**R-Car Series, 3rd Generation**

Capacity Aware Migration Strategy

**Introduction**

Purpose

Modern in-vehicle information systems are required to be capable of running several applications, including navigation, the playback of music and video, and the display of information on meters, at the same time. For use cases such as this, we recommend the use of a multi-core processing environment, which allows multiple CPUs to operate in parallel. Doing so improves the computational capability of the system and leads to the applications behaving in a user-friendly fashion.

Third-generation R-Car series products have two types of CPU: Arm® Cortex®-A57 and Cortex®-A53 cores (for a total of eight cores in the R-Car H3 and six in the R-Car M3-W/R-Car M3-W+). Applying a capacity aware migration strategy (CAS) is a way to obtain efficient utilization of multi-core processing. This document describes the CAS functionality and associated debugging.

Target Readers

Readers of this document are assumed to have general knowledge in the fields and specific technologies listed below.

* Engineering, logic circuits, microcontrollers, and Linux.
* The functionality of the multiple processor cores of R-Car H3, R-Car M3-W and R-Car M3-W+ products.
* The electrical specifications of the multiple processor cores of R-Car H3, R-Car M3-W and R-Car M3-W+ products.
* The functions of the BSP drivers for R-Car H3, R-Car M3-W and R-Car M3-W+ products.

Note

* Statements in relation to operating systems in this document apply to Yocto v3.7.0 from Renesas.

**Target Devices**

* R-Car H3
* R-Car M3-W/ R-Car M3-W+

**Contents**

[**1.** **Overview** 3](#_Toc517177424)

[**1.1** **What is CAS?** 3](#_Toc517177425)

[**1.2** **Outline of the CAS Functionality** 3](#_Toc517177426)

[**1.3** **CAS Operations in Outline** 4](#_Toc517177427)

[**2.** **Guide to Debugging** 5](#_Toc517177428)

[**2.1** **Monitoring Tasks with htop** 5](#_Toc517177429)

[**2.1.1** **Preparations in Advance** 5](#_Toc517177430)

[**2.1.2** **Procedure for Handling** 5](#_Toc517177431)

[**2.2** **Monitoring Tasks with ftrace** 6](#_Toc517177432)

[**2.2.1** **Preparations in Advance** 6](#_Toc517177433)

[**2.2.2** **Procedure for Handling** 6](#_Toc517177434)

[**3.** **Usage Notes on CAS Functionality** 7](#_Toc517177435)

[**3.1** **Load Balancing by the Scheduler** 7](#_Toc517177436)

[**3.2** **Realizing High Response Speeds in Task Processing** 8](#_Toc517177437)

[**3.2.1** **Processor Affinity as a Measure** 8](#_Toc517177438)

[**3.2.2** **Measure in the Form of Making Processes Realtime** 13](#_Toc517177439)

[**4.** **Disabling CAS Functionality** 14](#_Toc517177440)

[**Appendix** 15](#_Toc517177441)

[**A1.** **Times for Task Switching between the Cortex-A57 and Cortex-A53 Cores** 15](#_Toc517177442)

[**A2.** **Building an Environment for htop** 16](#_Toc517177443)

[**A3.** **Building an Environment for ftrace** 17](#_Toc517177444)

[**A4.** **Building an Environment for taskset** 18](#_Toc517177445)

# **Overview**

## **What is CAS?**

CAS is a function Renesas provides for use with asymmetric multiprocessing systems. It involves the consideration of SoC-dependent parameters in addition to support by the Linux kernel for asymmetric CPU capacities. Here, “CPU capacity” refers to the computational capability of a CPU, and “support for asymmetric CPU capacities” refers to functionality which is provided as part of the energy-aware scheduler (EAS) from Arm Limited.

Acceptance by the Linux kernel community of some parts of EAS

ARM has refactored EAS into several parts.

EAS

Capacity awareness

Frequency-invariance support

Core of EAS

Upstream Linux kernel 4.14

SoC-dependent parameters (device tree)

CAS

Renesas Linux   
BSP

Incorporation by Renesas of a facility for enabling big.LITTLE on our relevant BSP

Figure 1‑1 Configuration and Background of CAS

## **Outline of the CAS Functionality**

In devices in which the Cortex-A57 and Cortex-A53 cores are operating at the same time, CAS assigns tasks in consideration of the performance of the cores.

The standard multi-processor scheduler of Linux is the completely fair scheduler (CFS). This allows the effective use of multi-core processing by assigning tasks to CPUs such that the load on each CPU is equal (see CFS in Figure 1‑2).

On the other hand, the Cortex-A57 and Cortex-A53 cores differ in performance. Assigning more tasks to the Cortex-A57 cores with their greater computational capability instead of making the load on each CPU equal improves the performance of the system as a whole.

In CAS, the scheduler is made aware of the difference between the CPU capacities of Cortex-A57 and Cortex-A53 cores and gives higher priority to the Cortex-A57 cores in assigning tasks (see CAS in Figure 1‑2).

CPU 0

CPU 1

CPU 2

CPU 3

Cortex-A57

Cortex-A57

Cortex-A53

Cortex-A53

CPU 0

CPU 1

CPU 2

CPU 3

CFS

CAS

Task

Cortex-A57

Cortex-A57

Cortex-A53

Cortex-A53

Figure 1‑2 Assigning Tasks

## **CAS Operations in Outline**

CAS features the scheduling of tasks in awareness of the capacities of the target CPUs. The concrete behavior of the scheduler is described below.

The capacity of each of the CPUs is defined in the Linux kernel. The CPU capacities are divided into two parts in terms of the current assignment of tasks: the capacity in use and the idle capacity. When a task is assigned to a CPU, the part of the capacity of the given CPU that is in use is increased by the load of the task and its idle capacity is thus decreased by the same amount.

As the Cortex-A53 and Cortex-A57 cores have different computational capabilities, their maximum CPU capacities differ accordingly. Enabling CAS has the scheduler assign tasks to the CPU with the largest idle capacity. As a result, the scheduler gives higher priority to the Cortex-A57 cores in assigning tasks because the maximum CPU capacity of a Cortex-A57 core is greater than that of a Cortex-A53 core.

Specifically, an example of assigning tasks with different loads is given in Figure 1‑3. Note that the configuration of CPUs in the figure is simplified for the sake of explanation and is different from those in the actual devices.

Example of a low load: Idle capacity (a) of a Cortex-A57 core > Idle capacity (b) of a Cortex-A53 core

The scheduler starts assigning tasks to the Cortex-A57 cores.

Example of a medium load: Idle capacity (a) of a Cortex-A57 core > Idle capacity (b) of a Cortex-A53 core

The scheduler starts assigning tasks to the Cortex-A57 cores.

Example of a heavy load: Idle capacity (a) of a Cortex-A57 core < Idle capacity (b) of a Cortex-A53 core

The scheduler starts assigning tasks to the Cortex-A53 cores.

As described above, enabling CAS gives higher priority to the Cortex-A57 cores in assigning tasks because of the difference in the CPU capacity, and this allows capacity-aware scheduling of tasks for the CPUs.

Cortex-A57

Cortex-A57

Cortex-A53

Cortex-A53

CPU 0

CPU 1

CPU 2

CPU 3

Low load

Cortex-A57

Cortex-A57

Cortex-A53

Cortex-A53

CPU 0

CPU 1

CPU 2

CPU 3

Cortex-A57

Cortex-A57

Cortex-A53

Cortex-A53

CPU 0

CPU 1

CPU 2

CPU 3

Medium load

Heavy load

Idle capacity

(a)

(b)

(a)

(b)

(a)

(b)

Task

Figure 1‑3 Example of Assigning Tasks According to Loads Imposed on the Individual CPUs

# **Guide to Debugging**

Improving the performance of applications by using CAS strongly depends on the use cases of users. To make full use of CAS in your use case, monitoring how tasks are being assigned and the transitions and progress of tasks is very important. This section describes two methods of monitoring tasks (htop and ftrace) which are effective in debugging.

## **Monitoring Tasks with htop**

This covers htop as a simplified method of monitoring how tasks are being assigned.

htop is a tool with which loads imposed on CPUs, the state of assignment to CPUs of tasks in progress, and other information can be monitored on the console. htop can be used by adding it to the file system beforehand.

Note that htop displays information as a snapshot. Therefore, htop is not capable of continuously capturing the transitions and progress of tasks. To monitor continuous changes in the situation regarding tasks, we recommend using the ftrace tracing tool for the Linux kernel as the method of monitoring tasks (see section 2.2).

### **Preparations in Advance**

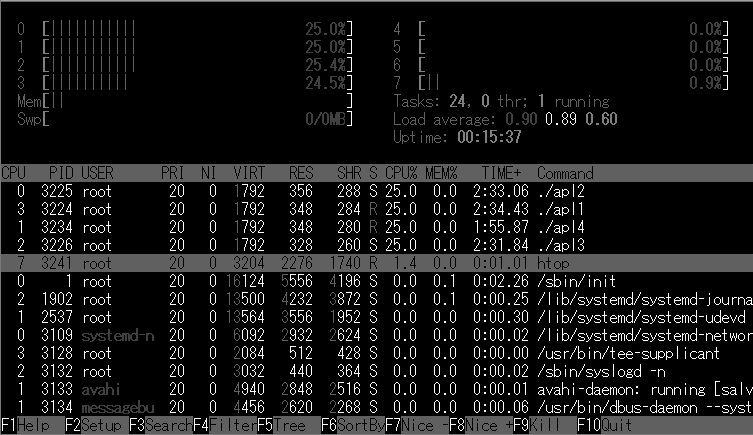
See appendix A2, Building an Environment for htop.

### **Procedure for Handling**

Enter the following command in a console window on the host PC.

$ htop

Executing the command displays the states of tasks in the console window as shown in the figure below.



**Figure 2‑1 Example of a Display Produced by htop for the R-Car H3**

For details on how to use htop, see the “Help” for htop.

## **Monitoring Tasks with ftrace**

To continuously monitor the detailed state of the assignment of tasks, such as the transitions and progress of tasks, use the ftrace tracing tool for the Linux kernel.

ftrace is the internal tracer of the Linux kernel and can output logs of events in kernel operations that have been executed.

Use trace-cmd to analyze the output logs and KernelShark to display results from trace-cmd in a GUI.

### **Preparations in Advance**

See appendix A3, Building an Environment for ftrace.

### **Procedure for Handling**

1. Handling of Tracing (Recording)

Enter the following command in a console window to start tracing while tasks to be monitored are running.

$ trace-cmd record -e sched -b 4000

To end tracing, press the Ctrl ^ C (‘Ctrl’ + ‘C’) keys in the console window in which you entered the command.

The traced data are stored as a file named ”trace.dat” in the current folder.

1. Display of traced data

Use KernelShark in the following way to display the results of tracing that were obtained.

Use the following command to analyze an obtained trace.dat file on a Linux host PC on which KernelShark has been installed.

$ kernelshark {position of the file/}trace.dat

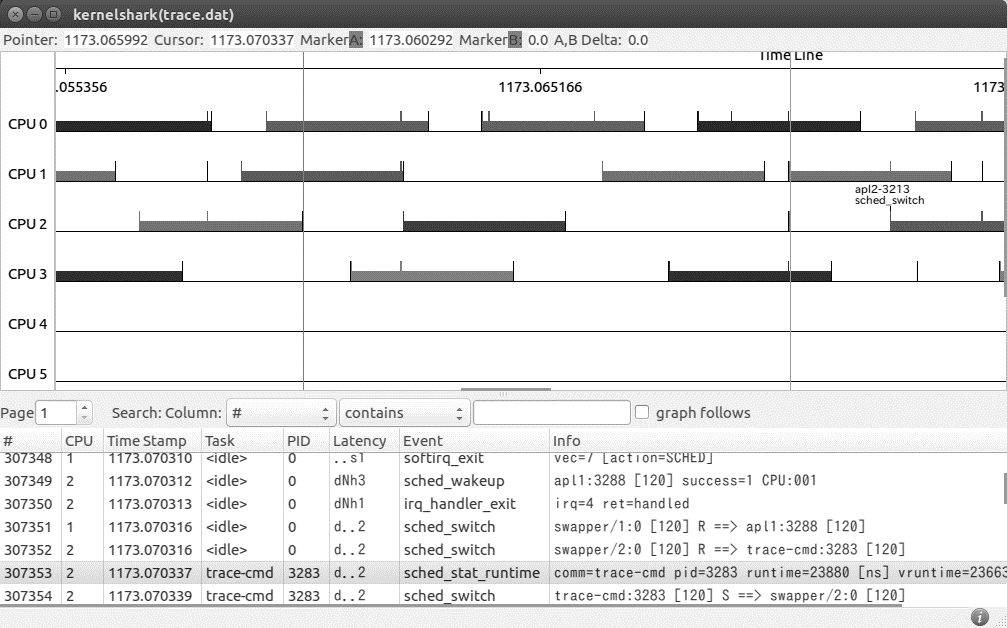


Figure 2‑2 Example of Using KernelShark to Display the Results of Tracing by ftrace  
 for the R-Car H3

# **Usage Notes on CAS Functionality**

## **Load Balancing by the Scheduler**

Although using CAS causes the scheduler to give higher priority to Cortex-A57 cores in assigning tasks, the scheduler may momentarily assign a task to a Cortex-A53 core due to the load balancing process which is the operation of the standard Linux scheduler, that is, completely fair scheduling (CFS). In such cases, the scheduler subsequently reassigns the task to a Cortex-A57.

This behavior of the scheduler leads to lowering of the processing performance, which may lead to poor response by a task handled in the way described above relative to the case where the task is run solely on a Cortex-A57 core, due to the difference in performance between the Cortex-A57 and Cortex-A53 cores.

Task

CPU 0

CPU 1

Cortex-A57

Cortex-A53

Cortex-A57

Cortex-A53

CPU 0

CPU 1

CPU 0

CPU 1

Cortex-A57

Cortex-A53

Normal state

A task is momentarily assigned to a Cortex-A53 core.

タスクを割り付け

Assignment of a task by CFS

The task is reassigned to a Cortex-A57 core.

Figure 3‑1 Momentary Assignment of a Task to a Cortex-A53

## **Realizing High Response Speeds in Task Processing**

### **Processor Affinity as a Measure**

Processor affinity is the method of allowing users to bind a specified application to a particular CPU or range of CPUs for running the application. Applying processor affinity in an environment where the Cortex-A57s and Cortex-A53s are booted up at the same time allows users to bind an application to the Cortex-A57s to prevent a Cortex-A53 from unexpectedly having to handle a heavy load of processing.

There are several methods of realizing processor affinity under Linux. They are listed and described in Table 3‑1. We recommend control group (cgroup) from the viewpoint of controlling processes in groups.

Table 3‑1 Realizing Processor Affinity

|  |  |
| --- | --- |
| **Method of Realizing Processor Affinity** | **Outline** |
| Cgroup | Cgroup is a Linux standard feature, where processes are classified into groups for control of the assignment of resources such as CPUs and memory. Cgroup can be handled through sysfs. |
| taskset | The taskset command can be used to realize processor affinity by specifying process IDs (PIDs) and the CPUs to run the processes from the command line. With affinity through the taskset command, CPUs must be assigned per process. |

We recommend the methods described up to this point in this document. Meanwhile, the Linux community suggests another method in which the bLsched daemon in the user space assigns tasks to CPUs based on the idea of realizing processor affinity. Details on bLsched are available on the Web page at the URL below. Refer to the page if you wish to know more it.

https://github.com/BayLibre/bLsched

#### **Procedure for Setting Cgroup**

Cgroup is a standard Linux feature in which processes are classified into groups for control of the assignment of resources such as CPUs and memory. Multiple groups can be created. Processes classified in the same group run by using specified resources. In an environment in which the Cortex-A57s and Cortex-A53s are booted up at the same time, applying cgroup allows the classification of applications as being for execution on a Cortex-A57 or a Cortex-A53 and assigning the applications to specified CPU resources as shown in Figure 3‑2.

Here, the setting to prevent a Cortex-A53 from unexpectedly having to handle a task intended for a Cortex-A57 by assigning applications solely to a Cortex-A57 is described as an example.

App. 1

App. 2

App. 3

Cortex-A53

Cortex-A53

Cortex-A57

Cortex-A57

Cortex-A57

Cortex-A57

CPU resources

App. 1

Application

Cortex-A53

Cortex-A53

CPU resources

App. 2

Applications

App. 3

Group classified for the Cortex-A53

Group classified for the Cortex-A57

Figure 3‑2 Example of Assignment of Applications through Cgroup

##### **Creating Groups for Use with Cgroup**

How to use cgroup to set up a group in which the Cortex-A57 cores handle tasks as the “big” group is described below.

Step 1: Confirming the CPUs that have been booted up  
Execute the following command to confirm the CPU number and type (Cortex-A57 or Cortex-A53).

$ cat /proc/cpuinfo

/\* The types of CPU can be confirmed from the CPU part in the result of execution as follows. Cortex-A57 -> 0xd07 and Cortex-A53 -> 0xd03 \*/

Step 2: Creating the “big” group

Execute the following command to create the “big” group.

$ mkdir /sys/fs/cgroup/cpuset/big

Step 3: Setting the memory node  
The memory node is a parameter for use in setting the assignment of memory to the cgroup. If you do not wish to specify the assignment of memory, set it to 0 as in our example.

$ echo 0 > /sys/fs/cgroup/cpuset/big/cpuset.mems

Step 4: Assigning CPU resources to the groups  
Register the CPU numbers for the Cortex-A57 processors confirmed in step 1 with the “big” group.

* Registration for the R-Car H3

$ echo 0-3 > /sys/fs/cgroup/cpuset/big/cpuset.cpus

/\* For the R-Car H3, the Cortex-A57s have been registered with the CPU numbers 0 to 3.\*/

* Registration for the R-Car M3-W/ R-Car M3-W+

$ echo 0-1 > /sys/fs/cgroup/cpuset/big/cpuset.cpus

/\* For the R-Car M3-W, R-Car M3-W+, the Cortex-A57s have been registered with the CPU numbers 0 and 1.\*/

##### **Assigning Applications for Use with Cgroup**

How to assign applications to the “big” group created on the previous page is described below.

Step 1: Confirming the PIDs of applications  
Execute the following command to confirm the PIDs for applications to be assigned with cgroup.

$ ps

/\* Confirm the PID lines in the result of executing the command. \*/

Step 2: Assigning the applications  
Execute the following command to assign applications to the “big” group. Replace [PID] in the command with the PIDs for the applications to be assigned before execution.

$ echo [PID] > /sys/fs/cgroup/cpuset/big/tasks

* Registration of assignment to the “big” group

Executing the following command releases the settings which had been made for assignment to the “big” group.

* Release from assignment to the “big” group

$ echo [PID] > /sys/fs/cgroup/cpuset/tasks

#### **Procedure for Setting taskset**

taskset is used to apply CPU affinity to processes in progress.

CPU affinity is a property of the scheduler and is used to link processes to specific CPUs in a system.

This is used to specify a CPU for each of the processes that are in progress.

To use taskset, taskset must be added when the file system is configured.

For details, see appendix A4.

A specified process is made to run on a specified CPU by executing the following command.

$ taskset -pc X [process ID]

/\* X is replaced with the CPU number (0, 1, 2, …). \*/

### **Measure in the Form of Making Processes Realtime**

Processor affinity can be used to prevent task switching unexpectedly assigning processes that are intended to be assigned solely to a Cortex-A57 to a Cortex-A53 through the method described in section 3.2.1, Processor Affinity as a Measure. Moreover, changing the attribute of a task to a realtime process is a further way of giving higher priority to the execution of the process.

Table 3‑2 shows the method of making a process realtime, taking chrt as an example.

Table 3‑2 Example of Setting the Priority Level for Realtime Processes

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Outline** | **Details** | **Command** |
| 1 | Setting the priority level for a realtime process and executing it | Give a unique priority level to a realtime process before executing it.\* | chrt -r X [process name]  X represents a priority level in the range from 0 to 99. Specify the name of the process to be run with that priority level in [process name].  Example: The command for starting the /usr/local/bin/sample process with the priority level specified as 99 is as follows.  chrt -r 99 /usr/local/bin/sample |

Note: \* Use root access to do this.

# **Disabling CAS Functionality**

If you wish to confirm the effect of applying CAS or CAS is not required, you can disable the CAS functionality.

To disable the CAS functionality, disable the big.LITTLE architecture.

For the concrete details of the method, see the description titled “Step 6 change the bootargs by U-Boot” in section 5, Confirm starting of U-Boot and Linux, of RENESAS\_RCH3M3M3NE3\_YoctoStartupGuide\_UME\_v3.7.0 (Linux Interface Specification Yocto recipe Start-Up Guide).

# **Appendix**

# **Times for Task Switching between the Cortex-A57 and Cortex-A53 Cores**

Table A1‑1 shows the times for task switching.

Table A1‑1 Times for Task Switching

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Cortex-A57**  **-> Cortex-A57** | **Cortex-A57**  **-> Cortex-A53** | **Cortex-A53**  **-> Cortex-A57** |
| Straddling of clusters | Same cluster, so none | Yes | Yes |
| Switching time | 14 µs | 19 µs | 21 µs |

<Measurement environment>

* Software: Yocto v3.7.0 (Linux BSP 3.6.2)
* SoC: Version 3.0 of the R-Car H3 on a Salvator-XS board
* Measurement method

1. Awaken any executable process (process A) which endlessly loops on CPU1.

2. Use the taskset command\* to switch the CPU on which process A runs to CPU2.

Note: \* taskset is a command used to change the settings for assigning processes to CPUs.

Measurement period

: Operating

: Suspended

1. CPU1 is running process A.
2. Process A is suspended.
3. The CPU to run process A is switched to CPU2.
4. Process A is awakened on CPU2.

A

CPU 1

Run queue

CPU 2

Run queue

A

A

Run queue

Run queue

CPU 2

Run queue

CPU 1

Run queue

CPU 2

A

Run queue

CPU 1

CPU 1

CPU 2

Run queue

Figure A1‑1 Flow of Task Switching

# **Building an Environment for htop**

To use htop, edit the given file in the way shown below and rebuild the file system.

[Add the following line to build/conf/local.conf.]

CORE\_IMAGE\_EXTRA\_INSTALL += "htop"

Figure A2‑1 Adding htop

# **Building an Environment for ftrace**

* + - * 1. Enabling ftrace

To enable ftrace, make the following setting in the kernel configuration.

Kernel hacking --->

[\*] Tracers --->  **\***

[\*] Kernel Function Tracer  **\***

[\*] Kernel Function Graph Tracer  **\***

Figure A3‑1 Kernel Configuration

Note: \* Check these boxes to enable ftrace.

* + - * 1. Installing trace-cmd in the file system.

To use trace-cmd, edit the given file in the way shown below and rebuild the file system.

[Add the following line to build/conf/local.conf.]

CORE\_IMAGE\_EXTRA\_INSTALL += "trace-cmd"

Figure A3‑2 Adding trace-cmd

* + - * 1. Installing KernelShark

If you intend to run KernelShark on the Linux host PC, use apt-get or your preferred commands to install KernelShark on the Linux host PC.

# **Building an Environment for taskset**

To use taskset, edit the given file in the way shown below and rebuild the file system.

[Add the following line to build/conf/local.conf.]

CORE\_IMAGE\_EXTRA\_INSTALL += "util-linux"

Figure A4‑1 Adding taskset

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
|  |  |
|  |  |  |  |
|  | March |  |  |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
|  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
| Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. |  |  |



|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |