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
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
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**

- Concurrent jobs at the same time

- Resource constrained
  - Footprint and memory consumption
  - Computing power
  - Energy consumption

![Diagram of Embedded System]

- Controller
- Actuator
- Sensor
- Physical System
- Embedded System
- Energy consumption
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
    – Accellerator 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

![Diagram showing TaskB2 with b11, ch1, b12, and ch2]
1) Module resolution

**SystemC task specification**

```systemc
SC_MODULE(B1) {
    int A;
    sc_port<iChannel> myCh;
    SC_CTOR(B1) {} 
    void main(void) {
        A = 1;
        myCh->chCall(A*2);
    }
}

SC_MODULE(TaskB2) {
    CH1 ch11, ch12; //channel inst
    B1 b11, b12; //module inst
    SC_CTOR(TaskB2) :
        ch11("ch11"), ch12("ch12"),
        b11("b11"), b12("b12") {
            b11.myCh(ch11); // connect ch11
            b12.myCh(ch12); // connect ch12
        }
    void main(void) {
        b11.main();
        b12.main();
    }
}
```

**ANSI-C task code**

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int A;
};

struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};

void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}

void TaskB2_main(struct TaskB2 *This){
    B1_main(&This->b11);
    B1_main(&This->b12);
}

struct TaskB2 taskB2= {
    &taskB2.ch11,0/*A*/}/*b11*/,
    &taskB2.ch12,0/*A*/}/*b12*/,
    {} /*ch11*/,
    {} /*ch12*/
};

void TaskB2() {
    TaskB2_main( &task1);
}
```
2) Hierarchy translation

SystemC task specification

```c
SC_MODULE(B1){
    int A;
    sc_port<iChannel> myCh;
    SCCTOR(B1) {}  
    void main(void) {
        A = 1;
        myCh->chCall(A*2);
    }
};

SC_MODULE(TaskB2){
    CH1 ch11, ch12; //channel inst
    B1 b11, b12;    //module inst
    SCCTOR(TaskB2):
        ch11("ch11"), ch12("ch12"),
        b11("b11"), b12("b12") {
            b11.myCh(ch11); // connect ch11
            b12.myCh(ch12); // connect ch12
        }
    void main(void) {
        b11.main();
        b12.main();
    }
};
```

ANSI-C task code

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int A;
};

struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};

void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}

void TaskB2_main(struct TaskB2 *This) {
    B1_main(&This->b11);
    B1_main(&This->b12);
}

struct TaskB2 taskB2= {
    &taskB2.ch11),0/*A*/}/"b11*/,
    &taskB2.ch12),0/*A*/}/"b12*/,
}; /*ch11*/, { /*ch12*/
}

void TaskB2() {
    TaskB2_main( &task1);
}
```
3) Variable translation

### SystemC task specification

```c
SC_MODULE(B1) {
    int A;
    sc_port<iChannel> myCh;
    SC_CTOR(B1) {} 
    void main(void) {
        A = 1;
        myCh->chCall(A*2);
    }
}
```

```c
SC_MODULE(TaskB2) {
    CH1 ch11, ch12; //channel inst
    B1 b11, b12; //module inst
    SC_CTOR(TaskB2): ch11("ch11"), ch12("ch12"), b11("b11"), b12("b12") {
        b11.myCh(ch11); // connect ch11
        b12.myCh(ch12); // connect ch12
    }
    void main(void) {
        b11.main();
        b12.main();
    }
}
```

### ANSI-C task code

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int A;
};
```

```c
struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};
```

```c
void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}
```

```c
void TaskB2_main(struct TaskB2 *This) {
    B1_main(&taskB2.ch11);
    B1_main(&taskB2.ch12);
}
```

```c
void TaskB2() {
    TaskB2_main( &task1);
}
```
4) Port resolution

SystemC task specification

```c
SC_MODULE(B1){
    int A;
    sc_port<iChannel> myCh;
    SCCTOR(B1){}
    void main(void) {
        A = 1;
        myCh->chCall(A*2);
    }
}

SC_MODULE(TaskB2){
    CH1 ch11, ch12; //channel inst
    B1 b11, b12; //module inst
    SCCTOR(TaskB2):
        ch11("ch11"), ch12("ch12"),
        b11("b11"), b12("b12") {
            b11.myCh(ch11); // connect ch11
            b12.myCh(ch12); // connect ch12
        }
    void main(void) {
        b11.main();
        b12.main();
    }
}
```

ANSI-C task code

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int A;
};

struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};

void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}

void TaskB2_main(struct TaskB2 *This){
    b11.main();
    b12.main();
}
```

SystemC task specification

```c
TaskB2
b11

b12
ch1
ch2
```
5) Method globalization

SystemC task specification

```c
SC_MODULE(B1){
    int A;
    sc_port<iChannel> myCh;
    SCCTOR(B1){
        void main(void) {
            A = 1;
            myCh->chCall(A*2);
        }
    }
    SC_MODULE(TaskB2){
        CH1 ch11, ch12; //channel inst
        B1 b11, b12;    //module inst
        SCCTOR(TaskB2):
            ch11("ch11"), ch12("ch12"),
            b11("b11"), b12("b12") {
        b11.myCh(ch11); // connect ch11
        b12.myCh(ch12); // connect ch12
        }
        void main(void) {
            b11.main();
            b12.main();
        }
    }
}
```

ANSI-C task code

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int A;
};
struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};
void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}
void TaskB2_main(struct TaskB2 *This) {
    B1_main((&(This->b11));
    B1_main((&(This->b12));
}
struct TaskB2 taskB2= {
    &(taskB2.ch11),0/*A*//*b11*/,
    &(taskB2.ch12),0/*A*//*b12*/,
    } /*ch11*/, { } /*ch12*/
};
void TaskB2() {
    TaskB2_main( &task1);
}
```
6) Global context creation

SystemC task specification

```c
SC_MODULE(B1){
    int A;
    sc_port<iChannel> myCh;
    SCCTOR(B1){}
    void main(void) {
        A = 1;
        myCh->chCall(A*2);
    }
}

SC_MODULE(TaskB2){
    CH1 ch11, ch12; //channel inst
    B1 b11, b12; //module inst
    SCCTOR(TaskB2):
        ch11("ch11"), ch12("ch12"),
        b11("b11"), b12("b12") {
            b11.myCh(ch11); // connect ch11
            b12.myCh(ch12); // connect ch12
        }
    void main(void) {
        b11.main();
        b12.main();
    }
}
```

ANSI-C task code

```c
struct B1 {
    struct CH1 *myCh; /* port iChannel*/
    int A;
};

struct TaskB2 {
    struct B1 b11, b12;
    struct CH1 ch11, ch12;
};

void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}

void TaskB2_main(struct TaskB2 *This){
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}

void B1_main(struct B1 *This) {
    (This->A) = 1;
    CH1_chCall(This->myCh, (This->A)*2);
}

void TaskB2_main(struct TaskB2 *This){
    B1_main(&(This->b11));
    B1_main(&((This->b12));
}

struct TaskB2 taskB2 = {
    &((taskB2.ch11),0/*A*/)/*b11*/,
    &((taskB2.ch12),0/*A*/)/*b12*/,
    } /*ch11*/, {} /*ch12*/
};

void TaskB2() {
    TaskB2_main(&task1);
}
```
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. µC/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

```systemc
SC_MODULE(B2B3) {
   public:
      sc_port<iRTOS> rtos;
      TaskB2 taskB2;
      TaskB3 taskB3;
   SC_CTOR(B2B3):
      taskB2("taskB2", 5, 4096),
      taskB3("taskB3", 2, 4096) {
         taskB2.rtos(rtos);
         taskB3.rtos(rtos);
      }
   void main(void) {
      taskB2.release();
      taskB3.release();
      taskB2.join();
      taskB3.join();
   }
};
```

```c
struct B2B3{
   struct TaskB2 task_b2;
   struct TaskB3 task_b3;
};
void *TaskB2_main(void *arg){
   struct TaskB2 *this=(struct TaskB2*)arg;
   /* ... */
}
void *TaskB3_main(void *arg){
   struct TaskB3 *this=(struct TaskB3*)arg;
   /* ... */
}
void *B2B3_main(void *arg){
   struct B2B3 *this= (struct B2B3*)arg;
   os_task_handle task_b2, task_b3;
   task_b2 = taskCreate(TaskB2_main,
      &this->task_b2, 5, 4096);
   task_b3 = taskCreate(TaskB3_main,
      &this->task_b3, 2, 4096);
   taskJoin(task_b2);
   taskJoin(task_b3);
}
```
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

```c
/* interrupt handler */
void intHandler_I1() {
    release(S1);    /* set S1 ready */
    executeTask0(); /* task state machine */
}

/* task state machine */
void executeTask0() {
    do { switch(Task0.State) {
        /* ... */
        case ST1: C1(...);
            Task0.State = ST2;
        case ST2: if(attempt(S1))
            T1_receive(...);
                else break;
        C2(...);
            Task0.State = ST3;
        case ST3: /* ... */
    } } while (Task0.State == ST1);
```
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 Communication

- Implementation example
  - Single handshake
    - One way synchronous without data
  - Xilkernel (Xilinx proprietary RTOS) implementation
    - Semaphore-based

```c
/** SHS OS-specific struct */
t typedef struct {
    sem_t req; /* os semaphore */
} tCh_shs;

void Shs__init(tCh_shs *pThis){
    int retVal = sem_init(&pThis->req, 0, 0);
    /* ... error handling */
}

void Shs_send(tCh_shs *pThis){
    int retVal = sem_post(&pThis->req);
    /* ... error handling */
}

void Shs_receive(tCh_shs *pThis){
    int retVal = sem_wait(&pThis->req);
    /* ... error handling */
}
```
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
    - Endianess (byte order)
    - Packing rules
    - Bit widths for data types
  - Convert typed data into flat untyped byte stream
    - Interpretable by everybody

<table>
<thead>
<tr>
<th>tReq</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>startTime</td>
</tr>
<tr>
<td>short</td>
<td>coeff1</td>
</tr>
<tr>
<td>unsigned short</td>
<td>base</td>
</tr>
</tbody>
</table>

```
tReq
long    startTime
short   coeff1
unsigned short base
```
External Communication

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

```c
typedef struct stReq {
    long          startTime;
    short         coeff1;
    unsigned short base;
} tReq;

void myCh_send(/* ...*/ *This,
              struct tReq *pD){
    unsigned char *pB = This->buf;
    htonlong(pB,   pD->startTime);
    pB += 4;
    htonshort(pB,  pD->coeff1);
    pB += 2;
    htonushort(pB, pD->base);
    pB += 2;
    DLink0_trans_send(/*...*/This->buf, 8);
}
```
External Communication

• Packetization

• Break user defined length stream into packets
• Limit storage requirements during transport (CEs)
External Communication

- Packetization example code
  - Transmits input `pMsg`, of length `len` in packets of up to \texttt{CONFIG_PACKET_SIZE}

```c
DLink0_trans_send(void *pMsg, unsigned int len) {
    unsigned char *pPos = pMsg;
    while(len) {
        unsigned long pktLen;
        /* length is minimum of max size and len */
        pktLen = min(len, CONFIG_PACKET_SIZE);
        DLink0_net_send(pPos, pktLen); /* transfer */
        len -= pktLen; /* decr. transferred len */
        pPos += pktLen; /* advance pointer */
    }
}
```
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

![Diagram](https://example.com/diagram.png)
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

```c
void masterWrite(unsigned int addr, void *pData, unsigned int len) {
    unsigned char *pD = (unsigned char*)pData;
    while (len >= 4) {
        *((unsigned int*)addr) = *((unsigned int*)pD);
        len -= 4; pD += 4;
    }
    if (len >= 2) {/* remaining short */
        *((unsigned short*)addr) = *((unsigned short*)pD);
        len -= 2; pD += 2;
    }
    if (len >= 1) {/* the last byte */
        *((unsigned char*)addr) = *((unsigned char*)pD);
        len -= 1; pD += 1;
    }
}
```

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

```c
void main(void) {
    PE_Struct_Init(&PE0);
    BSP_init();
    OSInit();

    c_os_handshake_init(&PE0->sem1);
    c_os_handshake_init(&PE0->sem2);
    BSP_UserIrqRegister(INT1, Int1Handler, /*..*/);
    BSP_UserIrqRegister(INT2, Int2Handler, /*..*/);

    taskCreate(task_b2b3, NULL,
               B2B3_main, &this->task_b2b3);

    OSStart();
}
```
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