     long         startTime;
3     short        coeff1;
4     unsigned short base;
5 } tReq;
```

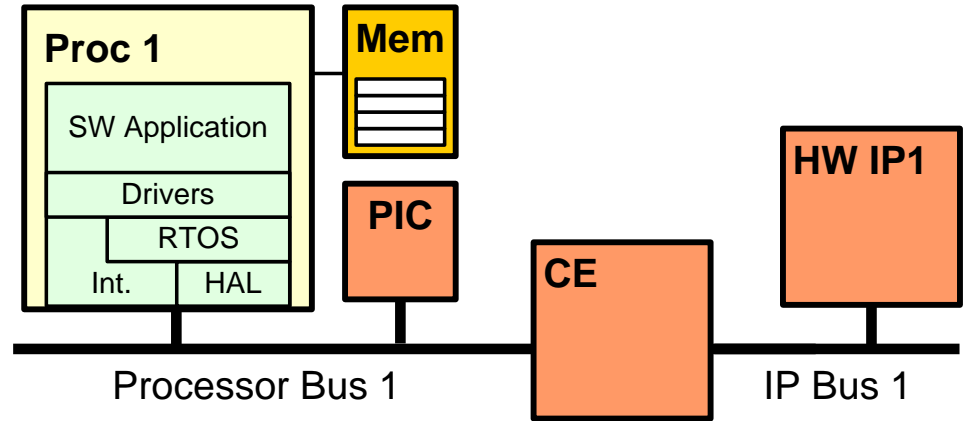
## Marshalling code

```
1 void myCh_send(/* ...*/ *This,
2               struct tReq *pD){
3     unsigned char *pB = This->buf;
4     htonl(pB, pD->startTime);
5     pB += 4;
6     htons(pB, pD->coeff1);
7     pB += 2;
8     htons(pB, pD->base);
9     pB += 2;
10    DLink0_trans_send(/*...*/This->buf, 8);
11 }
```



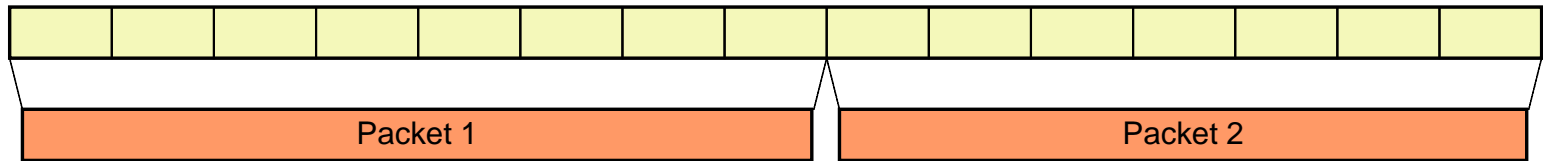
# External Communication

- Packetization



- Break user defined length stream into packets
- Limit storage requirements during transport (CEs)

Byte stream





# External Communication

- **Packetization example code**

- Transmits input *pMsg*, of length *len* in packets of up to *CONFIG\_PACKET\_SIZE*

```
1 DLink0_trans_send(void *pMsg, unsigned int len){
2   unsigned char *pPos = pMsg;
3   while(len) {
4     unsigned long pktLen;
5     /* length is minimum of max size and len */
6     pktLen = min(len, CONFIG_PACKET_SIZE);
7
8     DLink0_net_send(pPos, pktLen); /* transfer */
9
10    len -= pktLen; /* decr. transferred len */
11    pPos += pktLen; /* advance pointer */
12  }
13 }
```

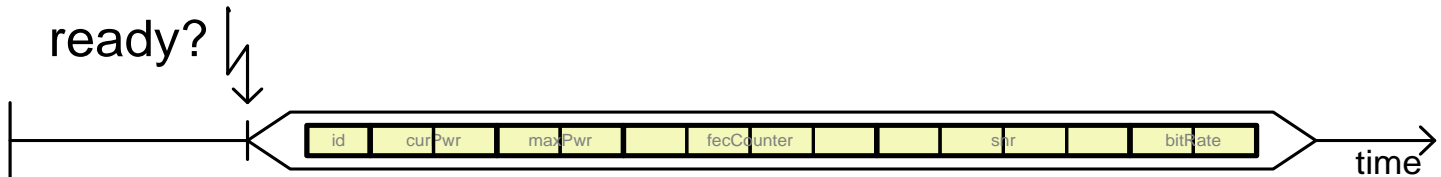


# External Communication

- **Synchronization**

- Ensure slave is ready before master initiates transaction

- Ready to receive data / Data ready



- Master / slave bus: Separate event
- Node-based bus: Synchronization packet

- Processor synchronization options:

- Interrupt
  - » Separate interrupt line connected to interrupt controller
  - » Low latency
  - » Interrupt overhead
  - » Shared interrupts
- Polling
  - » Master periodically checks slave using data connection
  - » Polling period
- Hybrid solutions possible



# External Communication

- **Synchronization by interrupt**

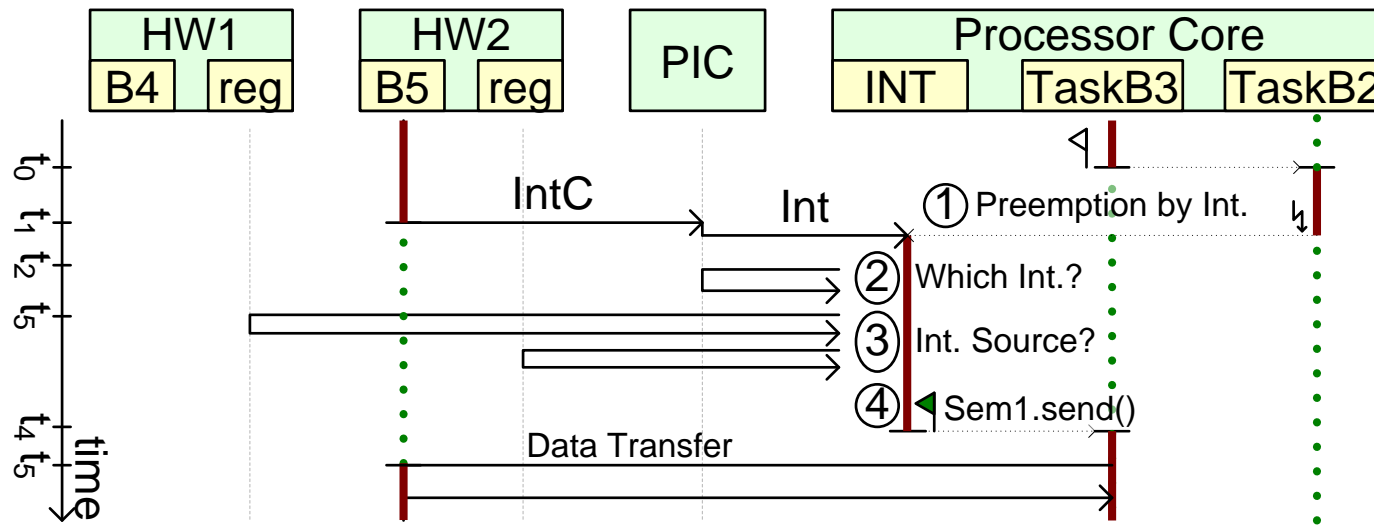
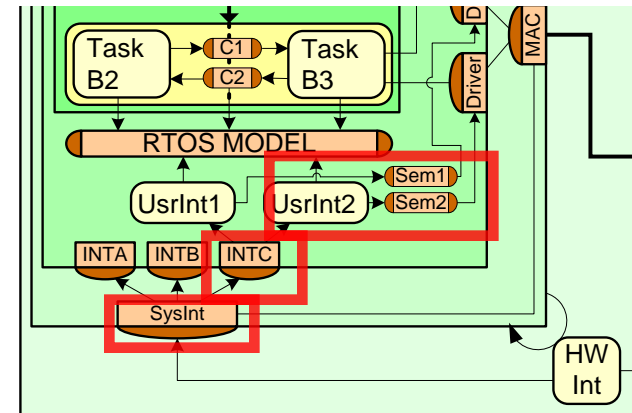
- 1) Low level interrupt handler
  - Preempts current task

- 2) System interrupt handler
  - Checks PIC

- 3) User-specific interrupt handler
  - Handles shared interrupts

- 4) Semaphore

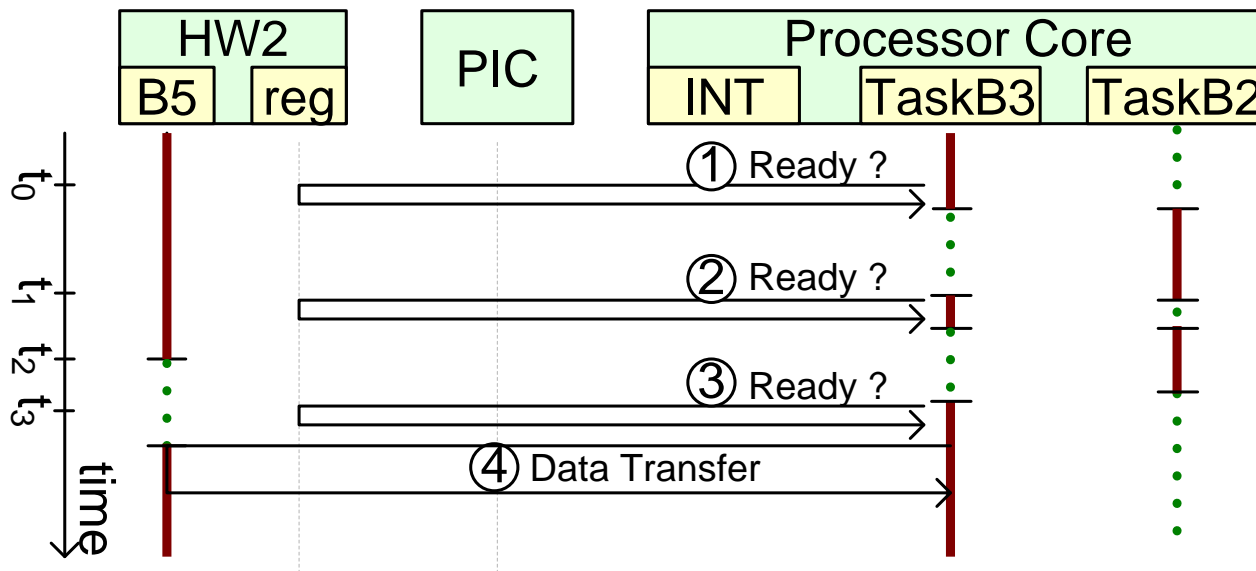
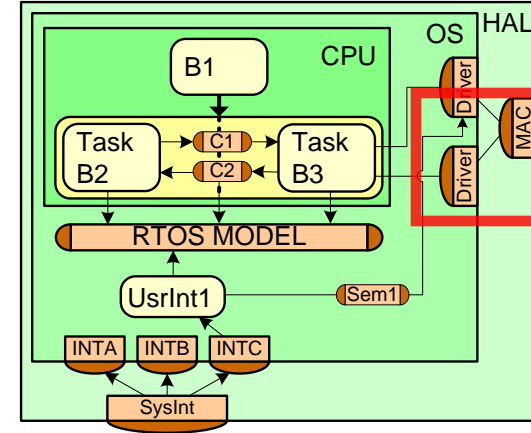
- Releases task



# External Synchronization

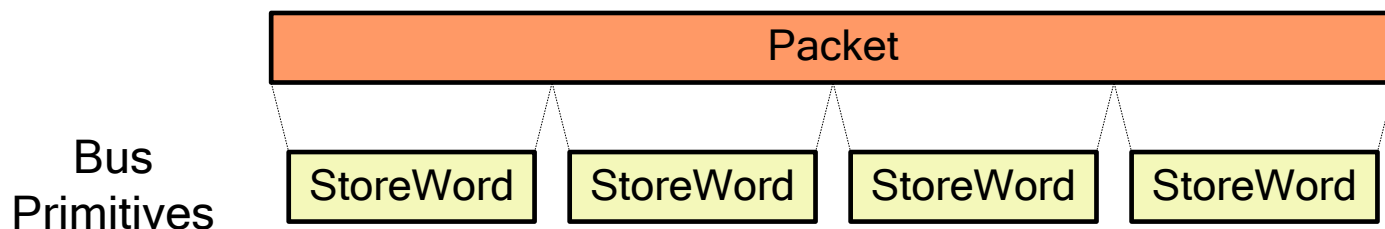
- **Polling example**

- 1) Expect message, check ready?
  - Suspend for polling period
- 2) Check ready?
  - Suspend for polling period
- 3) Check ready?
  - Finish synchronization
- 4) Data transfer



# External Communication

- **Media Access Control (MAC)**
  - Provides access to bus medium
    - Low level driver
  - Break packet into bus transactions



- Simple drivers use processor memory interface
  - Memory mapped bus interface
- More complex communicate / synchronize with protocol controller
  - I2C, CAN, FlexRay



# External Communication

- **Media Access Control (MAC)**
  - Example for memory mapped bus access
    - Left: Cast integer *addr* to pointer, take value of
    - Right: Cast void *\*pD*, to int pointer, get value
    - Assign value to “value of” results in bus transaction
    - Repeat for word, short, byte

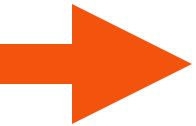
```
1 void masterWrite(unsigned int addr, void *pData, unsigned int len) {
2   unsigned char *pD = (unsigned char*)pData;
3   while (len >= 4 ) {
4     *((unsigned int*)addr) = *((unsigned int*)pD);
5     len -= 4; pD += 4;
6   }
7   if (len >= 2 ) { /* remaning short */
8     *((unsigned short*)addr) = *((unsigned short*)pD);
9     len -= 2; pD += 2;
10  }
11  if (len >= 1) { /* the last byte */
12    *((unsigned char*)addr) = *((unsigned char*)pD);
13    len -= 1; pD += 1;
14  }
15 }
```

Note: code assumes 32bit processor.



# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



# Startup Code

- **Initialize system and release tasks**

- Global data structure
- Board Support Package (BSP)
- Operating System
- Create synchronization channels
- Register interrupts
- Create and release user tasks

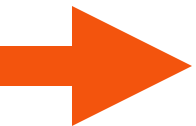
```
1 void main(void) {
2     PE_Struct_Init(&PE0);
3     BSP_init();
4     OSInit();
5
6     c_os_handshake_init(&PE0->sem1);
7     c_os_handshake_init(&PE0->sem2);
8     BSP_UserIrqRegister(INT1, Int1Handler, /*...*/);
9     BSP_UserIrqRegister(INT2, Int2Handler, /*...*/);
10
11     taskCreate(task_b2b3, NULL,
12                B2B3_main, &this->task_b2b3);
13
14     OSStart();
15 }
```





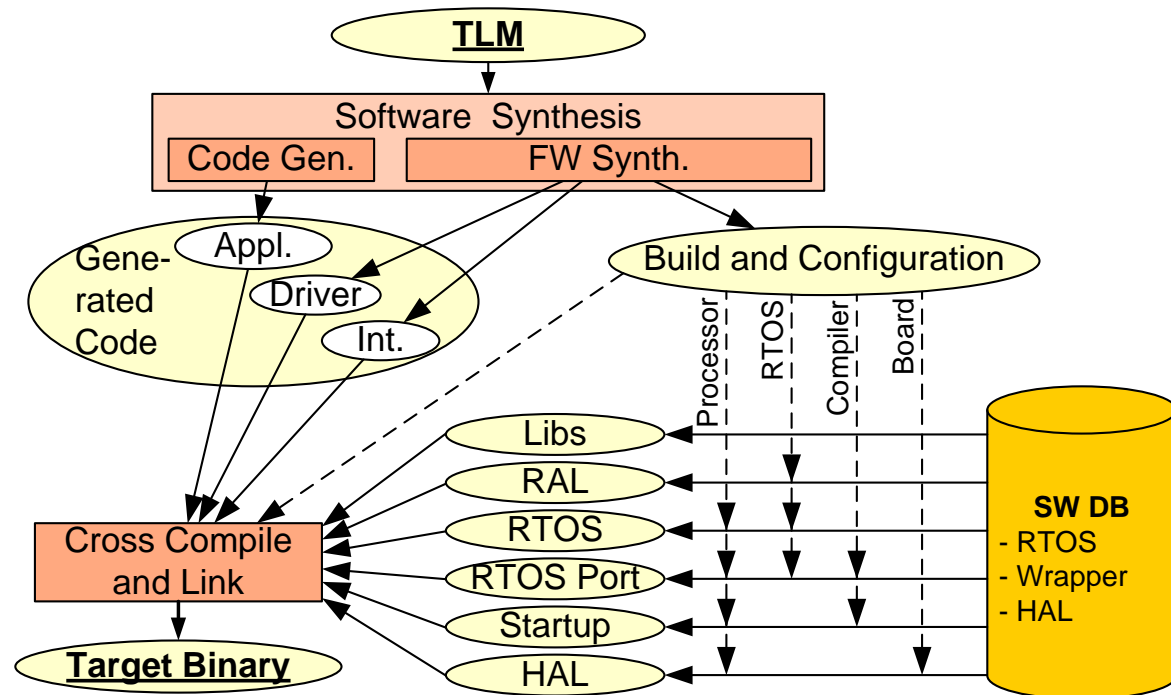
# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



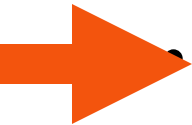
# Binary Image Generation

- **Generate binary for each processor**
  - Generate build and configuration files
    - Select software database components
    - Configure RTOS
  - Cross compile and link
  - Significant effort in DB design
    - Minimize components
    - Analyze dependencies
    - Goal: flexible composition



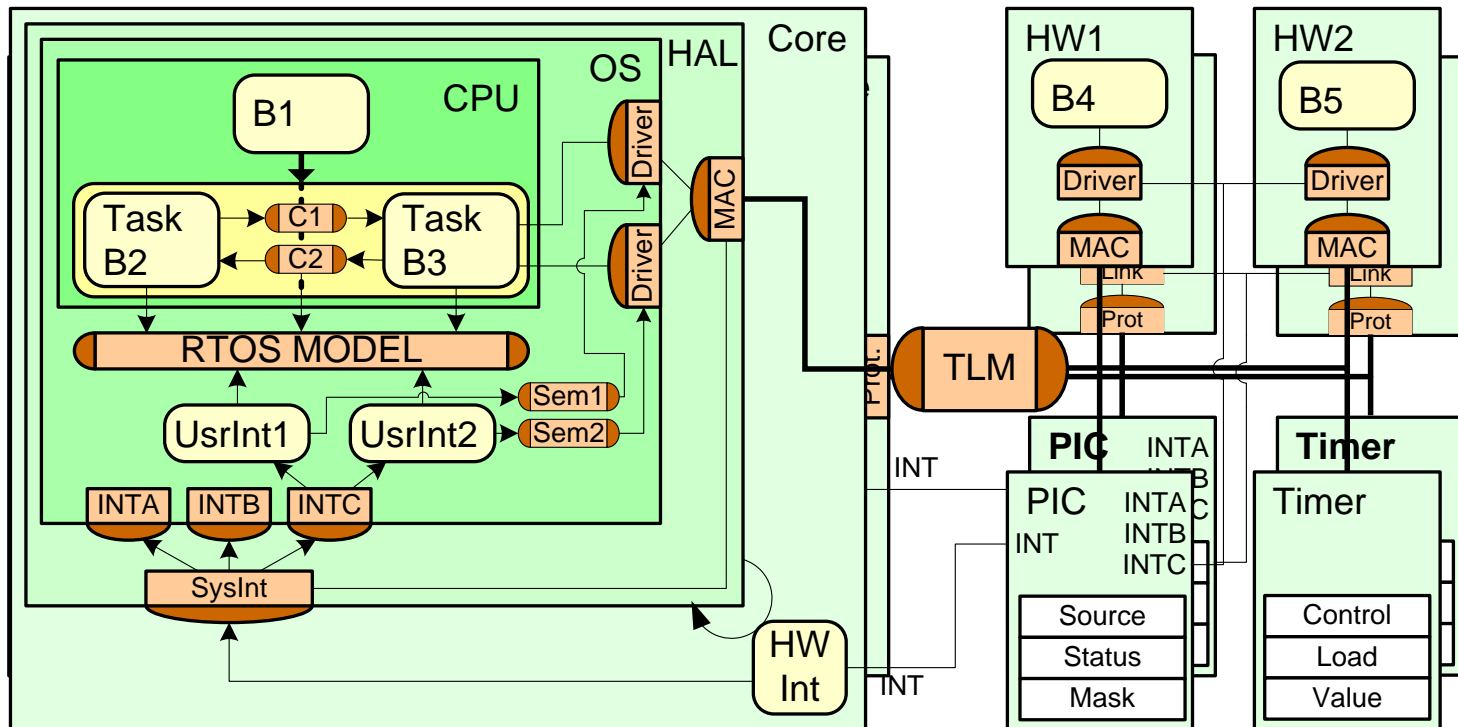
# Outline

- **Introduction**
  - Preliminaries
  - Software Synthesis Overview
- **Code Generation**
- **Hardware-dependent Software**
  - Multi-Task Synthesis
  - Internal Communication
  - External Communication
  - Startup Code Generation
  - Binary Image Generation
- **Execution**
- **Summary**



# Execution

- **Validate binary image**
  - Target processor on prototype
  - ISS-based virtual platform



# Summary

- **Embedded software generation from system model**
  - Including
    - Communication synthesis (external / internal)
    - Multi-task synthesis
    - Binary image creation
- **Integrated into ESL flow**
  - Seamless solution
- **Complete: from abstract model to implementation!**
  - Completes ESL flow for software
  - Eliminates tedious and error prone manual HdS development
  - Significant productivity gain
  - Enables rapid design space exploration

