ChipCoach Library (CCL) for AURIX™

Automated search for bugs and performance issues

About this document

Scope and purpose

Introduce the CCL concept, framework and checks for AURIX™

Intended audience

AURIX™ system and software developers.

Table of contents

About this document **1**
Table of contents **2**
1 Introduction **3**
2 CCL framework **5**
2.1 Recorder Lib 6
2.2 Configurator and Decoder Libs 6
2.3 CCL Python Utility Lib 8
3 CCL checks and checkpoints **9**
3.1 Basic considerations 9
3.1.1 Device operation phases (SOH) 9
3.1.2 Device connection requirements 9
3.2 CCL check design 9
3.2.1 Option 1: Keep the check very simple 10
3.2.2 Option 2: Consider all scenarios 10
3.2.3 Option 3: Focus on the normal scenario 11
3.3 CCL checkpoints 11
4 CCL for AURIX™ **12**
4.1 ChkAccen 12
4.2 ChkClc 12
4.3 ChkClkBasic 13
4.4 ChkCpuBasic 13
4.5 ChkCpuCache 13
4.6 ChkCpuLockstep 14
4.7 ChkCpuModAcc 14
4.8 ChkFlashWaitCycles 14
4.9 ChkIntBasic 15
4.10 ChkOcds 15
5 Configuration file **16**
6 Getting started **17**
6.1 Installation 17
6.2 Run CCL 17
6.3 Review results 17
Revision history **18**
Disclaimer **19**

1 Introduction

AURIX™ based systems can reach a very high complexity level which is far beyond a simple proof of correctness by a human being or by any mathematical or formal method. The AURIX™ TC39x documentation has about 7000 pages. The complexity of the SW can be even higher. This means there is high risk for bugs and performance issues.

Debugging starts when there is a symptom, but sometimes a bug has only an effect in very specific situations. A typical example is a wrong, but marginal programming of the flash access wait cycles. Such a system might perfectly work under lab conditions, but will fail under temperature stress or worse it will have a high FIT rate when it was already deployed to the field. On the other side, when the flash wait cycles are set too high, the performance is unnecessarily lowered.

For minimizing the debugging effort, it is desirable to find issues automatically, even if there is no symptom. The ChipCoach Library (CCL) provides a set of AURIX™ HW related checks and a generic framework which can be extended for similar system specific checks.

The checks that come with CCL will be extended by Infineon over time for cases with these characteristics:

- Issue is a very likely error a user will make
- Or the issue is hard to debug:
  - No obvious link between symptom and root cause
  - Symptom is very sporadic or only triggered by very special conditions
- Issue can be identified without specific knowledge about the SW

Note: *The set of checks in CCL will never be complete and there is no guarantee that a check will detect all possible variants of an issue. Iin many cases debugging effort will be reduced.*

CCL Overview

- Framework for automated analysis
  - Non-intrusive target access
  - Non-intrusive observation by HW trace with MCDS
- Python based
- Set of checks
- Can be integrated in debuggers and regression test environments
- Can be extended by users with system specific checks

Fundamental principles

- Non-intrusive checks by reading or observation by trace
- Trust only data from HW (register and memory reads and activities observed by tracing)
- No or minimum input parameters for checks
- Simple pass or error result
- Diff stable and complete result and report files
- Regression friendly
- Debugging friendly
- No GUI

1 Introduction

Typical CCL checks

- Proper implementation of a workaround for an erratum
- Proper programming sequences

Requirements for the System Under Observation (SUO)

- Start from reset needed for many CCL checks
- Phases:
  - **S**tartup
  - **O**S startup
  - OS **H**yperperiod
- Periodic behavior allows sequencing of different CCL checks

2 CCL framework

The CCL framework provides the basis for the different CCL checks in the library.

CCL framework requirements

- Target access
- Support of on-chip trace
- Full featured scripting including trace generation and analysis

These requirements are fulfilled by a typical high-end debugger. However, the scripting language API and thus the CCL check scripts will be specific for this debugger vendor. CCL uses the DAS (www.infineon.com/DAS) for target access and Python for scripting. The CCL framework is prepared to exchange this part by another tool HW, which exposes the same API.

CCL framework capabilities

Depending on the CCL check, different capabilities of the CCL framework are needed. In particular the trace features depend on the target device and the device access HW.

CCL framework architecture

- Recorder library for target access via a Device Access HW
- MCDS Configurator
- MCDS Decoder
- MCDS Analyzer (experimental)
- CCL specific Python utility library (reporting and error handling)

Figure 1 shows the architecture with the listed components.

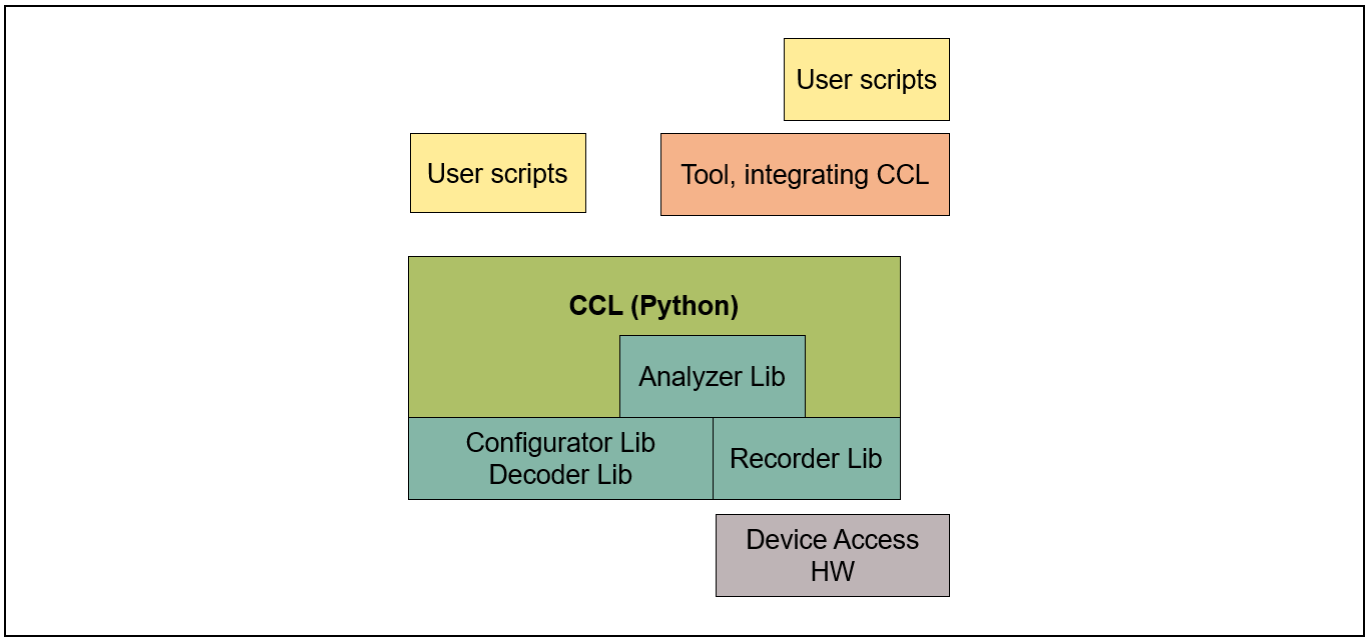


Figure 1 CCL Framework

2.1 Recorder Lib

As already menitioned CCL uses the DAS for target access. The CCL architecture is however supporting the option to use another tool hardware for target access. This is reflected in the CCL release scope (Figure 2).

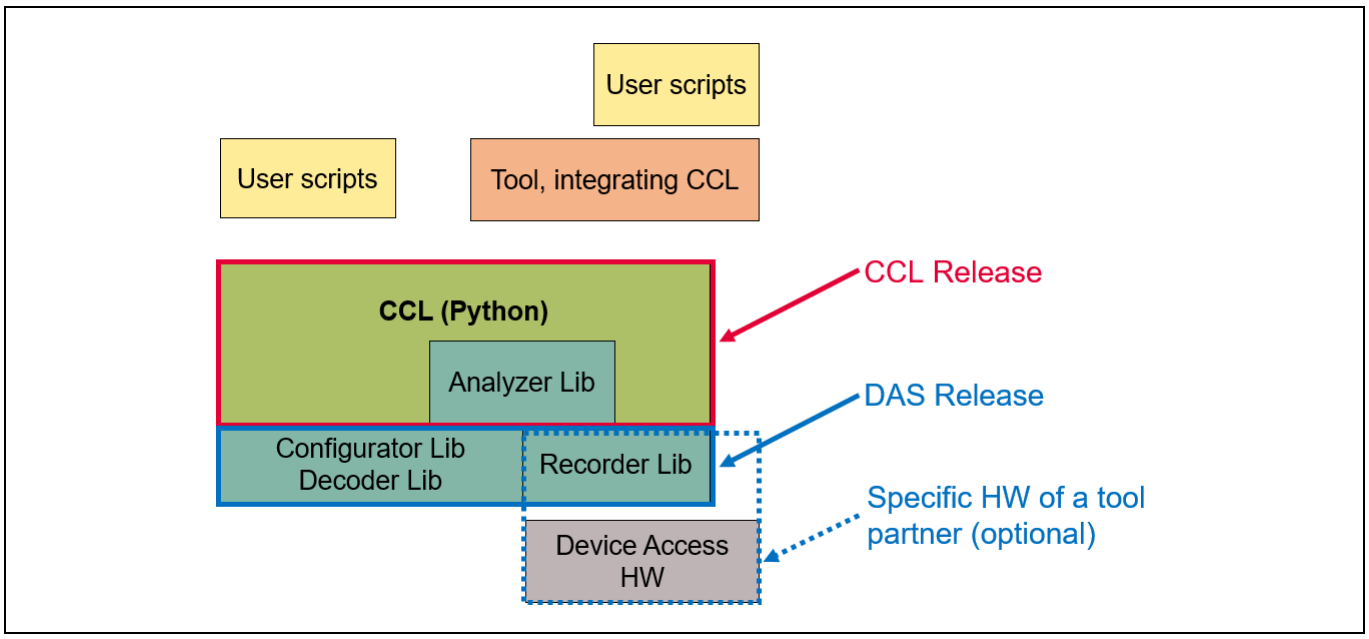


Figure 2 CCL Release scope

The Recorder Lib in combination with the device capabilities can support four different levels. For continuous trace the supported bandwidth can be limited by the physical device connection (e.g. DXCPL, DXCM – device access via CAN infrastructure) or the tool HW capabilities (maximum DAP frequency, DAP Wide Mode support, tool HW efficiency, etc.).

Table 1 Device connection capabilities

Capability		DAS
NO_TRACE	Only R/W access to device (e.g. device without MCDS)	yes
ONCHIP_TRACE_ONLY		yes
CONT_TRACE_UNRELIABLE	E.g. DAP miniWiggler with Windows PC	yes
CONT_TRACE_RELIABLE	Commercial tool HW	no

The very cost-effective Recorder Lib implementation from Infineon is based on the DAS infrastructure (www.infineon.com/DAS).

2.2 Configurator and Decoder Libs

MCDS is a very powerful trace module with hundreds of registers (Figure 3). The MCDS Configurator Lib is only needed for CCL checks which require tracing. The Configurator Lib allows to generate MCDS configurations based on abstract inputs. The Decoder Lib converts the binary trace data to an array of messages.

Figure 4 shows a typical MCDS configuration and trace. The program flow of CPU0 is observed along with all data accesses excluding the context save area. The trace recording is stopped with the trigger condition that a specific value is written to a specific bit of a specific register. This simple example demonstrates that it is possible with MCDS to record only relevant data by using trace qualification and triggering. This usually allows to capture the relevant trace information with the limited on-chip trace memories.

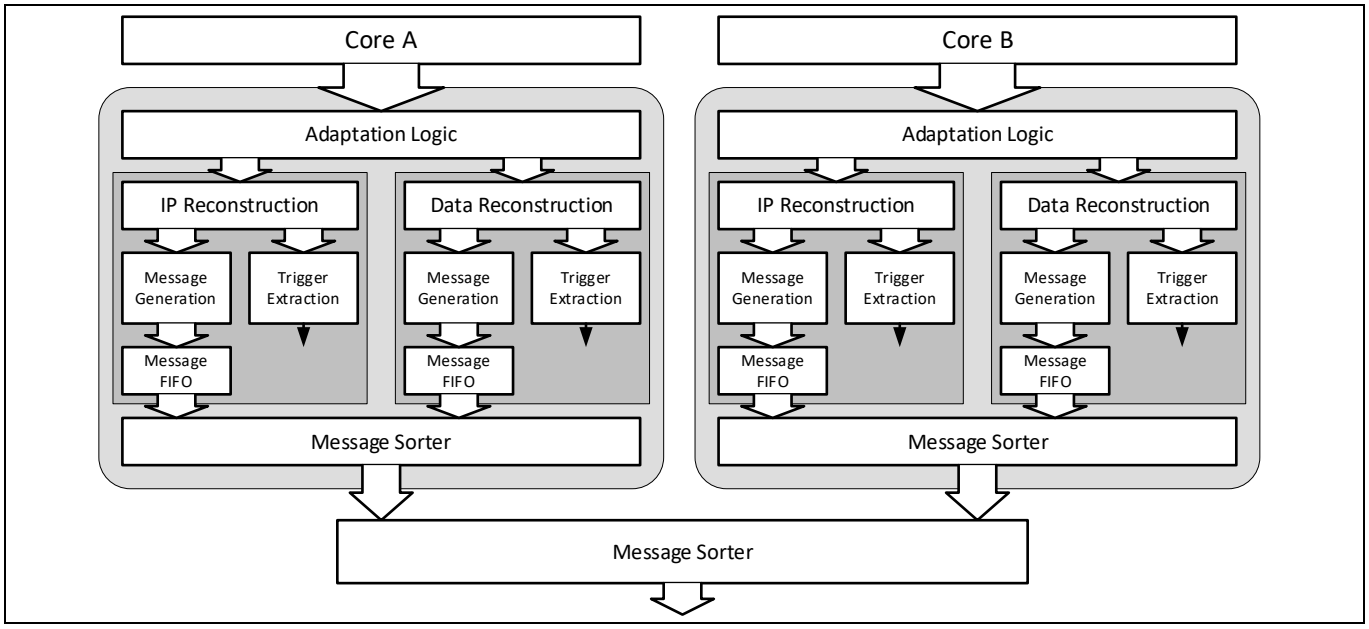


Figure 3 MCDS (Multi-Core Debug Solution) IP block

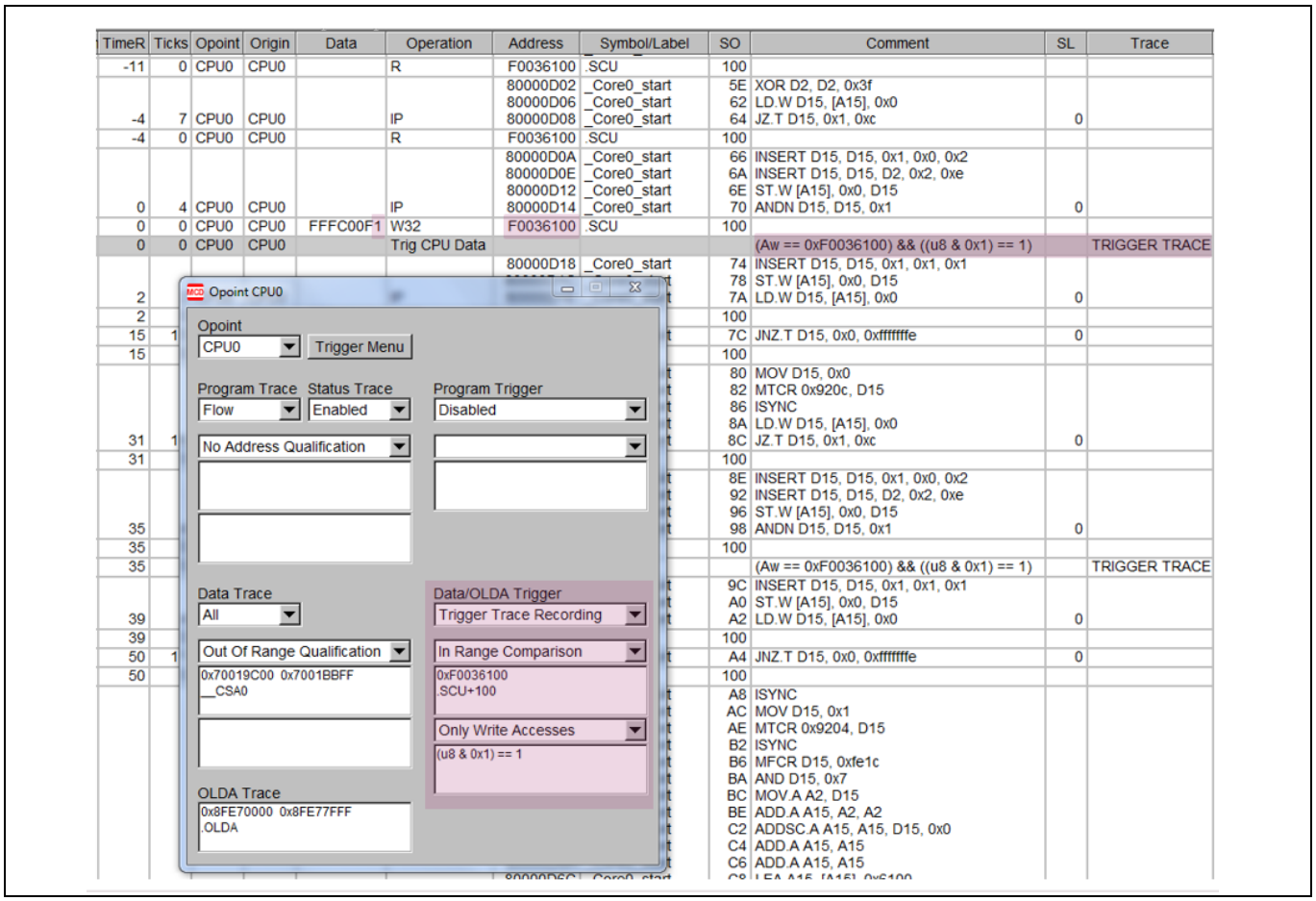


Figure 4 Typical MCDS trace configuration

2.3 CCL Python Utility Lib

This library standardizes several aspects of the CCL checks:

- Error handling (log and continue, stop)
- Logging
  - Human reading friendly reports
  - Debugging friendly result logs
  - Diff friendly and complete (e.g. with added time stamps) variants of reports and logs
- Debug support (e.g. storage of intermediate data, logs of compiled libs, diff friendly logs)
- Support libs (device utilities e.g. CPUn abstraction)
- Symbol handling (SFR addresses, ELF files, etc.)
- CCL global and check specific parameters (e.g. min/max thresholds for execution times)

3 CCL checks and checkpoints

3.1 Basic considerations

3.1.1 Device operation phases (SOH)

The device operation can be outlined in three parts:

- **S**tartup
- **O**S startup
- OS **H**yperperiods

CCL checks assume that many vital configurations are made in the startup phase and then never changed until the next reset.

Table 2 Device operation phases

Level	Configured at the end of this phase	Parameter
Startup	Clock system NVM system Peripheral protections Caches	
OS startup	CPU memory protection	

Note: In CCL this SOH encoding is used to mark the validity of parameters and results in their name.

3.1.2 Device connection requirements

The target for a specific CCL check is to use the lowest possible HW capability level. Ideally this allows to run this check with a production device without trace. Table 3 lists the different device connection requirement levels and their parameter. This table corresponds to Table 1. The current AURIX™ checks from Infineon only use level NO_TRACE and ONCHIP_TRACE_ONLY.

Table 3 CCL check device connection requirements

Level	Comment	Parameter
NO_TRACE	Only R/W access to device needed.	
ONCHIP_TRACE_ONLY	Only on-chip trace needed.	Trace buffer size
CONT_TRACE_UNRELIABLE	Continuous trace needed but it can be unreliable	Trace bandwidth
CONT_TRACE_RELIABLE	Reliable continuous trace needed	Trace bandwidth

3.2 CCL check design

The wrongly set flash wait cycles example, which was already mentioned, fulfills all three points for a needed CCL check:

- Issue is a likely error a user will make
- The issue can be hard to debug
- Issue can be identified without specific knowledge about the SW

A system with marginally programmed flash access wait cycles might perfectly work under lab conditions, but will fail under temperature stress or worse it will have a high FIT rate when it was already deployed to the field. On the other side, when the flash wait cycles are set too high, the performance is unnecessarily lowered.

This example will be used to explain how to design a good CCL check.

CCL check options:

- Keep the check very simple
- Consider all scenarios
- Focus on the normal scenario

3.2.1 Option 1: Keep the check very simple

The very simple CCL check is just doing the following steps:

- Hot attach to the running device
- Measure the clock frequency
- Read the wait cycles settings from all flash modules
- Analyze that the settings are correct

This check will not need trace and it can be executed at any time without a reset.

3.2.2 Option 2: Consider all scenarios

The simple CCL check from the previous section only checks the final state. This assumes that the state is static and on the way from reset to this state there are no transient wrong settings. The following sequence considers that the scenario could be much more complicated, intentionally or unintended (bug).

- From reset onwards trace all relevant accesses to the clock system and the flash module control registers
- Measure the final clock frequency
- Analyze that the settings are correct at every point in time

This check needs trace and it starts with a reset. The analysis of the trace data is not simple. For instance, to determine the clock frequency over time the crystal frequency needs to be known. Since CCL input parameters shall be avoided, this can only be done by measuring the final clock frequency and tracing or reading the associated clock system settings. With this information the intermediate clock frequencies can be determined as well.

In addition, the correctness analysis needs to consider that the flash wait cycles may never be too little, but during the setup, they do not need to be always optimum. Only the final operating settings should be optimal. And this gets even more complicated if the assumption is given up, that there is just a linear configuration path from reset to the operating state. When considering all scenarios, the CCL check will also allow dynamic reconfigurations of the clock system e.g. to a lower frequency.

Also the case needs to be considered that other CPUs than CPU0 make redundant and/or conflicting configurations, which should be detected.

This CCL check option is quite complicated to implement, test and maintain. It also bears the risk that a complicated sequence passes this check since it follows the rules of this check, for instance, a bug leads to an unnecessary double configuration

3.2.3 Option 3: Focus on the normal scenario

In a normal startup scenario, the user startup SW on CPU0 will configure the clock system and the flash wait cycles in an efficient way. The following sequence which will take less than 5ms:

- Ramp up the clock to the target clock frequency with one or several steps
- Adapt the flash wait cycles accordingly

In this scenario the CCL check is based on similar steps as in option 2:

- From reset onwards trace all relevant accesses to the clock system and the flash module control registers
- Measure the final clock frequency
- Analyze that the settings are correct at any point in time

The difference to option 2 is that the analysis is much more straightforward. The trace recording is restricted to an 8ms time window after reset to support also a small on-chip trace buffer (e.g. miniMCDS). Later it is statically checked that the configuration has not changed. Depending on the device, the reset value for the flash wait cycles can be already large enough or even larger than needed for the target frequency.

The CCL check design pattern used in option 3 is the clear recommendation for other CCL checks. It has the right balance between flexibility and implementation complexity. It also guides the user to a standard sequence for the clock and flash wait cycles configuration.

3.3 CCL checkpoints

A check can cover one or more checkpoints. The coverage of checkpoints can depend for instance on the availability of trace. The concept of checkpoints handles this uncertainty, that the successful execution of a check does not necessarily mean that all checkpoints of this check were covered.

CCL checkpoint

- Refers to a clearly defined behavior or configuration which is checked (e.g. static value of flash wait cycles)
- The set of checkpoints is the basis for coverage metrics (checks are not)
- A checkpoint can pass, fail or is not covered
- A checkpoint can have a confidence level which is for most checkpoints 100% in case of pass or failed
- A checkpoint can have one or more parameters (e.g. tolerated additional flash wait cycles than needed)
- A checkpoint parameter can have a default value (e.g. one flash wait cycle more than needed is tolerated)

4 CCL for AURIX™

This chapter lists the checks which are available in the associated CCL version.

Table 4 gives an overview. The colmn AF lists for which AURIX™ family the check is supported (TC3xx, TC4xx). The column TRP lists, whether this check needs trace (T/t), will do an initial reset (R/r) or needs parameters (P/p). A capital letter for T and R indicates that trace and reset respectively are mandatory for this check. In case of a small letter at least a subset of the check can be executed if these prerequisites are not fulfilled. For parameters a small letter p indicates that the default values of the parameters are normally sufficient to pass.

Table 4 CCL checks and checkpoints for AURIX™

Check	AF	TRP	Comment
ChkAccen	3	- - P	Checks ACCEN register settings
ChkClc	3	- - P	Checks CLC register settings
ChkClkBasic	3	t r p	Checks the clock system setup
CpClkBasic		- - -	Checks the basic clock system setup
CpClkStartup		T R p	Check the clock system configuration during startup
ChkCpuBasic	3	- - P	Basic CPU configuration checks and if running
ChkCpuCaches	3	- r p	Checks if CPU caches are enabled and for which segments
ChkCpuLockstep	3	- - p	Checks if CPUs are in lockstep mode
ChkCpuModAcc	3	- r p	Checks module accesses by CPUs
CpCpuModAccData		- r p	Segments (e.g. D), memories and modules used for data
CpCpuModAccCode		- r P	Segments (e.g. C) and memories used for code fetch
CpCpuModAccPfLocal		- r p	Checks if CPUs use only local PFlash
CpCpuModAccFamily		- r p	Checks if SW is compatible to other family devices
ChkFlashWaitCycles	3	t r p	Checks the configured flash wait cycles
CpDataFlashWait		- - p	D flash wait cycles
CpPFlashWait		- - p	P flash wait cycles
CpFlashWaitStartup		T R p	Check wait cycles settings during startup
ChkIntBasic	3	- - P	Basic check of all configured interrupts
ChkOcds	3	- r p	Checks if CPUs write to debug and trace resources

4.1 ChkAccen

- Reads the ACCEN0/1 register of all modules
- Lists modules with clock enabled and ACCEN registers having their reset value
  An error is raised if the associated param list is not matching
- Lists for each configured master the modules it can access
  An error is raised if the associated param list is not matching
- Lists for each used master group the modules it can access
  This is for human inspection. The information is redundant to the master/module lists

4.2 ChkClc

- Reads the CLC (clock control) register of all modules
- Lists all modules where SW has enabled the clock (CLC reset value is disabled)

- Lists all modules where SW has disabled the clock (CLC reset value is enabled)
- An error is raised if there are no matching param lists

4.3 ChkClkBasic

- Checks the clock system configuration

CpClkBasic

- Checks that this is a normal configuration with a crystal as clock source
- Checks that PLLs are locked
- Determines the frequencies of differenct clock domains

CpClkStartup

- Determine time until clock is ramped up by trace analysis
- Checks that last write to CCUCON0/5 has UP bit set
- Checks that only CPU0 is writing the CCU clock control registers
- Checks some configuration sequence rules
- Calculates the absolute time for the different points in the clock frequency ramp
- Measures the steepness in µs/MHz for the frequency steps

4.4 ChkCpuBasic

- Reads the SYSCON and DBGSR registers of all CPUs
- Lists all used CPUs
- Lists all unused CPUs
  An error is raised if the associated param list is not matching
- Lists all not running CPUs
  An error is raised if the associated param list is not matching
- Lists all CPUs not using memory protection
  An error is raised if the associated param list is not matching
- Lists all CPUs not using temporal protection
  An error is raised if the associated param list is not matching
- Lists all CPUs where the FCD (Free Context List Depleted) trap bit is set and raises an error in this case

4.5 ChkCpuCache

- Reads the DCON0, PCON0 and PMA0, PMA1 registers of all CPUs
- Lists all CPUs where the program/data cache is disabled
  An error is raised if:
  - the associated param list is not matching
  - or the param list is empty and the cache is disabled for a used CPU
- Lists all used CPUs where the program/data cache is using special segments (not 8 and 9)
  An error is raised if the associated param list is not matching

Not yet implemented:

The CPU0 entry to user code is in the uncached A segment. So the normal behavior is that after a few instructions in segment A, the rest of the execution is from segment 8. This can be checked using OCDS triggers.

Need to consider also the cases where code is executed from PSPR. Maybe have another check for all memory locations from which code is being executed.

4.6 ChkCpuLockstep

- Reads the SCU_LCLCON0/1 register
- Lists all CPUs with lockstep where the lockstep is disabled
  An error is raised if the associated param list is not matching

4.7 ChkCpuModAcc

- Identifies module accesses by CPU software
- Only CPU load/store and code fetch are included (no CSA context operations)
- The OCDS trigger logic (4 ranges) of all CPUs is used in parallel to minimize the execution time

CpCpuModAccData

- Lists all CPUs using segment D
- Lists all CPUs using segment A (uncached by default) for constants access in Pflash
- Lists per CPU all modules (memories or peripheral) which are accessed by load or store operations

CpCpuModAccCode

- Lists all CPUs using segment C for fetching code from PSPR
- Lists all CPUs using segment A (uncached by default) for fetching code from Pflash
- Lists per CPU all used code memories

CpCpuModAccPfLocal

- Lists all CPUs not using CPU local Pflash

CpCpuModAccFamily

- Lists for each device of the family, which modules are missing
- For memories, e.g. DSPR the different size for different devices is considered
- Different peripheral configurations (e.g. GTM) are not considered

4.8 ChkFlashWaitCycles

CpPFlashWait, CpDataFlashWait

- Reads the DMU_HF_PWAIT/DWAIT register
- Calculates the required program/data flash and ECC wait cycles
  The measured the clock frequency is used as input
- Compares this to the wait cycles configured in PWAIT/DWAIT and raises an error if
  - The configured wait cycles are smaller
  - The configured wait cycles are larger than is value and a margin given by a parameter

CpFlashWaitStartup

- Checks that the wait cycles are always sufficient during the startup phase

4.9 ChkIntBasic

- Lists all configured interrupt sources per service provider
  - Item is source name and priority
  - List is sorted by priority
  - If not empty the associated param list is compared
- Lists duplicated used priorities per service provider
  - Duplicates will raise an error
  - This can be waived with an associated param list
- Lists interrupts mapped to non-existing service providers
- Lists all interrupt trigger overflows IOV bit in SRC set)

4.10 ChkOcds

- Identifies CPU software writes to on-chip debug and trace modules
- This includes CBS, CPU debug register and MCDS
- CBS ACCEN0, IOSR, COMDATA, OIFM and TRIGS register writes are allowed per default parameter

5 Configuration file

Chip coach library (CCL) checks, are compared agains some reference in order to assess if the checks is passed or not. This reference have some default values that depending on the application and configuration running on the device may not be the desired ones.

CCL allows the user to customize these values using a JSON configuration file. This configuration file can be used to change the checks' default reference values by user defined ones.The parameters that are not included in the configuration file will keep the default reference value.

The configuration file should contain key-value pairs, keys that represent the parameter, and values that will be the reference for the checks. The possible values can be integers, strings and lists of the previous.

```
{
    "CclTc3xxCpPFlashWait.param_AccessCyclesMarginMax": 4,
    "CclTc3xxCpCpuModAccData.param_DataModByCpu" : [["ASCLIN0"],[],[],[],[],[]],
    "CclTc3xxCpCpuBasic.param_CpuMemProtNotEn": [0, 1, 2, 3, 4, 5]
}
```

Figure 1 Figure 1Configuration file example

6 Getting started

6.1 Installation

Install DAS V7.3 from here:
www.infineon.com/DAS

Install python version 3.6
https://www.python.org/downloads/release/python-365/
Note: Run the python installer with the Windows PATH variable option

Unzip the CCL package to a local drive.

Start ccl/install.bat

6.2 Run CCL

Connect the device via the DAS infrastructure. This is just the USB cable for Infineon evaluation boards or use a DAP miniWiggler (www.infineon.com/DAS) for other target boards.

Start ccl/run.bat [configuration file path]

This will create a folder ccl/LogsRun and store the results in a device specific subfolder.

If they do not exist the two folders <device>/ref and <device>/trace are created there.

The reference log files will be used for a comparison with future runs

The trace files can be viewed with the MCDS Trace Viewer tool (included in DAS installation)

6.3 Review results

The four log files follow the requirements for human reading but also diff friendliness for a very fast review of new results. The recommended procedure is to carefully review the CclLog_Report.txt file and then use a copy of CclRun.py for the following actions:

- Set check and checkpoint parameters in myParamForChecks()
- Remove checks in myCheckSelection(), if there is a justification for this

In many cases the output in CclLog_Raw_DiffStable.txt is a good source for these parameters.

The quick but not very clean variant is to rely on a reviewed reference in <device>/ref and ignore the errors for the individual checks.

In any case a good diff tool is the basis for fast analysis of new devices and new software.

Revision history

Document version	Date	Description of changes
V1.0	2021-11-25	First version
V1.1	2023-06-23	Configuration file chapter added

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-06-23

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2023 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AP32585

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.