

Migration from Single Core to Multi-core

ESE-Kongress 2013

1 Multi-core Concept

2 Prelinking

3 Analyzing

4 Multi-Core OS

Multi-core architecture

Objective

- Seamless migration from Single-Core to Multi-Core
- Minimal source code changes
- One Executable (ELF) covering all CPUs
- Linker should map of code & data to CPUs

Example

- Exporting/Hiding of global variables to/from other CPUs
 - ⇒ Clear definition of interfaces
 - within a CPU domain
 - across CPU domains

Multi-core the challenge for system architects



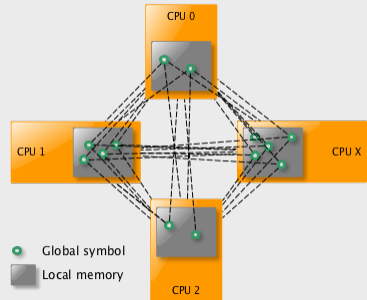
Common Linking

- One application split across multiple cores with interaction
 - ⇒ How to share data and make access efficient and safe
- Physical memory with a core local address and a global address
- Multi-core with heterogeneous core

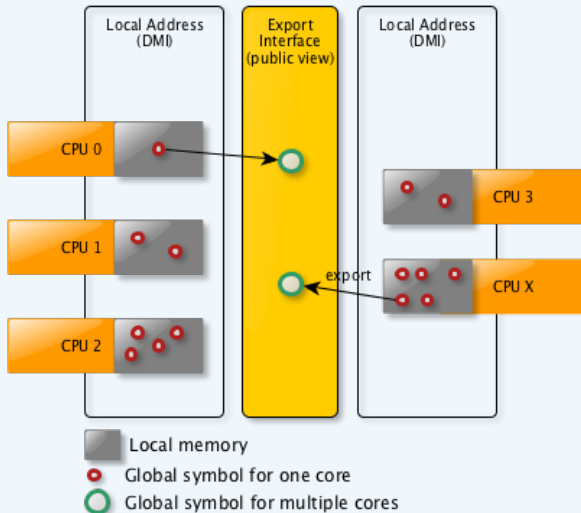
Conclusion

Multi-core application **without** clear interfaces is **not** maintainable.

Sharing of global data and code



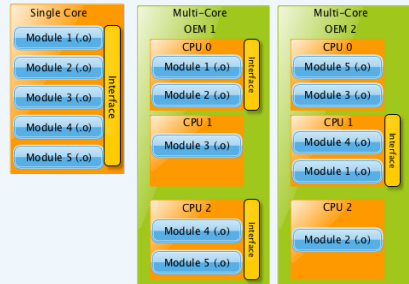
Sharing symbols **with** clear interfaces



Migration and Software-Sharing

Software-Sharing

- Use existing Single-Core object files
- Core assignment on linker level
- Define OEM dependent **interfaces**
 - ⇒ Information hiding
- Locate and create one elf-file



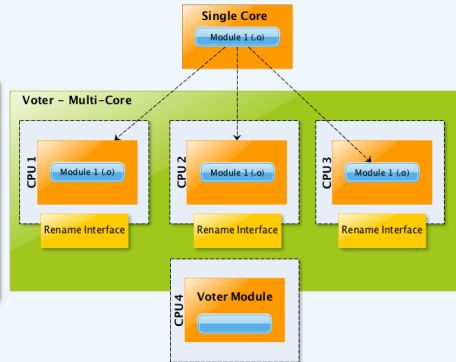
Conclusion

Existing software can be re-structured with minimal efforts; Flexible re-grouping (task-based with interfaces); Reduced test costs; Improved maintainability

Safety - Diversity and Voting

Voting

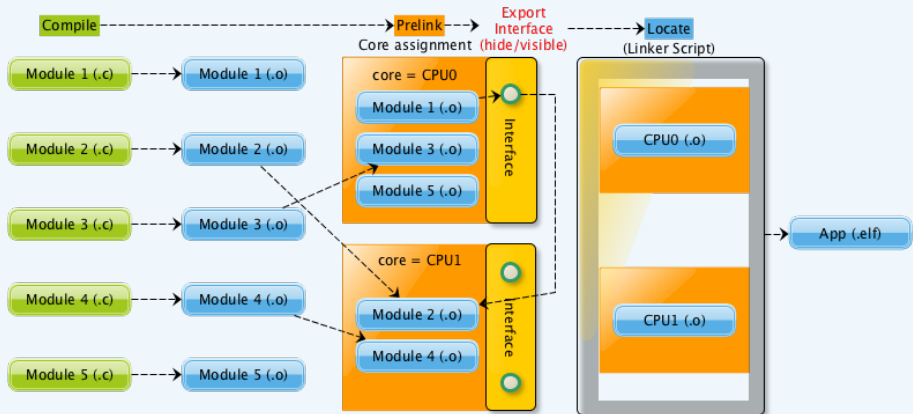
- Use existing Single-Core object files
- Core assignment on linker level
- Voter interfaces (hide/visibility)
- Locate and create one elf-file



Conclusion

HighTec tooling supports flexible assignment to cores. Even **heterogeneous** multi-cores are supported.

Workflow for building multi-core application HIGHTEC



Prelinking and Locating

Relocatable Linking

–warn-flags checks consistency of flags from input and output sections

```
$(LD) -warn-flags -export export_core1 --core=CPU1  
-r -o $@ $(CORE1_OBJS) -Map core1.map
```

```
$(LD) -warn-flags -export export_core2 --core=CPU2  
-r -o $@ $(CORE2_OBJS) -Map core2.map
```

Generate one resulting elf-File and locate it with <name>.ld

```
$(LD) -T tricore.ld -Wl,-Map,Mapfile.map -Wl,-cref -o a.elf  
$(STARTUP) $(SHARED) $(CORE0) $(CORE1) $(CORE2) $(GTM)
```



Linker is able to handle different architectures and to generate one resulting elf-file. The memory mapping of different memory views are handled by the linker.

Analyzing - CPU assignment

Hidden Symbols

```
readelf -s a.elf | grep HIDDEN
```

OBJECT	LOCAL	HIDDEN	[CPU1]	31	runcount
OBJECT	LOCAL	HIDDEN	[CPU2]	40	runcount

Global Symbols

```
readelf -s a.elf | grep VISIBLE
```

FUNC	GLOBAL	VISIBLE		10	lock_wdtcon
FUNC	GLOBAL	VISIBLE	[CPU2]	38	init_applproc2
FUNC	GLOBAL	VISIBLE	[CPU0]	20	init_applproc0
FUNC	GLOBAL	VISIBLE	[CPU1]	29	init_applproc1

Renamed Symbols

```
readelf -s a.elf | grep UNIQUE
```

OBJECT	UNIQUE	VISIBLE	[CPU2]	40	shared_runcount
--------	--------	---------	--------	----	-----------------

Advanced Multi-core Support



Features

- Implementation **conform** with existing ISO and EABI standards
- On **linker** level: Export interface with Hide and Visibility concept
 - Symbols are only locally visible per core
 - Exported symbols are globally visible to other cores
 - Exported symbols can be renamed
- Traceable assignment of code and data to cores
- Clear interfaces: reduce costs for testing; security aspect

Multi-Core Operating System

HighTec's advanced multi-core support helps structuring multi-core application. To solve the following requirements an adequate operating system is required.

Requirements

- Concurrent execution of task requires efficient mapping of tasks to cores to fulfill real-time requirements (Partitioning)
- Synchronization of access to shared resources like memory, peripherals (disabling interrupts not applicable)
- Prevent Dead-Locks and Race-Conditions (due to insufficient synchronization)
- Define intertask communication across multiple cores
- Use hardware based protection (MPU) to fulfill safety standards

PXROS-HR Fundamentals

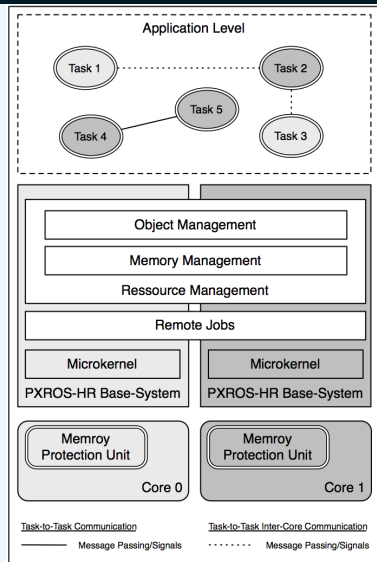
Basics

- Tasks live in capsules (MPU support)
- Intertask communication via
 - Events
 - Message object passing
- Micro-kernel architecture
- Interrupt lock-free
- Kernel is independent from peripherals
- Supports dynamic resource management

Multi-Core Operating System



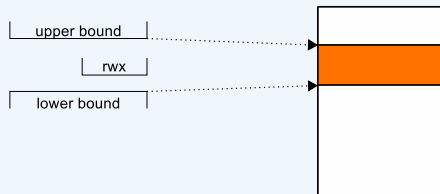
- Same programming model for single and multi-core can be used
- Tasks can be spread across several cores
- Per core one instance of the microkernel
- Use local memory for OS and task data
- Tasks and objects have unique identifiers
- OS supports memory partitioning based on MPU strategy



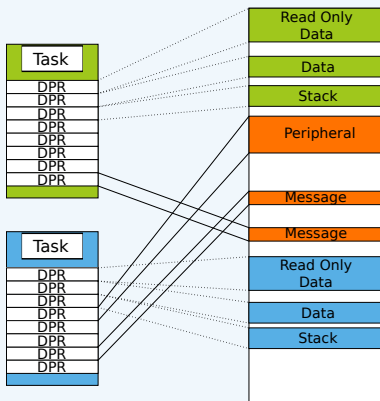
HW architecture - Memory Protection Unit HIGHTEC

- Register pair (**D**ata **P**rotection **R**egister) defines the memory area
- μ C has one set of MPU register pairs for supervisor and user modes

- Upper boundary
- Lower boundary
- Access rights



PXROS-HR system architecture (AURIX) HIGTEC



- Stack overflow is detectable by MPU
- Use MPU for task-specific context
- Objects can have time-outs or can be invalidated
 - ⇒ Safe sharing of resources and load balancing
- Peripheral access can be passed like a message
 - ⇒ Only one instance has exclusive access to peripherals
- Illegal access to peripherals are detected and prevented
 - ⇒ Generates a trap and allows adaptive error handling

Remote Jobs - Inter-Core Messaging

- All objects are stored in global memory
- Object pools per core (Quota)
- Object allocation/deallocation exclusively by owner core
- **Lock free** (enqueueing/dequeueing) of Messages (MPU protected)
- No interrupt and bus-system locks
- No spinlocks
- Multiple writer cores - Single reader core
- Reader core gets access rights to the object
- Objects will be released to the origin object pool

