ETSI TS 103 681-4 V1.1.1 (2020-03)

TECHNICAL SPECIFICATION

**Reconfigurable Radio Systems (RRS);
Radio Equipment (RE) information models and protocols
for generalized software reconfiguration architecture;
Part 4: generalized Radio Programming Interface (gRPI)**

Reference

DTS/RRS-0225

Keywords

architecture, interface, radio, SDR, software, system

Contents

Intellectual Property Rights

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Reconfigurable Radio Systems (RRS).

The present document is part 4 of a multi-part deliverable covering the Radio Equipment (RE) information models and protocols, as identified below:

Part 1: "generalized Multiradio Interface (gMURI)";

Part 2: "generalized Reconfigurable Radio Frequency Interface (gRRFI)";

Part 3: "generalized Unified Radio Application Interface (gURAI)";

Part 4: "generalized Radio Programming Interface (gRPI)".

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The scope of the present document is to define the generalized Radio Programming Interface (gRPI) for radio equipment reconfiguration. The work is based on the Use Cases defined in ETSI TR 103 585 [i.1], on the system requirements defined in ETSI TS 103 641 [1] and on the radio reconfiguration related architecture for radio equipment defined in ETSI TS 103 648 [i.2].

The present document will be based on ETSI EN 303 146-4 [i.4] and provide a generalized interface definition for the generalized Radio Programming Interface (gRPI).

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://docbox.etsi.org/Reference/.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

[1] ETSI TS 103 641: "Reconfigurable Radio Systems (RRS); Radio Equipment (RE) reconfiguration requirements".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI TR 103 585: "Reconfigurable Radio Systems (RRS); Radio Equipment (RE) reconfiguration use cases".

[i.2] ETSI TS 103 648: "Reconfigurable Radio Systems (RRS); Radio Equipment (RE) reconfiguration architecture".

[i.3] Directive 2014/53/EU of the European Parliament and of the Council of 16 April 2014 on the harmonisation of the laws of the Member States relating to the making available on the market of Radio Equipment and repealing Directive 1999/5/EC.

[i.4] ETSI EN 303 146-4: "Reconfigurable Radio Systems (RRS); Mobile Device (MD) information models and protocols; Part 4: Radio Programming Interface (RPI)".

[i.5] ETSI TS 103 681-1: "Reconfigurable Radio Systems (RRS); Radio Equipment (RE) information models and protocols for generalized software reconfiguration architecture; Part 1: generalized Multiradio Interface (gMURI)".

[i.6] ETSI TS 103 681-2: "Reconfigurable Radio Systems (RRS); Radio Equipment (RE) information models and protocols for generalized software reconfiguration architecture; Part 3: generalized Unified Radio Application Interface (gURAI)".

[i.7] ETSI TS 103 681-3: "Reconfigurable Radio Systems (RRS); adio Equipment (RE) information models and protocols for generalized software reconfiguration architecture; Part 3: generalized Unified Radio Application Interface (gURAI)".

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

Abstract Processing Element (APE): abstracts computational resource that executes any computations downloaded from Radio Library

NOTE: APE is connected with input and output DOs. APE is reactive. Any computations are started if all input DOs are filled with real data.

basic operations: operations either provided by the Radio Library and/or UDFB Set to eRVM or by the Radio Library and/or RVM/eRVM Configcodes to RVM

NOTE: Each Basic Operation is mapped to a corresponding APE in the case of eRVM or mapped to a corresponding APE or RVM/eRVM in the case of RVM.

data flow chart: reactive data flow computational model consisting of data and operators where data are connected with operators

NOTE: Operators abstract computations. They are triggered by full data. Results of operator computations are written in connected output data if they are empty.

Data Object (DO): typeless token abstracting any type of data

NOTE: DO provides a container for storing data. It can be empty if no data in the container or it can be full if there is data in the container. DO is allocated in the infinite and flat memory. Any RVM has access to this memory. One or a few APEs from RVM can be connected with DO. DO acknowledges connected APEs about its status whether it empty or full.

dynamic operation: operation that is performed by allocating the computational resources during run-time for each APE required executing the given operation

NOTE 1: The resources are deallocated upon completion of the corresponding operation.

NOTE 2: Dynamic operation is available only in the case of RERC-7 defined in ETSI TS 103 641 [1]. In other words, dynamic operation is needed when RA requires the dynamic resource sharing.

native radio library: library providing platform-specific description of each SFB that represents the target platform hardware

port configuration: specification of the number of APEs inputs and outputs

Radio Equipment (RE): *"an electrical or electronic product, which intentionally emits and/or receives radio waves for the purpose of radio communication and/or radiodetermination, or an electrical or electronic product which must be completed with an accessory, such as antenna, so as to intentionally emit and/or receive radio waves for the purpose of radio communication and/or radiodetermination"*.

NOTE: The definition above is as defined in the Radio Equipment Directive, Article 2(1)(1) [i.3].

radio library authority: authority empowered to decide which components can be registered as new SFBs

NOTE: Any suitable organization can take the role of a Radio Library Authority. The choice of the organization is beyond the scope of the present document.

Radio Virtual Machine (RVM): abstract machine that supports reactive and concurrent executions

NOTE: A RVM may be implemented as a controlled execution environment that allows the selection of a trade-off between flexibility of base band code development and required (re-)certification efforts.

Radio Virtual Machine Runtime Environment (RVM RE): software that allows running Radio Applications that might be Configcodes or executable codes

reference radio library: library providing normative definition of each SFB

NOTE: There may be multiple such Reference Radio Libraries. For a given RA, a unique Reference Radio Library is used.

Software Intermediate Representation (SWIR): RA representation as data flow chart

NOTE: SWIR file contains information on all terminal objects, their parameters (cost, implement function, size, etc.) and connections (links, access type, source and destination).

terminal operation: operation that will always be executed without any other interruption

NOTE 1: Furthermore, terminal operation cannot be decomposed into smaller operations.

NOTE 2: "Terminal operations" are equivalent to "atomic operations", but additionally it indicates that a hierarchy is being used in which the "terminal operations" are on the lowest level of hierarchy and they can be part of another operation.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

AOT	Ahead-Of-Time
APE	Abstract Processing Element
ASF	Abstract Switch Fabric
CC	Configcodes Counter
CSL	Communication Services Layer
CU	Control Unit
DO	Data Object
eRVM	elementary RVM
eSFB	elementary SFB
FB	Functional Block
FBRI	FB Reusability Index
FFT	Fast Fourier Transform
gMURI	generalized Multiradio Interface
gRPI	generalized Radio Programming Interface
gRRFI	generalized Reconfigurable Radio Frequency Interface
gURAI	generalized Unified Radio Applications Interface
HD	Hardware Dimension
HW	Hardware
ID	IDentification
IFFT	Inverse Fast Fourier Transform
IR	Intermediate Representation
JIT	Just-In-Time
LCF	Last Configuration Flag
NAF	Next Address Flag
NAPE	Number of Abstract Processing Elements
NCAO	Next Configcode Address Offset
NDO	Number of Data Objects
NOP	No OPeration

RA	Radio Application
RAP	Radio Application Package
RAT	Radio Access Technology
RCF	Radio Control Framework
RE	Radio Equipment
RF	Radio Frequency
RLA	Radio Library Authority
ROS	Radio Operating System
RPI	Radio Programming Interface
RVM RE	RVM Runtime Environment
RVM	Radio Virtual Machine
SD	Software Dimension
SFB	Standard Functional Block
SWIR	SoftWare Intermediate Representation
UDFB	User Defined Functional Block
UML	Unified Modelling Language
URA	Unified Radio Applications
VDO	Virtual Data Object
VHDL	Very high speed integrated circuit Hardware Description Language
XML	eXtensible Markup Language
XOR	eXclusive OR

4 Introduction

A reconfigurable RE is capable of running multiple radios simultaneously, changing the set of radios by loading new Radio Application Packages (RAP) and setting their parameters. All Radio Applications (RAs) are called Unified Radio Applications (URAs) when they exhibit a common behaviour from the reconfigurable RE's point of view in ETSI TS 103 648 [i.2]. In order to run multiple URAs, the reconfigurable RE will include Communication Services Layer (CSL), Radio Control Frameworks (RCFs), Radio Platforms and 4 sets of interfaces for their interconnection.

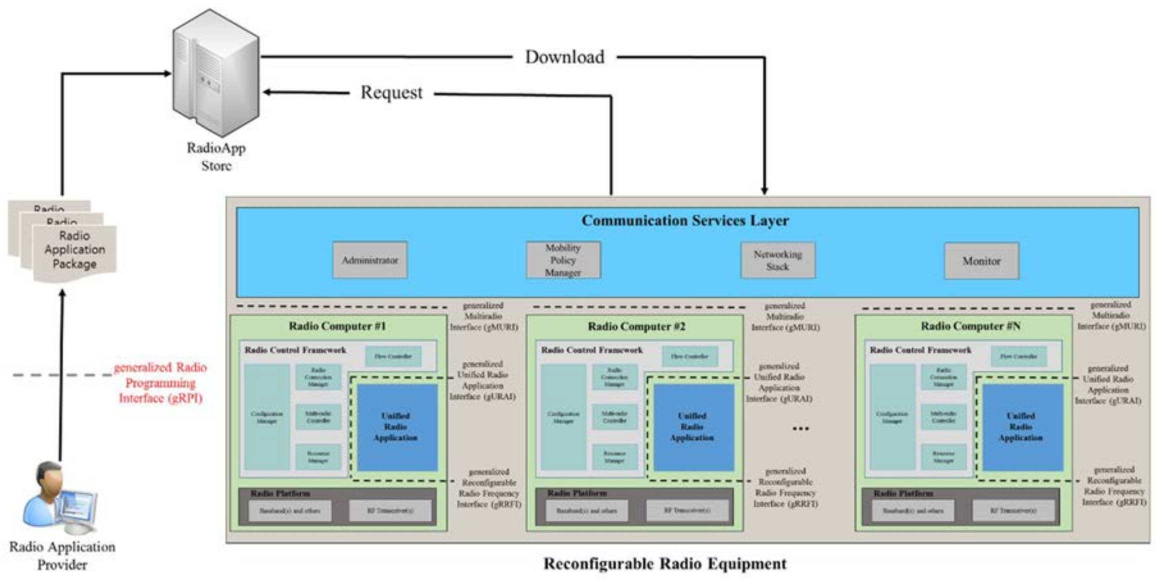

Download
Request
RadioApp Store
Radio Application Package
generalized Radio Programming Interface (gRPI)
Radio Application Provider
Communication Services Layer
Administrator
Mobility Policy Manager
Networking Stack
Monitor
generalized Multiradio Interface (gMURI)
Radio Computer #1
Radio Computer #2
Radio Computer #N
Radio Control Framework
Unified Radio Application
generalized Unified Radio Application Interface (gURAI)
generalized Reconfigurable Radio Frequency Interface (gRRFI)
Radio Platform
...
Reconfigurable Radio Equipment

Figure 4.1: Four sets of interfaces for Reconfigurable RE

Figure 4.1 illustrates the Reconfigurable RE architecture with the 4 sets of interfaces, i.e.:

- gMURI for interfacing CSL and RCF (in ETSI TS 103 681-1 [i.5]);
- gRRFI for interfacing URA and RF Transceiver (in ETSI TS 103 681-2 [i.6]);
- gURAI for interfacing URA and RCF (in ETSI TS 103 681-3 [i.7]);
- gRPI for allowing an independent and uniform production of RAs which is the scope of the present document.

The present document defines gRPI.

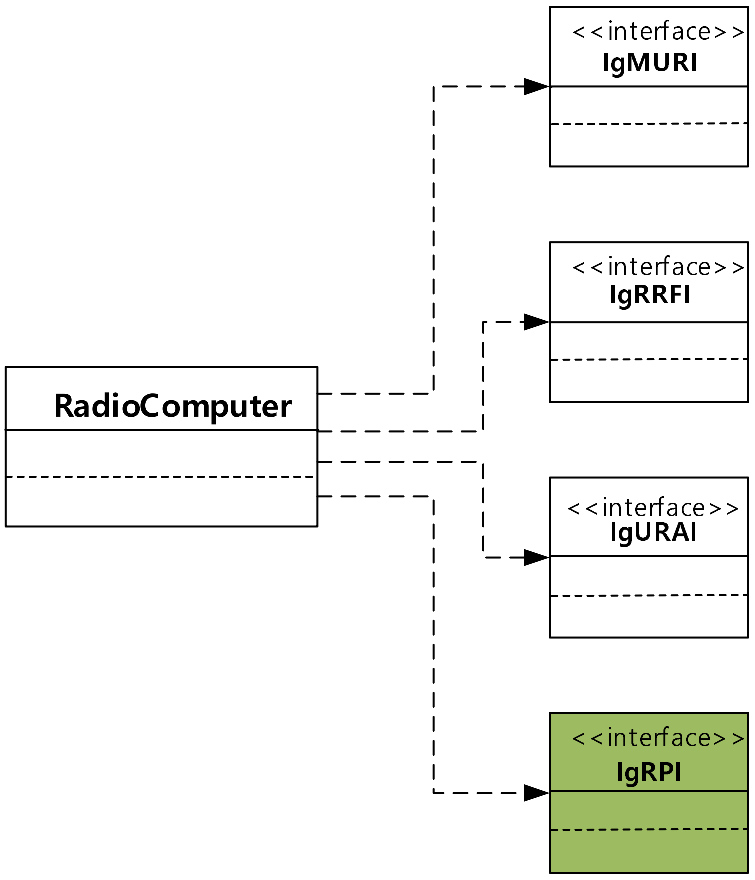


Figure 4.2: UMLclass diagram for Radio Computer interfaces

Figure 4.2 illustrates UML class diagram for Radio Computer interfaces. The reconfigurable RE may be seen as a set of multiple Radio Computers where individual URAs are engineered as software entities in ETSI TS 103 648 [i.2].

The present document is organized as follows:

- Clause 5 describes the system requirement mapping.
- Clause 6 describes the radio virtual machine specification.
- Clause 7 describes the Configcodes for RVM.
- Clause 8 describes the radio library structure.
- Clause 9 describes the loading, linking and initialization.
- Clause 10 describes the compiling for RVM.
- Annex A describes the mapping between Binary and XML.
- Annex B describes SFB Candidates.
- Annex C describes the replacement of selected components of an existing RAT.

While UML is used for defining the information model and protocol related to gRPI, other modelling languages could be used as well.

5 System Requirement Mapping

The Radio Programming Interface and its related components described in the present document shall support the system requirements shown in table 5.1 referring to clause 6 of ETSI TS 103 641 [1]. This is achieved by introducing the entities/components/units given in the 1st column of table 5.1.

Table 5.1: Mapping of Radio Programming Interface and its related components to the system requirements described in ETSI TS 103 641 [1]

Entity/Component/Unit	System Requirements [1]	Comments
Radio Programming Interface	R-FUNC-RER-04	The requirement shall be as described in clause 6.4.4 of [1].
Radio Virtual Machine	R-FUNC-RER-13	The requirement shall be as described in clause 6.4.13 of [1].
	R-FUNC-RER-14	The requirement shall be as described in clause 6.4.14 of [1].
	R-FUNC-RER-15	The requirement shall be as described in clause 6.4.15 of [1].
Radio Library	R-FUNC-FB-06	A library extension shall be supported. The requirement shall be as described in clause 6.3.6 of [1].

6 Radio Virtual Machine specification

6.1 Concept of RVM

As introduced in ETSI TS 103 648 [i.2], the Radio Virtual Machine (RVM) is an Abstract Machine which is capable of executing Configcodes and it is independent of the hardware. The implementation of a RVM is target Radio Computer specific and it shall have access to the Back-end Compiler (on the platform itself or externally as described in ETSI TS 103 648 [i.2], clause 4.4.1) for Just-in-Time (JIT) or Ahead-of-Time (AOT) compilation of Configcodes.

This clause describes the concept of RVM. As mentioned above, the RVM is an abstract machine, which executes a particular algorithm presented as a data flow chart. In other words, the RVM is the result of replacing all operators and tokens in the particular data flow chart with Abstract Processing Elements (APEs) and Data Objects (DOs), respectively. Each APE executes computations marked by the replaced operator identifier. These computations are taken from the Radio Library.

Figure 6.1 illustrates a conceptual view of RVM processing. This process requires APE, DO and Radio Library, of which the definitions are as follows:

- APE abstracts a computational resource corresponding to the operation in a particular data flow chart.
- DO abstracts a memory resource. In other words, DO is an abstracted memory for storing data used during the procedure of Radio processing.
- Reference/Native Radio Library includes normative definitions/native implementation of all Standard Functional Blocks (SFBs) [i.2] for front-end/back-end compilation. Note that the computations included in the Radio Library are represented in terms of normative definitions or native implementations of SFBs depending upon whether the Radio Library is used for front-end or back-end compilation, respectively.

NOTE 1: User Defined Functional Blocks (UDFBs) will be created through combination of SFBs and represented as a data flow chart to be executed in the RVM. Alternatively, a UDFB is implemented as a stand-alone module/function which can be mapped:

i) into one APE (i.e. this UDFB can be considered atomic); or

ii) into an eRVM/RVM (i.e. not atomic). UDFBs are not in general included into the Radio Library, but they are part of the Radio Application Package.

The RVM begins to work immediately after some DOs initialization. All APEs shall execute computations asynchronously and concurrently. An individual APE shall execute the allocated operator if all the corresponding input DOs are full. APEs shall access DOs with operations "read", "read-erase", or "write". After reading input data from DOs, the APE shall execute the allocated operator and, if output DOs are empty, then the APE shall write processed data. Any full output DO shall block the corresponding writing operation. The RVM shall execute computations until reaching the state when all APEs become inactive. In this state, there are not enough full DOs, which can activate the inactive operators. The result of computations are full DOs, which cannot activate the inactive operators.

NOTE 2: An Output DO can become an Input DO for a subsequent operator. Then, this input DO can activate the subsequent operator.

NOTE 3: The state or operation of a given APE is independent on the state of other APEs. I.e. each APE is atomic.

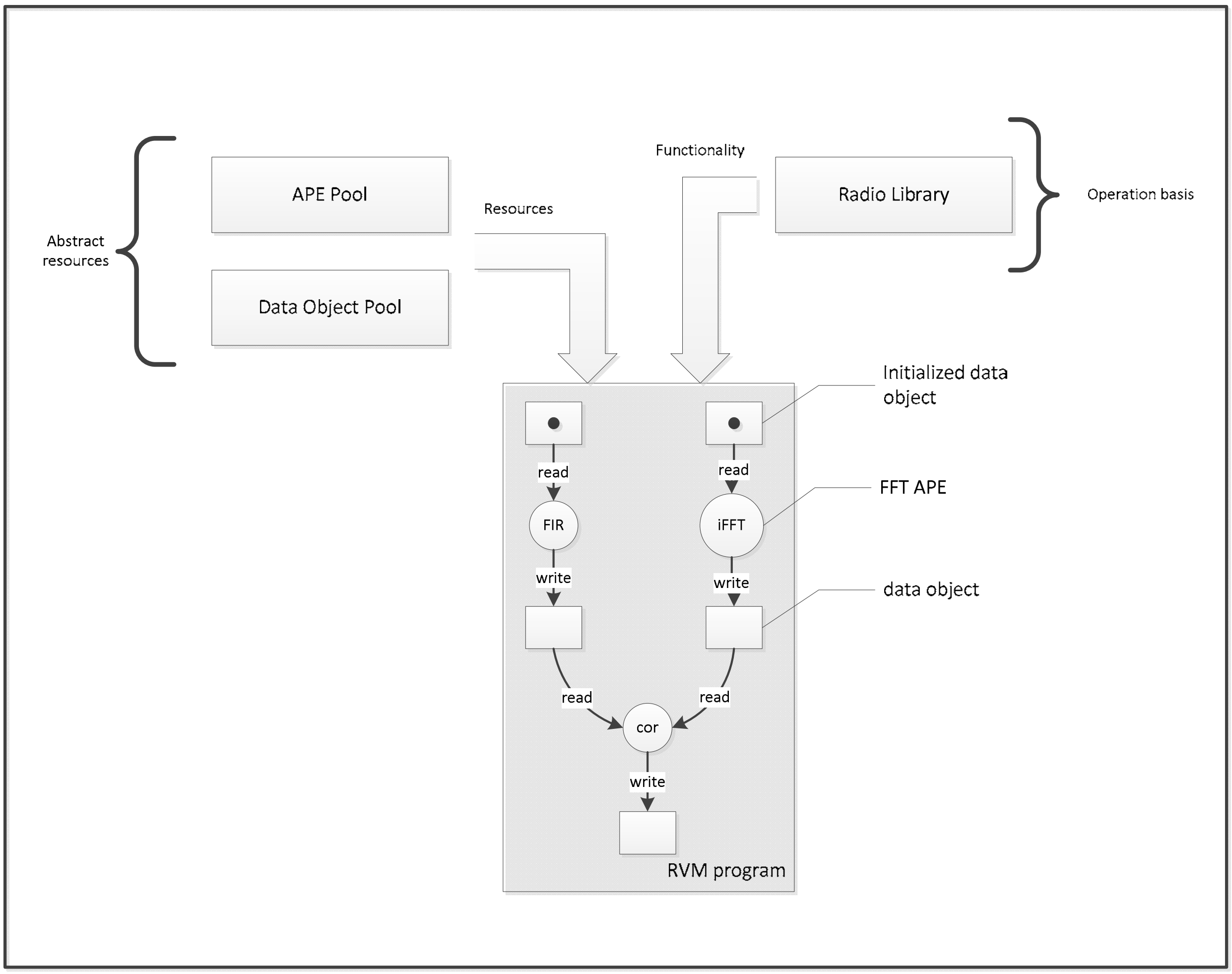


Figure 6.1: Conceptual Diagram of Radio Virtual Machine Processing

6.2 Elementary RVM (eRVM)

This clause describes the eRVM which shall consist of components of Basic Operations, Program memory, Control Unit (CU), Abstract Switch Fabric (ASF) as well as APEs and DOs, of which the definitions are as follows. eRVM shall not contain another eRVM or RVM.

- Basic Operations shall include operators either provided:

  i) from Radio Library as SFBs; and/or

  ii) from UDFB set as UDFBs, each of which is mapped onto one single APE.

NOTE 1: Since UDFBs might be implemented as a stand-alone module/function which can be mapped into one APE, in this case, Basic Operations include operators provided by UDFB Set as well as by Radio Library as SFBs. Note that those UDFBs are atomic.

NOTE 2: For a RVM, the SFB or UDFB can be mapped onto an APE or RVM or eRVM. In the eRVM case, the mapping to RVM or eRVM is not possible since it is the lowest level of hierarchy as it will be introduced in clause 6.3.

NOTE 3: From an execution perspective, there is no difference between SFBs and UDFBs.

- Program memory shall be provided with Configcodes which determine the eRVM configuration.
- CU shall generate Initialization and Set-up instructions for APEs, DOs and ASF based on decoding Configcodes stored in the Program memory.
- ASF shall connect APEs and DOs in accordance with CU signals. One DO can be connected with multiple APEs. One APE can be connected with multiple DOs. DO from other eRVMs can be connected with ASF through external data ports.

Figure 6.2 illustrates a block diagram of eRVM. Basic Operations in eRVM consist of operations provided by the Radio Library and/orUDFB Set.

NOTE 4: A target platform may or may not provide accelerators for some/all SFBs and/or UDFBs.

NOTE 5: Three cases can be considered:

i) RAP includes only SFBs;

ii) RAP includes only UDFBs;

iii) RAP includes SFBs and UDFBs.

NOTE 6: Furthermore, and independent of the upper note, Basic Operations may include:

i) SFBs only;

ii) UDFBs only; or

iii) SFBs and UDFBs.

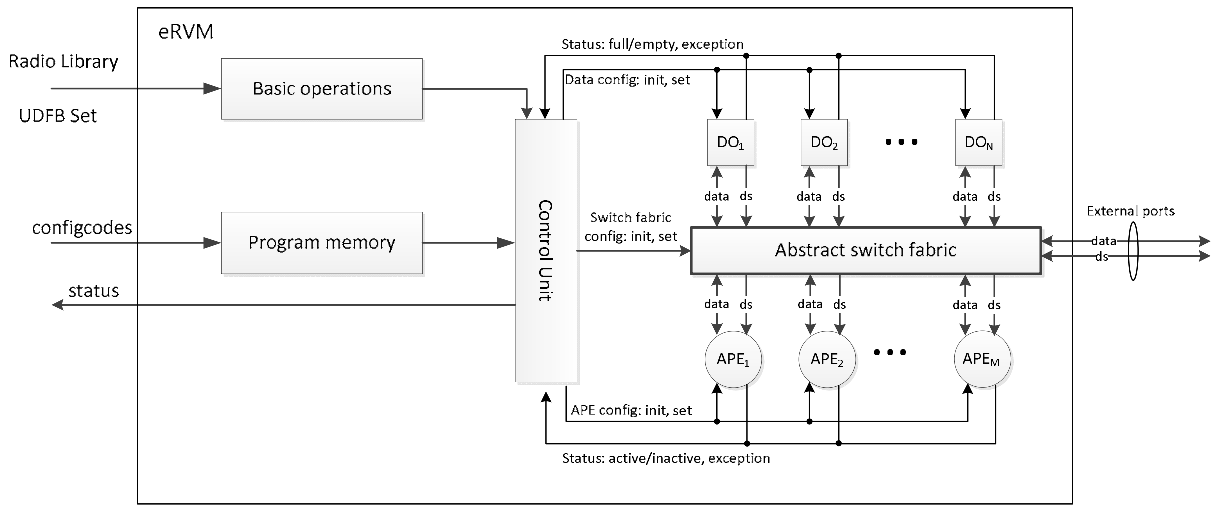


Figure 6.2: Elementary RVM

The Data path of an eRVM shall consist of the following blocks:

- DOs;
- APEs;
- ASF.

Each DO shall have a unique number and for this purpose the DOs shall be represented as DO_1, DO_2, …, DO_N, where N is a sufficiently large number. The structure of DO is shown in figure 6.3.

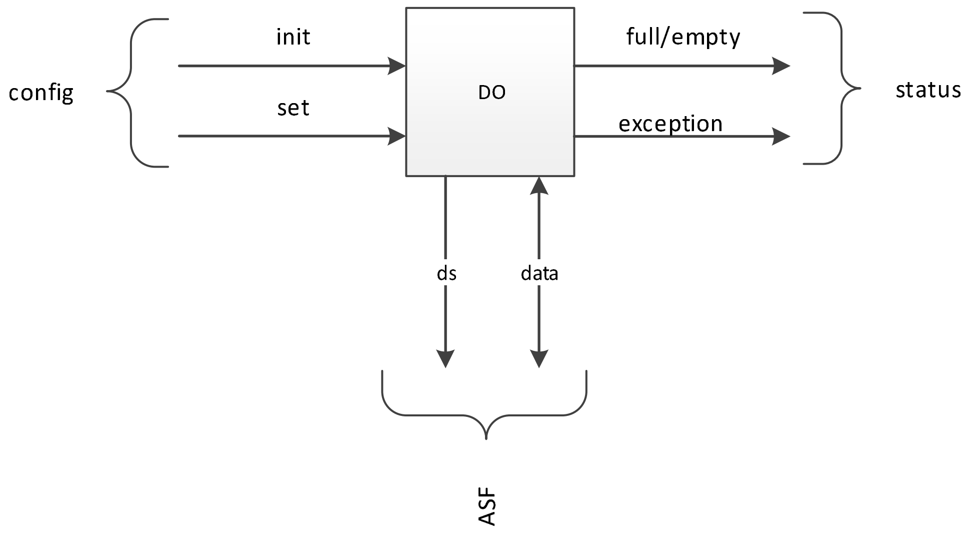


Figure 6.3: DO and its interfaces

Each DO shall be configured by a config instruction which consists of:

- **init** field initializes DO according to the specific initialization procedure (depending on implementation);
- **set** field is an instruction which sets up the DO attributes such as DO_ID, access time, size, etc. (as shown in clause 6.2).

DO shall communicate with APEs through ASF interface which consists of:

- **data status (ds)** signal to indicate whether the DO is full or empty;
- **data lines directed to or from DO** to read or write data to or from APEs.

Status interface shall provide the status information of DO to CU and consists of:

- **full/empty** describes whether DO is full of data or empty;
- **exception** describes the reason of fail when an APE operates with the DO.

Each APE shall have a unique number and shall be represented as APE_1, APE_2, …, APE_M, where M is a sufficiently large number. The structure of APE is shown in figure 6.4.

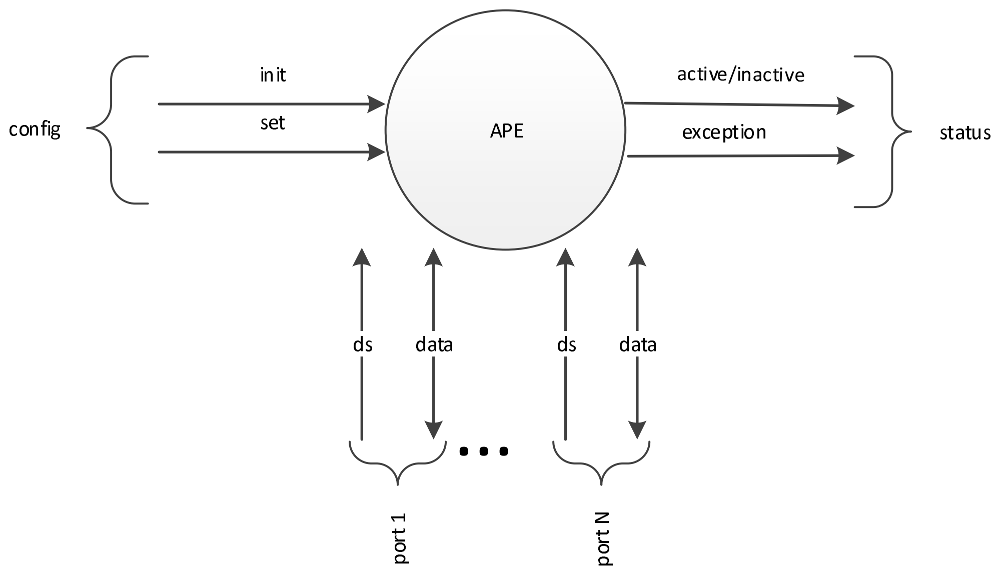


Figure 6.4: APE and its interfaces

APEs shall be configured by the config instruction which consists of:

- **init** field brings the op code operation from Basic Operations;
- **set** field sets up APE attributes such as the number of ports, port types, the execution cost and time.

APE's ports shall connect APE to ASF and shall include data interface which consists of:

- **ds** signal to indicate whether the DO is full or empty;
- **data** lines to read or write data through ASF.

Status interface shall provide status information of APE to CU and consists of:

- **active/inactive** describes state of the APE such as active and inactive;
- **exception** describes the reason of fail when an APE's operation has an error.

NOTE 7: The APE is active when it has consumed input DOs and processes them. The APE goes to the inactive state with corresponding indication to CU immediately after processing all the data associated to the APE.

ASF shall connect APEs and DOs as shown in figure 6.5. One DO can be connected to multiple APEs. One APE can also be connected to multiple DOs.

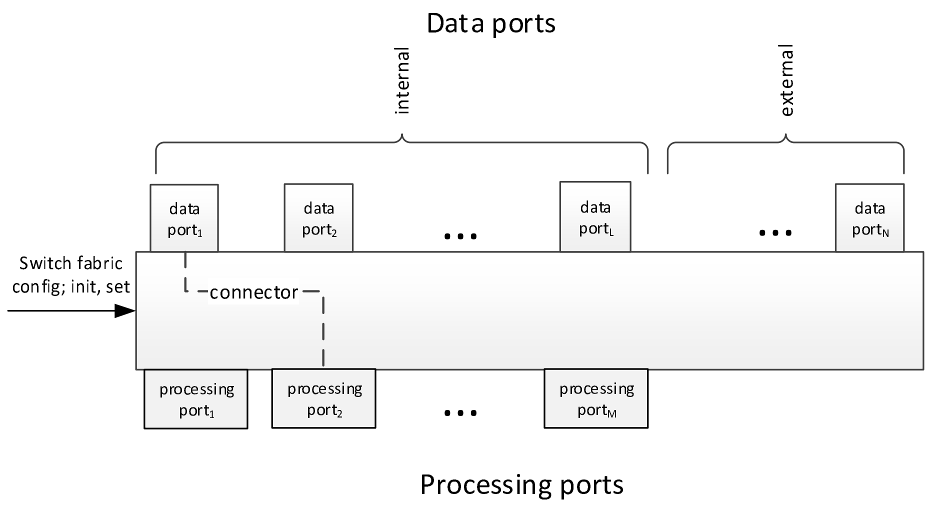


Figure 6.5: Abstract Switch Fabric

ASF shall connect DOs and APEs through ports which consist of:

- **data ports (internal)** connect the ASF to DOs via interface lines;
- **data ports (external)** connect the ASF to DOs from other eRVMs or RVMs;
- **processing ports** connect the ASF to APEs via interface lines.

Each connector of ASF shall connect ports bounded to DO with ports bounded to APE. Each connector shall have the same interface lines as ports do, i.e. ds, data. Connectors shall convey interface values between ports when they appear in corresponding ports.

CU shall configure the ASF by the following commands:

- **init** associates data ports with DOs and processing ports with APEs;
- **set** creates connections between data ports and processing ports.

6.3 RVM Hierarchy

Figure 6.6 illustrates RVM hierarchy which shall consist of APEs, DOs, Basic Operations, Program memory, CU, ASF and eRVMs/RVMs. Note that RVM might contain another eRVM(s)/RVM(s). Each eRVM/RVM in this case functions like an APE. Similarly to the case of figure 6.2, the CU in figure 6.6 connects the DO(s) to the corresponding APE(s) through the ASF. But in this case the data might be connected to eRVM/RVM as well as to APE.

For a RVM, the SFB or UDFB shall be mapped onto an APE or RVM or eRVM.

NOTE 1: In the eRVM case, the mapping to RVM or eRVM is not possible since it is the lowest level of hierarchy.

NOTE 2: Only Basic Operations can be mapped to an APE, but these Basic Operations can furthermore be mapped to an eRVM or RVM. If an SFB or UDFB is not included in the Basic Operations, it cannot be mapped to an APE, it is mapped to an eRVM or RVM.

The RVM shall be scalable vertically and/or horizontally. As for vertical scaling, since each eRVM contains exactly one particular data flow chart, i.e. specific algorithm, in order to build a RVM hierarchy, an APE can contain another eRVM/RVM which executes another particular data flow chart. As for horizontal scaling, several RVMs can be arranged on the same level. These horizontally arranged RVMs shall be independent, meaning that fully independent processes are executed in parallel.

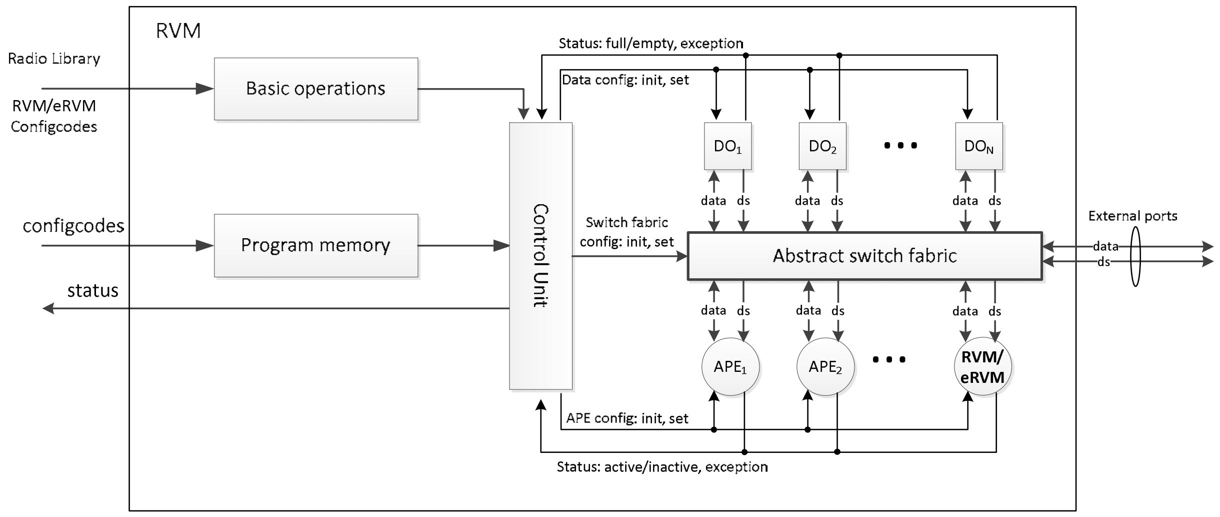


Figure 6.6: RVM hierarchy

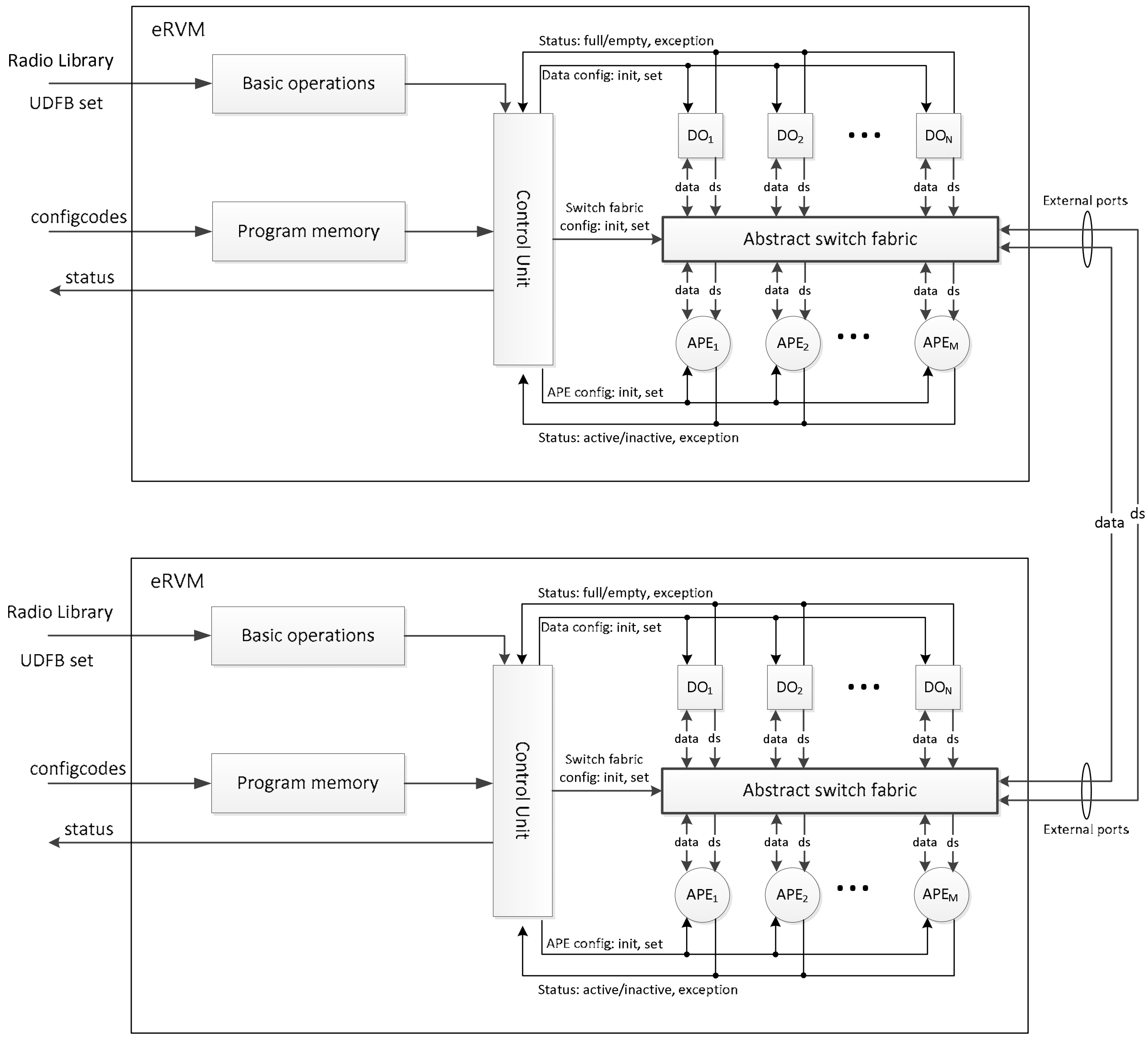


Figure 6.7: Horizontal scaling of eRVM

6.4 Data types

6.4.1 Types and Values

The only data types for DOs shall be tokens which have some size in bits. The DOs structure is recognized by initialized APE.

NOTE: As one example, full DOs can be treated an allocation of bits in memory.

6.4.2 Run-Time Data

There shall be the following Run-Time Data types:

1) CC (Configcodes Counter) register: positive 32-bit integer.

2) Constant DOs: dynamic allocation of bits existing only during task execution and initialized by some external agent. In short, this DO is related to externally provided data.

3) Intermediate DOs: dynamic allocation of bits existing only during task execution and initialized by intermediate value created as a result of corresponding APE operation.

6.5 Arithmetic

Arithmetic is not fixed but defined by operations from Basic Operations.

6.6 Exceptions

There shall be 2 types of exceptions:

- Generated by DOs, see table 6.1.

Table 6.1: DO Exceptions

Value	Semantic
00	No exception
01	Operation with size > DO size
10	Conflicting writes
11	Conflicting read-erase operations

- Generated by APEs/(e)RVMs, see table 6.2.

Table 6.2: APE/(e)RVM Exceptions

Exception	Semantic
00	No exception
01	Change CC flow
10	Arithmetic overflow/underflow
11	Incorrect operation (operation is carried out on data where the operation is not defined)

6.7 Control, Synchronization and Execution

Control shall be data-driven only. Execution shall be done by APE in case that:

- All input DOs are configured and full in case of terminal operations.
- All input DOs are configured and some of them are full in case of nonterminal operations.

- APE is configured.
- APE is connected with configured DOs.

After APE has been configured, some output DOs of APE might be non-empty while the APE is executing its operation. The task of execution is not instantaneous but has some finite duration. All operation on the data path shall be concurrent and asynchronous.

CU does not manage task execution directly. CU shall:

- configure all data path elements before Configcodes execution;
- receive status signals from all data path element during task execution;
- detect end of each Configcode;
- detect the last Configcode in the task;
- stop execution in case of any exception.

6.8 Operations with Memory

In the applied model, memory is considered be flat and infinite. Its implementation is out the scope of the present document. Any RVM shall have access to the memory. DOs shall be allocated in the memory during their configuration. The memory shall allow multiple parallel read write operations. Conflicting operations shall enforce exceptions. Attempts to write more data than there is allocated DO memory shall also enforce exceptions. During particular memory operation the entire DOs shall be consumed.

6.9 RVM run-time environment

The RVM Runtime Environment (RVM RE), e.g. software based, allows to run RAs which might be Configcodes or executable codes:

1) In case of CConfigcode execution, the RVM RE comprises a RVM interpreter.
2) In case of executable codes, binary platform-specific codes are created which are executed on the target platform.

The RVM RE provides access for RAs to platform hardware resources and radio spectrum. For that purpose RVM RE and RAs shall implement interfaces including gMURI, gRRFI, gURAI.

In particular, installation and uninstallation of RAs are supported by gMURI. Access to platform hardware resources is provided by gURAI. Access to radio spectrum is provided by gRRFI.

In case of Configcodes, JIT compiler might be a part of the RVM RE.

Any type of code representation (i.e. executable code, source code, IR) should comply with the interfaces defined in ETSI TS 103 648 [i.2].

NOTE 1: In the case of executable code, the ROS defined in ETSI TS 103 648 [i.2] allows the executable RA code to access the platform HW in order to generate or receive a radio signal.

NOTE 2: In the case of source code, compilation is performed on the platform as an additional step prior to execution.

NOTE 3: The RVM is only used for the IR case.

7 Configcodes for RVM

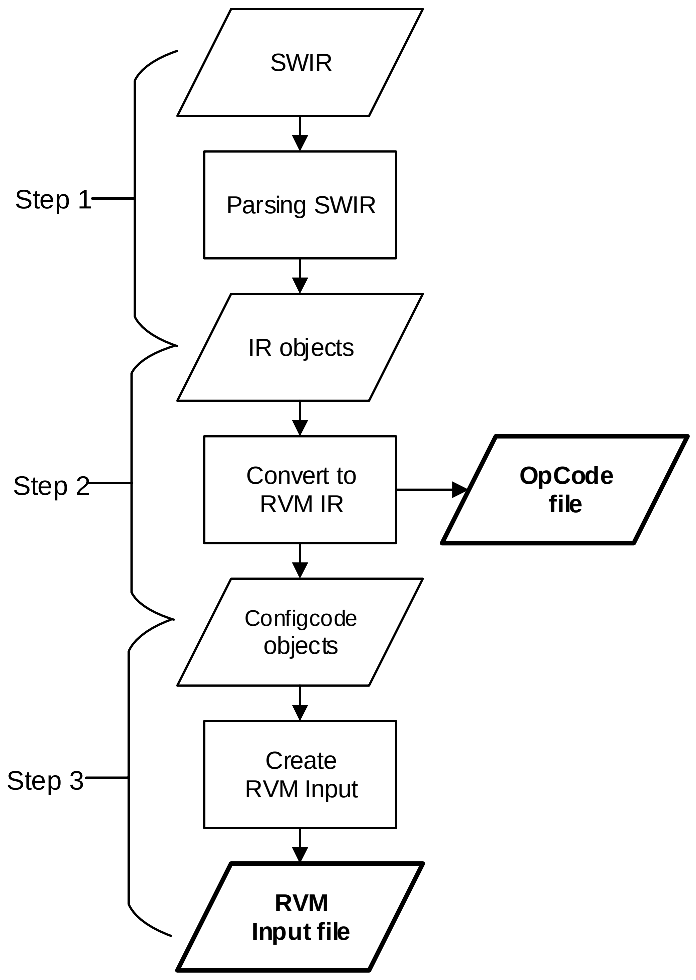
7.1 Introduction

This clause explains how the Configcodes are generated as a result of front-end compilation of the software Intermediate Representation (SWIR) during the design time of RA code distribution. The Configcodes are typically generated in either binary or eXtensible Markup Langugage (XML) depending on vendor's choice.

7.2 Configcodes generation

This clause explains how the Configcodes are generated in either binary or XML format, which is referred to as a RVM Input file in the present document. Figure 7.1 illustrates the processing steps of generating the Configcodes from corresponding SWIR. Processing of Configcodes generation shown in figure 7.1 starts from SWIR which consists of functions and processes of a given RA code. The SWIR shall be generated through the procedure of parallelizing a given RA code created in a high-level language, e.g. C, C++, etc. The operation of SWIR is based on data-driven mechanism as mentioned in clause 6.1. The conversion of SWIR into corresponding Configcodes shall consist of the following 3 steps:

1) Parsing SWIR file.
    - Input: SWIR file.
    - Output: Intermediate Representation (IR) objects which consist of terminal objects and links that connect the terminal objects.
    - Actions: Parsing of SWIR file and creating IR objects consisting of sections and links.
2) Mapping of IR objects into Configcodes objects.
    - Input: List of IR objects.
    - Output: Configcodes objects each of which represents one configuration determined by terminal operators, APE, DO and Switch configurations and text file with OpCode which specifies implemented data flow chart.
    - Actions: Conversion of each object from IR data format to Configcodes format with all configurations of the Configcodes in the task being determined by parameters for APE, DO and Switch. Creation of OpCode file with matches between the name of each implemented function and corresponding OpCode.
3) Creation of RVM Input file.
    - Input: Configcodes objects.
    - Output: RVM Input file.
    - Actions: Creation of RVM Input file from Configcodes objects.



NOTE: While there might be various formats in the RVM Input file, the present document considers XML and Binary format. Examples of Configcodes defined in Binary and XML format are shown in clauses 7.3 and 7.4, respectively.

Figure 7.1: Processing step of generating RVM Input file

7.3 Binary format for Configcodes

In this clause, the format of binary Configcodes is presented.

Each task shall include one or several Configcodes. Each of them shall consist of Control Configcode, DO Configcode, APE Configcode, ASF Configcode and optionally Next Configcode Address Offset (NCAO). The field NCAO shall be augmented if NAF = 1. The binary format of Configcodes is shown in figure 7.2 and shall consist of:

- The control section provides general information regarding the task and consists of:
  - LCF, 1 bit, LCF = 1 means that this is the last Configcode in the task;
  - NAF, 1 bit, NAF = 1 means that the field NCAO is augmented to this Configcode;
  - Task_ID, 8 bits, is an automatically generated identifier of this task;
  - gRPI version, 8 bits, is version number of supported gRPI;
  - Reference_ID, 8 bits, is SFB identifier of the reference Radio Library;
  - Implementation version, 8 bits, is the version number of the implemented task;
  - Developer_ID, 16 bits, is the identifier of the developer who creates the current task;

  - Creation_Date, 16 bits, is the date of task creation.
- DO section provides general information regarding DO configuration and consists of:
  - N_DO, 8 bits is the number of DOs involved in this Configcode;
  - N_DO particular DO configuration fields:
    - DO_config is a particular DO Configcode;
    - ASF_config is the ASF Configcode without DO field.
- APE section provides general information regarding APE configuration and consists of:
  - N_APE, 16 bits is the number of APEs involved in this Configcode;
  - N_APE particular APE configuration fields:
    - APE config field is a particular APE Configcode.

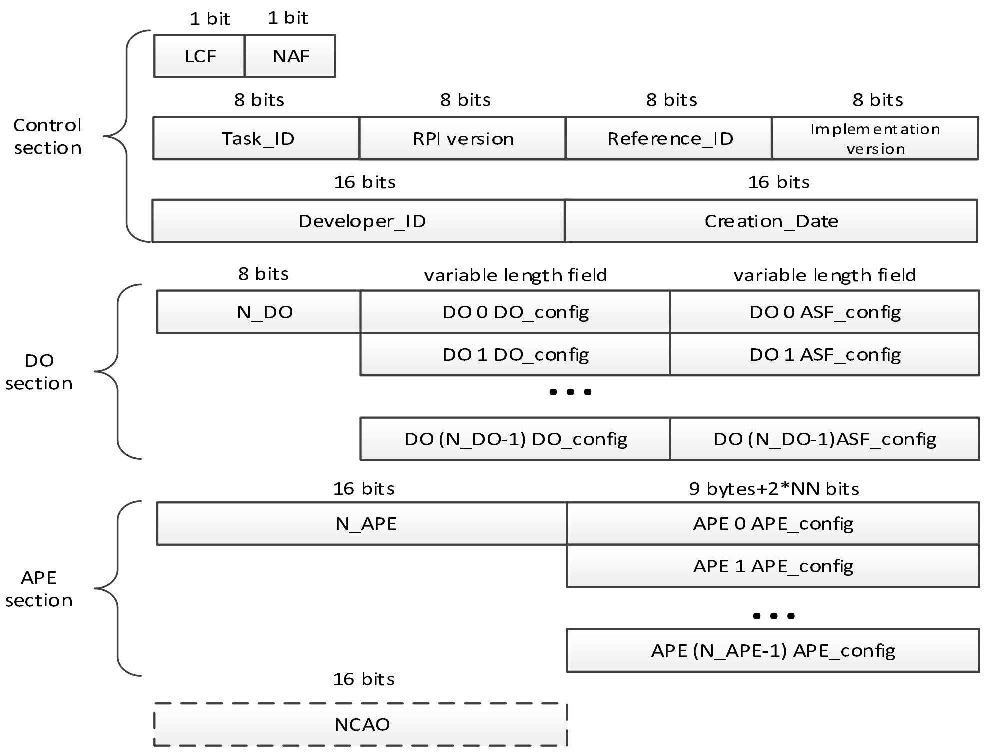


Figure 7.2: Binary format for Configcodes

Format of a particular **DO_Configcode** shall consist of two parts: one part is a fixed length field of DO set code and the other part is a variable length field of DO init code. It is shown in figure 7.3.

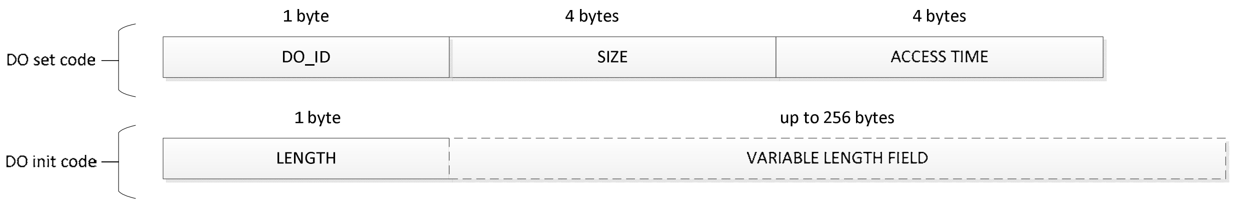


Figure 7.3: Format of a particular DO_Configcode

DO shall be configured by an instruction ***DO_config*** (***Set, Init***) with Set field and Init field.

Figure 7.4 illustrates the format of the **DO set** code and it shall consist of:

- SIZE is the positive integer number showing the DO size in bytes;
- ACCESS TIME is the positive integer number showing access time in ns.

4 bytes	4 bytes
SIZE	ACCESS TIME

Figure 7.4: Format of DO set code

The **Set** field shall set DO attributes using ***Set***(YYYY, ZZZZ) where YYYY is the size in bytes, ZZZZ is the access time in ns.

Figure 7.5 illustrates the format of **DO init** code and it consists of:

- LENGTH, 1 byte, is the length of the variable part of the field in bytes;
- VARIABLE LENGTH FIELD up to 256 bytes brings immediate values.

1 byte	up to 256 bytes
LENGTH	VARIABLE LENGTH FIELD

Figure 7.5: Format of DO init code

The **Init** field shall initialize the DO according to the specific initialization procedure (depending on implementation) and make DO full after initialization if LENGTH ≠ 0, i.e. ***Init***(XXXX), where XXXX contains the length and initialization value in the form of a bit sequence. Init field might be empty if LENGTH = 0, in such case the DO is empty.

Figure 7.6 illustrates the format of **APE_Configcode** and it shall consist of:

- APE_ID, 2 bytes, is the number of APE;
- op code, 20 bits is the code of operation from Basic Operations;
- T, 1 bit flag, when T = 1 for dynamic operations, APE is inactive just after completion of the operation, when T = 0 for static operations, APE is active even after completion of the operation;

NOTE 1: The statement "APE is inactive just after completion of the operation" indicates that another functionality may be allocated to the APE. The statement "APE is active even after completion of the operation" indicates that no change of functionality will occur.

- NN, 3 bits is the number of ports;
- cost, 2 bytes is the execution cost value;
- time, 2 bytes is the time constraint value;
- port access type, 2 bits per port describes the access type {r, re, w, rew}.

2 bytes	20 bits	1 bit	3 bits	2 bytes	2 bytes	2*NN bits
APE_ID	op code	T	NN	cost	time	access type

Figure 7.6: Format of a particular APE_Configcode

APE shall be configured by an instruction ***APE_config*** (***APE_Port_ID***, ***Set, Init***) with APE port identifier field, Set field and Init field.

The **APE_Port_ID** field shall have identifier ***APE_Port_ID*** (XXXX.YY, PORT_TYPE), where XXXX = APE_ID, YY is the number of APE ports and PORT_TYPE = {r, re, w, rew}.

The **Set** field shall set APE attributes using ***Set***(NN, PP, XXXX, YYYY) where NN is the number of ports (decimal), PP is the field defining port types, XXXX is the execution cost (decimal), if XXXX = 0 then there is no cost, YYYY is time in ms (decimal), if YYYY = 0 then there are no time constraints.

The **Init** field shall bring the identifier of an operation from the set of Basic Operations and port configuration. During the APE configuration procedure, the APE shall be bound with **op** operation from the Basic Operations using ***Init***(op). Input and output parameters of the operator shall be bound to the APE's ports of corresponding access types.

NOTE 2: The op code is obtained by applying some unique mapping function to the SFB identifiers. The choice of the mapping function needs to be defined. However, it is beyond the scope of the present document.

NOTE 3: APE Configuration comprises two stages, i.e. initialization and set-up. The Configcodes provide the parameters for both procedures.

NOTE 4: Access type are defined as {r, re, w, rew}.

After APE configuration procedure is successfully completed, the APE shall:

- be moved to the active state with corresponding indication to CU;
- process values in lines ds (data status) and data.

APE exception codes shall be used as shown in table 7.1.

Table 7.1: APE Exceptions

Exception	Semantic
00	No exception
01	Change CC (Configcodes Count) flow
10	Arithmetic overflow/underflow
11	Incorrect operation

Figure 7.7 illustrates the format of the **ASF_Configcode**. Ports shall connect ASF with DOs or with APEs as described in figure 6.5. Connectors of ASF shall connect ports bounded to DO with ports bounded to APE. Each connector shall have the same interface lines as ports: ds, data. Connectors shall convey interface values between ports when they appear in the corresponding ports.

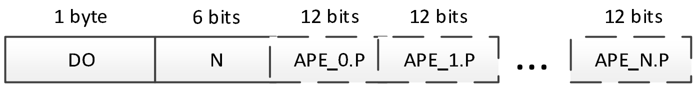


Figure 7.7: Format of ASF_Configcode

Figure 7.7 illustrates the format of **ASF_Configcode** and it shall consist of:

- DO, 1 byte field shall provide the DO ID number;
- N, 6 bits shall be the number of APEs connected with DO;
- APE_K.P, 12 bits where K runs from 0 to N-1:
  - APE_K, 8-bit shall denote the APE number;
  - P, 4-bit shall denote the port number.

ASF shall be configured by an instruction ***ASF_config*** (***Set, Init***) with Set field and Init field.

The **Set** field shall set the ASF attributes using ***Set***(DO, N) where DO is the ID number of DO, N is the number of APEs connected with the DO.

The **Init** field shall initialize the ASF using ***Init***(APE_K.P) where APE_K, with K running from 0 to N-1, denotes the APE number, P denotes the port number. After configuration connectors shall convey interface values to DOs and APEs.

7.4 XML schema for Configcodes

In this clause, the XML schema for Configcodes is presented. Each task includes one or several Configcodes.

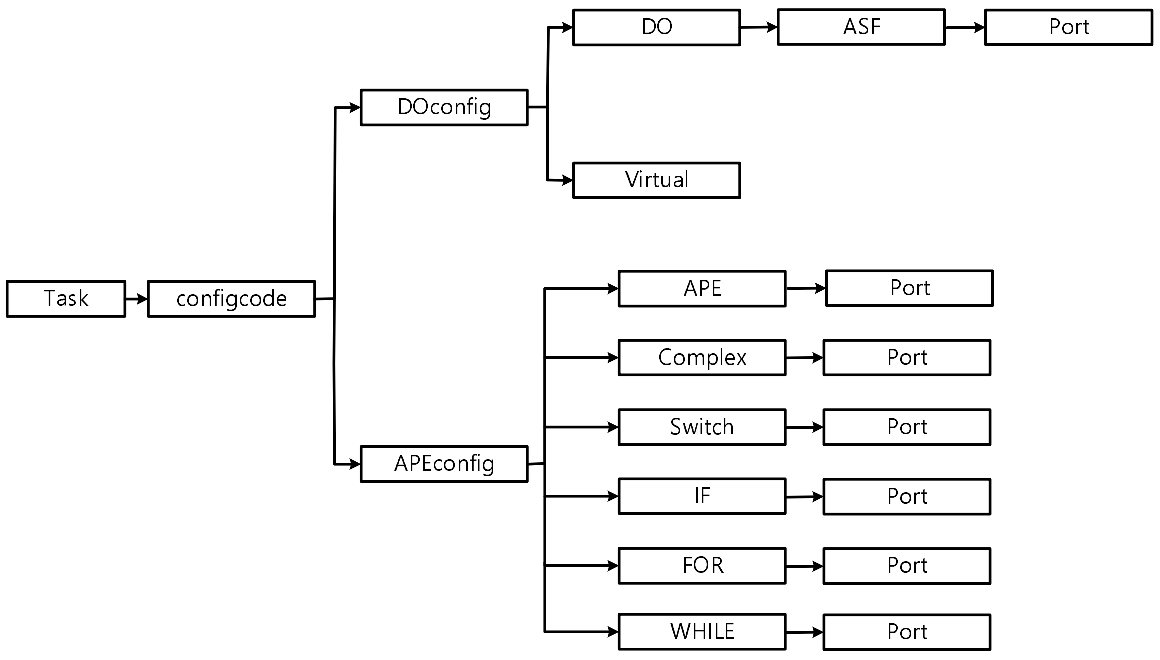


Figure 7.8: XML elements consisting of Configcodes

Figure 7.8 illustrates the XML elements consisting of Configcodes. The following XML elements shall be used:

- task;
- Configcode;
- DOconfig;
- DO;
- ASF;
- Virtual;
- APEconfig;
- APE;
- Complex;
- FOR;
- SWITCH;
- IF;
- WHILE;
- Port.

Attributes of each element shall be used as given by tables 7.2 to 7.15. It is illustrated in figure 7.8.

Element /task

- Parent element: document root

- Children:
    - <Configcode>

<task> is the root element of the *xml*.

Table 7.2: Attributes of <task>

Attribute	Description
TaskID	Task identifier.
gRPIVersion	Version of gRPI.
DeveloperID	Developer identifier.
CreationDate	Date of task creation.
ReferenceID	SFB identifier of reference Radio Library.
TaskImpVersion	Task implementation version.
num	Contain number of configurations in the task.

Element /task/Configcode

- Parent: <task>
- Children:
    - <DOconfig>
    - <APEconfig>

<Configcode> is tag for one configuration in the task.

Table 7.3: Attributes of <Configcode>

Attribute	Description
LCF	Last Config Flag. Identify last Configcode in the task.
ID	Automatically generated identifier.
NDO	Number of DOs in the configuration.
NAPE	Number of APE in the configuration.

Element /task/Configcode/DOconfig

- Parent: <Configcode>
- Children:
    - <DO>
    - <Virtual/>

<DOconfig> is tag for DOs configurations.

Element /task/Configcode/DOconfig/DO

- Parent: <DOconfig>
- Children:
    - <ASF>

<DO> is tag for configuration of one DO.

Table 7.4: Attributes of <DO>

Attribute	Description
ID	Automatically generated identifier.
Size	Size of DO.
AccT	Accesses time of DO.
Length	Length of initial data in object.
Data	Initial data.

Element /task/Configcode/DOconfig/DO/ASF

- Parent: <DO>
- Children:
  - <Port/>

<ASF> is tag for configuration of ASF for DO.

Table 7.5: Attributes of <ASF>

Attribute	Description
APENum	Number of APE connected to DO.

Element /task/Configcode/DOconfig/ASF/Port

- Parent: <ASF>
- Children: none

<Port> is tag for Switch port.

Table 7.6: Attributes of <Port>

Attribute	Description
APEid	APE identifier.
PortId	Port identifier.

Element /task/Configcode/DOconfig/Virtual

- Parent: <DOconfig>
- Children: none

<Virtual> is tag for virtual DO.

Table 7.7: Attributes of <Virtual>

Attribute	Description
ID	Automatically generated identifier.
Type	Size of DO.
PortRef	Port of the main control operator, to which VDO is connected.
InPortRef	Port of the main control operator, which will be the input port for the VDO.
OutPortRef	Port of the main control operator, which will be the output port for the VDO.

Element /task/Configcode/APEconfig

- Parent: <Configcode>

- Children:
  - <APE>
  - <Complex>
  - <Switch>
  - <IF>
  - <FOR>
  - <WHILE>

<APEconfig > is tag for abstract physical elements configurations.

Element /task/Configcode/APEconfig/APE

- Parent: <APEconfig>
- Children:
  - <Port>

<APE> is tag for configuration of one functional APE.

Table 7.8: Attributes of <APE>

Attribute	Description
ID	Automatically generated identifier.
OpCode	OpCode of implemented function.
T	Flag for dynamic operations.
Cost	Execution cost value.
Time	Time constrain value.
PortNum	Number of Switch ports for APE.

Element /task/Configcode/APEconfig/Complex

- Parent: <APEconfig>
- Children:
  - <Port>

<Complex> is tag for configuration of complex APE.

Table 7.9: Attributes of <Complex>

Attribute	Description
ID	Automatically generated identifier.
Body	ID of body configuration.
T	Flag for dynamic operations.
Cost	Execution cost value.
Time	Time constrain value.
PortNum	Number of Switch ports for APE.

Element /task/Configcode/APEconfig/FOR

- Parent: <APEconfig>
- Children:
  - <Port>

<FOR> is tag for configuration of FOR-cycle.

Table 7.10: Attributes of <FOR>

Attribute	Description
ID	Automatically generated identifier.
Body	ID of cycle body configuration.
T	Flag for dynamic operations.
Cost	Execution cost value.
Time	Time constrain value.
PortNum	Number of Switch ports for APE.

Element /task/Configcode/APEconfig/WHILE

- Parent: <APEconfig>
- Children:
  - <Port>

<WHILE> is tag for configuration of While-cycle.

Table 7.11: Attributes of <WHILE>

Attribute	Description
ID	Automatically generated identifier.
Body	ID of cycle body configuration.
T	Flag for dynamic operations.
Cost	Execution cost value.
Time	Time constrain value.
PortNum	Number of Switch ports for APE.

Element /task/Configcode/APEconfig/IF

- Parent: <APEconfig>
- Children:
  - <Port>

<IF> is tag for configuration of IF operator.

Table 7.12: Attributes of <IF>

Attribute	Description
ID	Automatically generated identifier.
Then	ID of body-then configuration.
Else	ID of body-else configuration.
T	Flag for dynamic operations.
Cost	Execution cost value.
Time	Time constrain value.
PortNum	Number of Switch ports for APE.

Element /task/Configcode/APEconfig/Switch

- Parent: <APEconfig>
- Children:
  - <Port>
  - <Case>

<Switch> is tag for configuration of Switch-Case operator.

Table 7.13: Attributes of <Switch>

Attribute	Description
ID	Automatically generated identifier.
T	Flag for dynamic operations.
Cost	Execution cost value.
Time	Time constrain value.
PortNum	Number of Switch ports for APE.

Element /task/Configcode/APEconfig/APE|Complex|For|WHILE|IF|Switch/Port

- Parent: <APE|Complex|FOR|WHILE|IF|Switch>
- Children: none

<Port> is tag for ASF port.

Table 7.14: Attributes of <Port>

Attribute	Description
ID	Port identifier.
AccType	Accesses type.

Element /task/Configcode/APEconfig/Switch/Case

- Parent: <Switch>
- Children: none

<Port> is tag for case part of Switch operator.

Table 7.15: Attributes of <Port>

Attribute	Description
Body	ID of case-body configuration.
Condition	Case condition.

8 Radio Library

8.1 Introduction

The Radio Library shall include the entire set of SFBs [i.2]. The Radio Library is categorized into two kinds as follows:

- Reference Radio Library shall provide normative definition of each SFB.
- Native Radio Library shall provide platform-specific description of each SFB which represents the target platform hardware.

NOTE 1: The computations included in the Radio Library are represented in terms of normative definitions or native implementations of SFBs depending upon whether the Radio Library is used for front-end or back-end compilation, respectively.

NOTE 2: The examples in the present document illustrate the use of the Radio Library for physical layer functions, but the library is extensible to provide functions of higher layers, e.g. encryption, channelization, radio link selection, etc.

Reference Radio Library and Native Radio Library are described in detail in clauses 8.2 and 8.3, respectively.

8.2 Reference Radio Library

The Reference Radio Library shall include the information regarding each operator downloaded from the Basic Operations of RVM introduced in clause 6 in the case of front-end compilation. The operator corresponds to each SFB defined in ETSI TS 103 648 [i.2].

The Reference Radio Library shall include normative definition of each SFB of the entire set of SFBs [i.2].

NOTE 1: The normative definition should include information for the 3rd party to use while making RA code, such as, e.g. resources required at each function of a given SFB, etc.

NOTE 2: Each SFB corresponds to an operator used in Radio Computing, e.g. FFT, IFFT, Convolution, etc. A normative description of each SFB might be given in a high-level language, e.g. C, C++, C#, etc. for the 3rd party RA provider to refer to.

NOTE 3: The most primitive leveled SFB is denoted as elementary SFB (eSFB). It means that eSFBs are terminal library elements such that they are not decomposed into more fine grained elements. Table 8.1 shows operators that may be defined as eSFBs.

Table 8.1: List of eSFB

Name	Input	Output	Description
NOP	1	1	No operation, coping input to output.
Copy operator	1	2	Coping 1 input to 2 outputs.
Filter	2	1	1 predicate input and 1 data input, if predicate input is true than coping data input to output.
Multiplexer	2	1	Any input triggers coping this input to output.
Demultiplexer	2	2	If predicate is true than copy input to the 1st output else to the 2nd.
Return	1	1	Stop configuration execution with corresponding code sent to output.
Add	2	1	Returns the sum of two inputs.
Subtract	2	1	Subtract the second input from the first input.
Multiply	2	1	Returns the multiple of two inputs.
Divide	2	1	Devides one input value by the other input.
Logical AND	2	1	Returns ture if both inputs can be converted to ture; otherwise, returns false.
Logical OR	2	1	Returns ture if either input can be converted to ture; if both can be converted to false, returns false.
Logical NOT	1	1	Returns false if input can be converted to ture; otherwise, returns ture.
Bitwise AND	2	1	Returns a one in each bit position for which the corresponding bits of both inputs are ones.
Bitwise OR	2	1	Returns a one in each bit position for which the corresponding bits of either or both inputs are ones.
Bitwise XOR	2	1	Returns a one in each bit position for which the corresponding bits of either but not both inputs are ones.
Bitwise NOT	1	1	Inverts the bits of its input.
Left shift	1	1	Shifts a in binary representation b (< 32) bits to the left, shifting in zeroes from the right.
Right shift	1	1	Shifts a in binary representation b (< 32) bits to the right, discarding bits shifted off.
Preincrement	1	1	Increment input by 1, then use the new value of input in the current operation.
Postincrements	1	1	Use the current value of input in the current operation, then increment input by 1.
Predecrement	1	1	Decrement input by 1, then use the new value of input in the current operation.
Postdecrements	1	1	Use the current value of input in the current operation, then decrement input by 1.

Typical candidates for SFBs are given in annex B.

8.3 Native Radio Library

The Native Radio Library shall include information on how each SFB in the Reference Radio Library is implemented in a target hardware platform. The information shall explicitly specify the implementation of each SFB in the Reference Radio Library using dedicated hardware accelerators and/or programmable devices provided in the target hardware platform. The native Radio Library shall be used for the back-end compilation of the Configcodes.

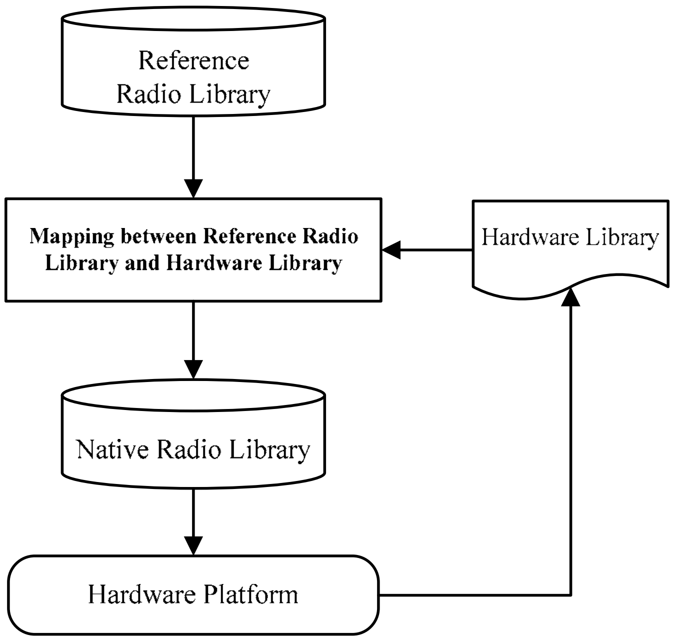


Figure 8.1: Generation of Native Radio Library

Figure 8.1 shows a flow chart for generating Native Radio Library from the corresponding Reference Radio Library. It is obtained from the available Reference Radio Library and the Hardware Library from which a target hardware platform is built. Elements of the Native Radio Library are created as the result of mapping Reference Radio Library elements onto the Hardware Library elements.

NOTE: A Reference Radio Library is preferably publicly available.

9 Loading, Linking and Initialization

RVM shall be instantiated for execution of platform-specific Configcodes obtained from the back-end compiler. The instantiation consists of three steps (following back-end compilation creating platform-specific ConfigCodes for RVM (note: in a different set-up, back-end compilation may create binary code, then no RVM processing is required, but only a run-time environment)):

- Loading.
- Linking.
- Initialization.

Loading is the process of finding the binary representation of particular Basic Operations which configure APEs. The binary files corresponding to the set of identified Basic Operations shall be downloaded into RVM from Native Radio Library.

NOTE: In this stage, the corresponding RVM is optimized for a specific target platform. The Basic Operations are specific to the target platform.

Linking is the process of taking Configcodes for a particular task and combining the corresponding Basic Operations included in the Configcodes to the run-time executable image. Linking process can be static or dynamic depending on the T flag value in *APE_config* as shown in figure 7.6. If T = 1 then the corresponding operation is linked dynamically during run-time execution. In the opposite case, when T = 0, all the Basic Operations included in the Configcodes are linked before run-time execution.

Initialization consists of the instantiation and initialization procedures for DOs, APEs and ASF. Information for this process is taken from corresponding Configcode Init fields (see clause 7). During this process all full DOs are initialized with immediate values according to implementation-dependent procedures (i.e. the way of storing the initial value and how to transfer them to DOs in the initialization phase typically depends on the target platform). Each APE is initialized with the binary code of the corresponding operations. Connections between APEs and DOs are established in the ASF according to the implementation-dependent initialization procedure.

After instantiation, an executable version of the original RA is available that is optimized for the specific target platform. The process is fully finalized and the RA can be executed.

10 Compiling for RVM (Front-End Compilation)

This paragraph describes the front-end part of the compilation process related to the transformation of input source code into platform independent RVM Configcodes. There are two alternatives to obtain Configcodes:

1) RadioApp code (e.g. in C, C++, Java, etc.) is used as an input. Following parallelization (SWIR is obtained as output of the parallelization operation) and RVM compilation, the Configcodes are obtained.

2) RadioApp represented as Data Flow Chart (corresponding to SWIR) is used as an input. Following RVM compilation, the Configcodes are obtained.

The generic compilation flow is shown in figure 10.1. The compilation procedure of SWIR to Configcodes is described in clause 7.2. The RVM Compiler uses also the Radio Lib to produce Configcodes.

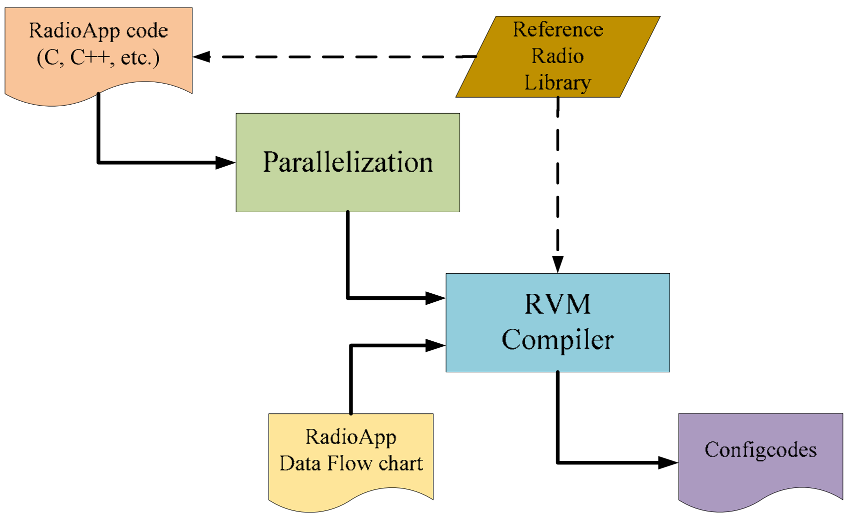


Figure 10.1: Compilation for RVM

NOTE: In the context of the present document, it is assumed that there is no mathematical representation of the compilation process. Consequently, there is no formal validation that the compilation output is a true representation of the original Source Code. A manufacturer will typically use its own (proprietary) model for validation.

Annex A (informative): Mapping between XML and Binary

Table A.1 shows the mapping between XML and binary.

Table A.1: Mapping of XML schema to binary format

XML schema	Binary format	Description
TaskID	Task_ID	Task identifier.
gRPIVersion	gRPI version	Version of gRPI.
DeveloperID	Developer_ID	Developer identifier.
CreationDate	Creation_Date	Date of task creation.
ReferenceID	Reference_ID	SFB identifier of reference Radio Library.
TaskImpVersion	Implementation version	Task implementation version.
num	NAF	Contain number of configurations in the task.
LCF	LCF	Last Config Flag. Identify last Configcode in the task.
NDO	N_DO	Number of DOs in the configuration.
NAPE	N_APE	Number of APE in the configuration.
ID(DO)	DO_ID	Automatically generated identifier.
Size	SIZE	Size of DO.
AccT	ACCESS TIME	Accesses time of DO.
Length	LENGTH	Length of initial data in object.
Data	VARIABLE LENGTH FIELD	Initial data.
APENum	N	Number of APE connected to DO.
APEid	APE_ID	APE identifier.
PortId	APE_Port_ID	Port identifier.
Type	Not applicable	Size of DO.
PortRef	Not applicable	Port of the main control operator, to which Virtual DO is connected.
InPortRef	Not applicable	Port of the main control operator, which will be the input port for the Virtual DO.
OutPortRef	Not applicable	Port of the main control operator, which will be the output port for the Virtual DO.
Body	Not applicable	ID of body configuration.
Then	Not applicable	ID of body-then configuration.
Else	Not applicable	ID of body-else configuration.
Condition	Not applicable	Case condition.
ID(APE)	APE_ID	Automatically generated identifier.
OpCode	op code	OpCode of implemented function.
T	T	Flag for dynamic operations.
Cost	cost	Execution cost value.
Time	time	Time constrain value.
PortNum	NN	Number of Switch ports for APE.
AccType	port access type	Accesses type.

Annex B (informative): SFB Candidate

Annex B describes some typical candidates of SFBs. Each SFB consist of eSFBs. Table B.1 shows the definition of data types. The generally applied definition of each SFB candidate are shown in tables B.2 to B.5.

Table B.1: Definition of Data types

Classification	Data type	Range	Byte(bit)
integer	char	-128 ~ 127	1(8)
	unsigned char	0 ~ 255	1(8)
	short	-32 768 ~ 32 767	2(16)
	int	-2 147 483 648 ~ 2 147 483 647	4(32)
	long	-2 147 483 648 ~ 2 147 483 647	4(32)
	unsigned short	0 ~ 65 535	2(16)
	unsigned int	0 ~ 4 294 967 295	4(32)
	unsigned long	0 ~ 4 294 967 295	4(32)
real-floating type	float	8,4 × 10 ^ -37 ~ 3,4 × 10 ^ 39	4(32)
	double	2,2 × 10 ^ -309 ~ 1,8 × 10 ^ 309	8(64)

Table B.2: Definition of eSFBs related to arithmetic operations

Name		Syntax	Number of operands	Description
Basic assignment		a = b	2	Allocate b to a.
Addition		a + b	2	Add a to b.
Subtraction		a - b	2	Subtract b from a.
Multiplication		a × b	2	Multiply a with b.
Division		a / b	2	Divide a by b.
Modulo (integer remainder)		a % b	2	Modulus Operator and remainder of after a division.
Increment	Prefix	++a	1	Increment the value of the a and return a reference to the result.
	Postfix	a++	1	Create a copy of a, increment the value of the a and return the copy to the result.
Decrement	Prefix	--a	1	decrement the value of the a and return a reference to the result.
	Postfix	a--	1	Create a copy of a, decrement the value of the a and return the copy to the result.

Table B.3: Definition of eSFBs related to comparison operations

Name	Syntax	Number of operands	Description
Equal to	a == b	2	Checks if the values of a and b are equal or not. If yes, then the condition becomes true.
Not equal to	a != b	2	Checks if the values of a and b are equal or not. If the values are not equal, then the condition becomes true.
Greater than	a > b	2	Checks if the value of a is greater than the value of b. If yes, then the condition becomes true.
Less than	a < b	2	Checks if the value of a is less than the value of b. If yes, then the condition becomes true.
Greater than or equal to	a ≥ b	2	Checks if the value of a is greater than or equal to the value of b. If yes, then the condition becomes true.
Less than or equal to	a ≤ b	2	Checks if the value of a is less than or equal to the value of b. If yes, then the condition becomes true.

Table B.4: Definition of eSFBs related to logical operations

Name	Syntax	Number of operands	Description
Logical negation (NOT)	!a	1	It is used to reverse the logical state of its operand.
Logical AND	a && b	2	If both the operands are non-zero, then the condition becomes true.
Logical OR	a || b	2	If any of the two operands is non-zero, then the condition becomes true.

Table B.5: Definition of eSFBs related to bitwise operations

Name	Syntax	Number of operands	Description
Bitwise NOT	~a	1	Switching state 0 to 1, and vice versa.
Bitwise AND	a & b	2	Binary AND Operator copies a bit to the result if it exists in both operands.
Bitwise OR	a | b	2	Binary OR Operator copies a bit if it exists in either operand.
Bitwise XOR	a ^ b	2	Binary XOR Operator copies the bit if it is set in one operand but not both.
Bitwise left shift	a << b	2	The a value is moved left by the number of bits specified by the b.
Bitwise right shift	a >> b	2	The a value is moved right by the number of bits specified by the b.

Annex C (informative): Replacement of selected components of an existing RAT

As mentioned in clause 4, a reconfigurable RE is capable of running multiple radios simultaneously and of changing the set of radios by loading new Radio Application Package (RAP). While a RAP may provide an entire novel RAT to a target radio equipment, it is also possible that a RAP is replacing one or several components of an existing (hardwired) RAT implementation.

For such a selected replacement of existing (hardwired) RAT components, the Radio Library will provide an SFB which provides interfaces to (3rd party) software developers for accessing the (hardwired) RAT internal interfaces. The basic principle is illustrated below.

Figure C.1 presents an example representation of a transmission chain which is comprised of components A, B, C, D, E.

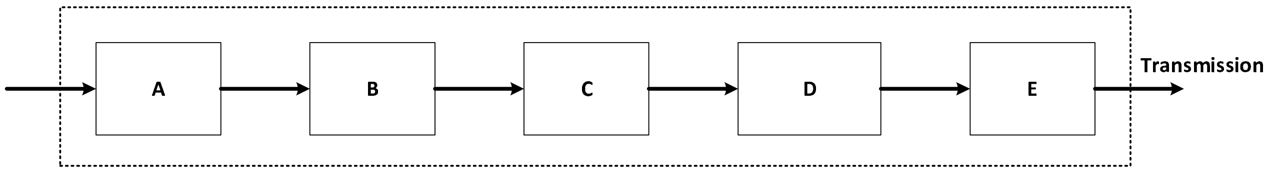


Figure C.1: Example representation of components of a transmission chain

It is assumed that a manufacturer may choose to enable (3rd party) software developers to replace one or multiple of these components by novel software components. Figure C.2 illustrates the replacement of component B through a novel implementation by (3rd party) software developers.

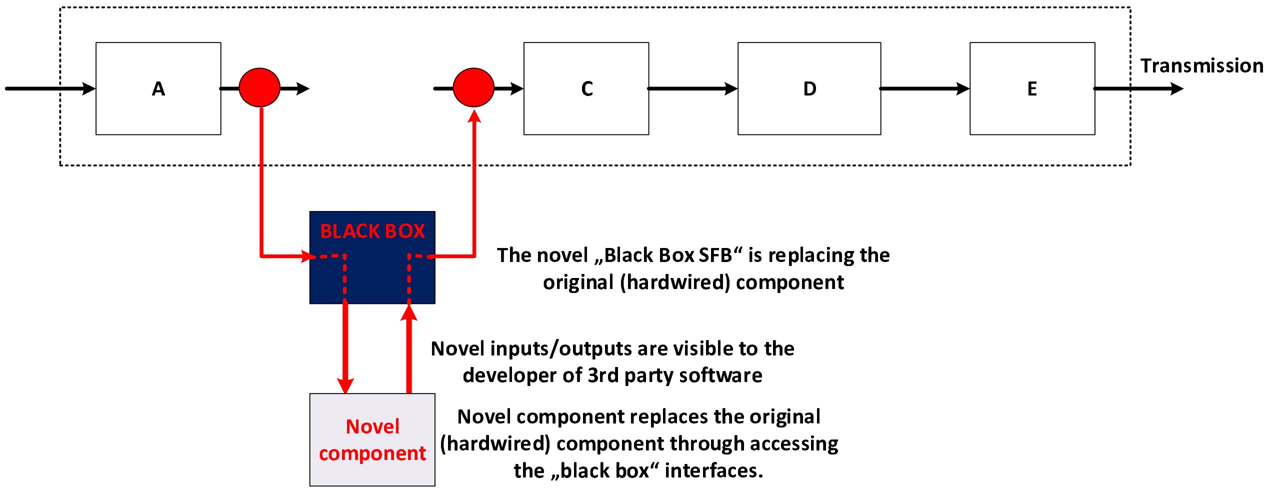

A
C
D
E
Transmission
BLACK BOX
The novel „Black Box SFB“ is replacing the original (hardwired) component
Novel inputs/outputs are visible to the developer of 3rd party software
Novel component
Novel component replaces the original (hardwired) component through accessing the „black box“ interfaces.

Figure C.2: Example replacement of a component through interfacing with novel SFB provided by a (3rd party) software provider

Annex D (informative): Introducing new SFBs

As for the determination of new SFBs, Radio Library Authority (RLA) will determine whether or not a candidate SFB is registered as a new SFB. The reusability of the candidate Functional Block (FB) is a key factor in the determination of SFB. Therefore, the RLA will evaluate "how importantly and/or how often is the candidate FB used in a given RA(s)?".

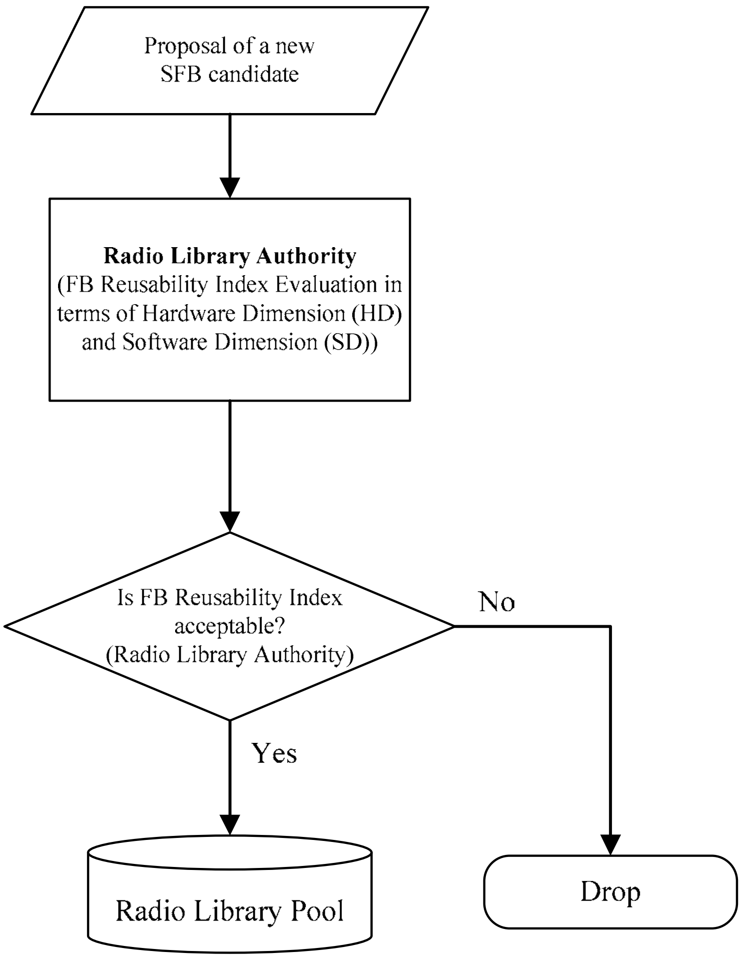

Proposal of a new SFB candidate
Radio Library Authority
(FB Reusability Index Evaluation in terms of Hardware Dimension (HD) and Software Dimension (SD))
Is FB Reusability Index acceptable? (Radio Library Authority)
No
Yes
Radio Library Pool
Drop

Figure D.1: Flow chart for determining SFBs

Figure D.1 illustrates a typical procedure for RLA to determine a new SFB. In order to be considered as an SFB candidate, the candidate FB will be presented in the form of a reference code, which consists of a proper combination of eSFBs. Since the reusability is the key factor of determining a new SFB, an evaluation quotient such as FB Reusability Index (FBRI) as shown in figure D.1 should be set up by the RLA in the viewpoint of both hardware- and software-related factors of the candidate FB. Hardware Dimension (HD) and Software Dimension (SD) shown in figure D.1 are concerned with hardware- and software-related factors of the candidate FB, respectively. In the evaluation for HD, the candidate FB might be evaluated in terms of the degree of performance improvement, hardware implementation difficulty, time to market, etc. In the evaluation for SD, the candidate FB might be evaluated in terms of the software correctness and completeness, complexity and/or reusability of the FB algorithm, etc.

Annex E (informative): Synchronous Approach

The Radio Virtual Machine is illustrated in figure E.1 as defined in the present document. The Abstract Switch Fabric connects Data Objects ("data") with Abstract Processing Elements (APEs). A functional block is processing its input data as soon as all Data Objects connected to the APE are "full".

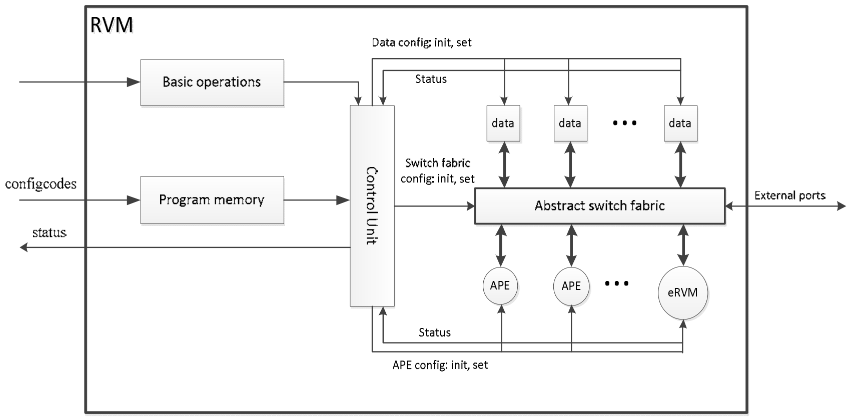


Figure E.1: Radio Virtual Machine

Depending on the target application, there is typically a selection between a synchronous and asynchronous approach.

NOTE: The Radio Virtual Machine as illustrated in figure E.1 is used to represent an algorithm in a concurrent way. The description of an algorithm is typically independent on implementation choices, including a synchronous or asynchronous implementation approach. Ideally, a compiler would process the algorithm description such that the desired implementation approach is applied. In practice, however, it can be preferred by a designer to facilitate the tasks of a compiler and describe the algorithm closer to an hardware implementation level, including provisions for a synchronous or asynchronous implementation approach. This is a common approach in hardware design programming languages such as VHDL (Very High Speed Integrated Circuit Hardware Description Language). Also, some critical applications requiring deterministic behaviour need to be implemented using a synchronous approach.

In order to synchronize the design to a clock signal, an input to a Data Object is generated leading to a "full Data Object" state when the clock changes to a predetermined state (e.g. from LOW to HIGH or from HIGH to LOW) as indicated below.

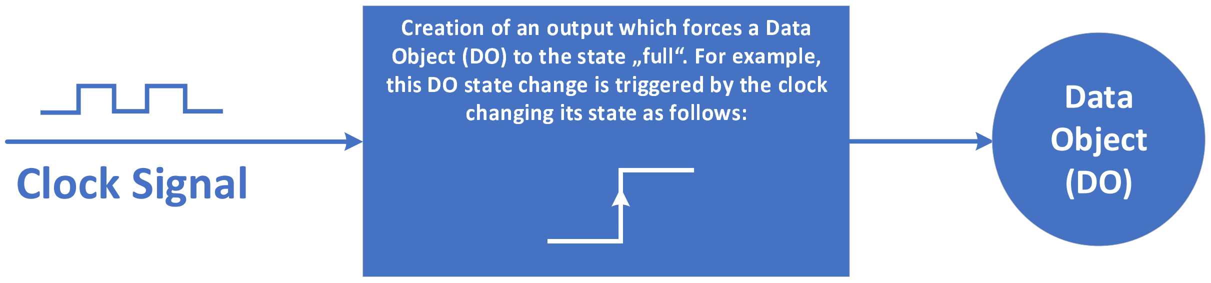

Creation of an output which forces a Data Object (DO) to the state „full“. For example, this DO state change is triggered by the clock changing its state as follows:
Clock Signal
Data Object (DO)

Figure E.2: Input to Data Object is generated when the clock changes from LOW to HIGH

Alternatively, the DO state change can also be enforced by a clock state change from HIGH to LOW.

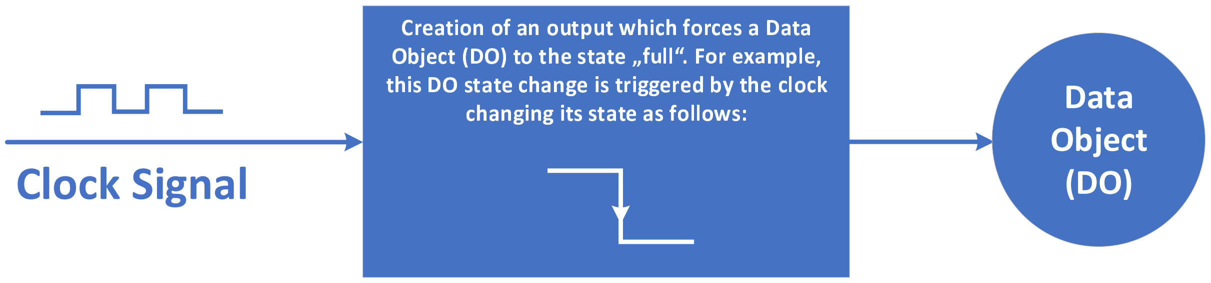

Clock Signal
Creation of an output which forces a Data Object (DO) to the state „full". For example, this DO state change is triggered by the clock changing its state as follows:
Data Object (DO)

Figure E.3: Input to Data Object is generated when the clock changes from HIGH to LOW

As yet another alternative, the DO state change can be enforced by any state change of the clock, i.e. from HIGH to LOW as well as from LOW to HIGH.

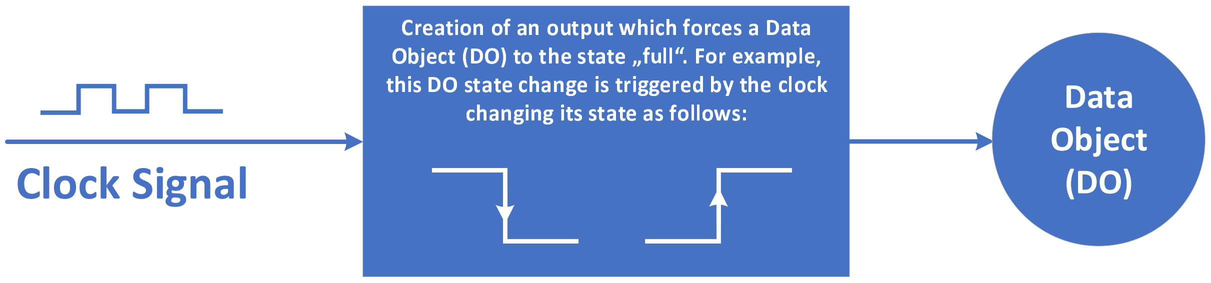

Clock Signal
Creation of an output which forces a Data Object (DO) to the state „full". For example, this DO state change is triggered by the clock changing its state as follows:
Data Object (DO)

Figure E.4: Input to Data Object is generated when the clock changes from LOW to HIGH or from HIGH to LOW

The related Data Object is used in the following way: Multiple Data Objects are fed with the outputs of the entity indicated above (i.e. generating a "full state" of the Data Object when the clock state changes from LOW to HIGH, from HIGH to LOW or in both cases). Then, different APEs are connected to different such Data Objects enforcing the execution to be synchronous.

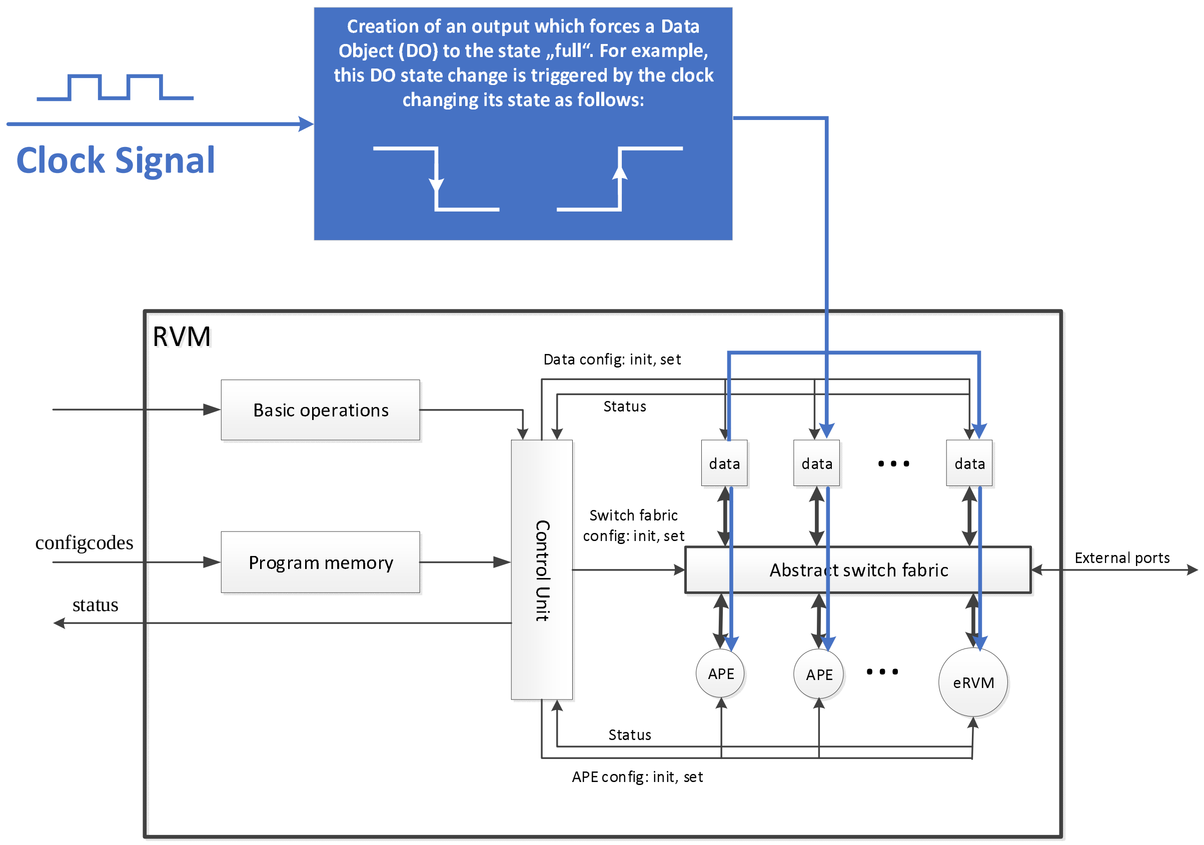

Clock Signal
Creation of an output which forces a Data Object (DO) to the state „full". For example, this DO state change is triggered by the clock changing its state as follows:
RVM
Data config: init, set
Status
Basic operations
data
data
data
Switch fabric config: init, set
configcodes
Program memory
Control Unit
Abstract switch fabric
External ports
status
APE
APE
eRVM
Status
APE config: init, set

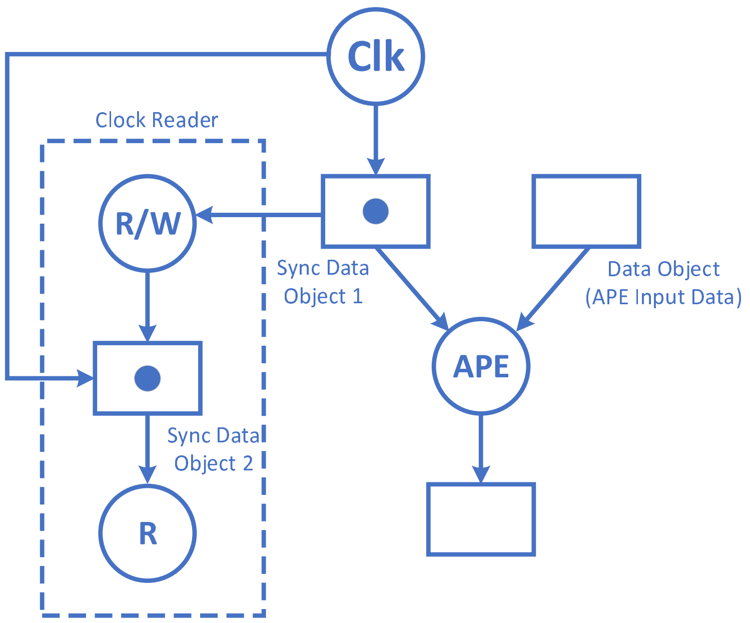
Figure E.5: Connection of a Data Object fed upon clock state changes to a single APEs

Note that there are two ways of creating the full Data Object state:

i) RVM internal: One or multiple entities inside the RVM are creating the full Data Object state in function of a clock.

ii) External: Data Objects for synchronization are part of the design in the present document, but the creation of the full Data Object state for clock synchronization is out of scope.

An example for clock distribution is given below, consisting of a Clock generator (Clk), a Clock Reader and an APE connected to a input Data Object, a Sync Data Object and an output Data Object. The specific case is addressed that a the input Data Object remains empty during multiple clock cycles. Typically, a single Clock generator feeds independent Sync Data Objects for each APE and corresponding Clock Readers.

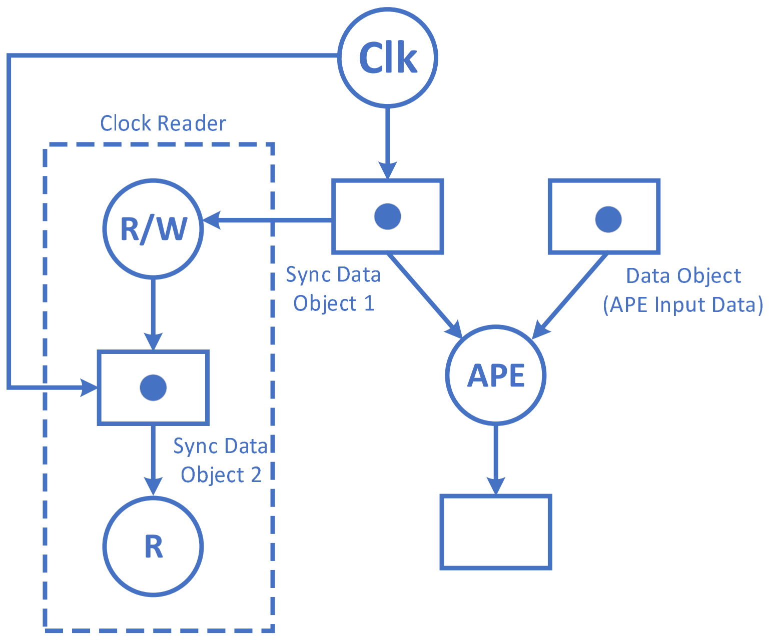


Issue to be addressed: Assume that Data Object remain empty for several clock cycles. Then, Clk would overwrite full Sync Data Object 1 which would trigger an exception. This exception is avoided by the following mechanism:

1) Sync Data Object 2 is full, blocking R/W operation.
2) Operator R consumes Sync Data Object 2 content.
3) If Data Object is empty, operator R/W will consume Sync Data Object 1 content.
4) Clk will create new input to Sync Data Object 1 and 2 accessing to an empty Data Object.

Figure E.6: Example for input Data Object remaining empty during one or multiple Clock cycles

An example for a full Data Object case is given below, consisting of a Clock generator (Clk), a Clock Reader and an APE connected to a input Data Object, a Sync Data Object and an output Data Object.



Issue to be addressed: Assume that Data Object is full, block R/W operation from reading Sync Data Object 1 and empty all Data Objects after reading. This is achieved by the following mechanism:

1) Sync Data Object 2 is full, blocking R/W operation.
2) Operator R consumes Sync Data Object 2 content.
3) If Data Object and Sync Data Object 1 are full, so APE is executed and consumes both Data Object contents. Data Object, Sync Data Object 1 and 2 are empty.
4) Clk will create new input to Sync Data Object 1 and 2.

Figure E.7: Example for full input Data Object

In the approaches presented above, exception handling needs to be addressed. Ideally, full Data Objects provide inputs to Abstract Processing Elements which execute the respective processing during the duration of a clock cycle. Then, the Data Objects return to an empty state and are being filled again for processing during the next clock cycle. In case of an exception, the concerned Data Object may not return to an empty state before the end of the clock cycle. In this situation, an exception may occur if new data is provided to a Data Object although it is still in full state. In such a case, it is proposed that the concerned Data Objects are forced to empty spaces and the Data Objects contents are destroyed. The Control Unit will trigger a corresponding exception status information.

History

Document history		
V1.1.1	March 2020	Publication