ETSI GS ECI 001-4 V1.1.1 (2017-07)

Embedded Common Interface (ECI) for exchangeable CA/DRM solutions; Part 4: The Virtual Machine

Reference
DGS/ECI-001-4

Keywords
CA, DRM, VM

Contents

List of Figures

List of Tables

Intellectual Property Rights

Essential patents

Trademarks

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Embedded Common Interface (ECI) for exchangeable CA/DRM solutions.

The present document is part 4 of a multi-part deliverable covering the Virtual Machine for the Embedded Common Interface for exchangeable CA/DRM solutions specification, as identified below:

Part 1: "Architecture, Definitions and Overview";

Part 2: "Use cases and requirements";

Part 3: "CA/DRM Container, Loader, Interfaces, Revocation";

Part 4: "The Virtual Machine";

Part 5: "The Advanced Security System";

Part 6: "Trust Environment".

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.

Introduction

The present document describes the concept of a Virtual Machine that executes in a Sandbox and offers a range of instructions and System Call functions. The VM is designed to work in a variety of environments. It interoperates with other applications that exist on the same machine using well-defined interfaces and provides a combination of support for its own instruction set and a modular mechanism for the execution of elements written in the native code of the **ECI Host** CPU and interacting with the hardware and other elements of the **ECI Host** environment. This provides the VM with the means to execute readily renewable code that can provide a wide range of potential secure applications, including the implementation of CA/DRM clients.

1 Scope

The present document specifies a Virtual Machine which is intended for inclusion in the implementation of digital television receivers and Set Top Boxes, and which is able to provide a secured environment for executing Conditional Access kernel or Digital Rights Management client applications. The intention is to provide a uniform execution environment in which such clients can operate in the knowledge that minimum **ECI Host** performance requirements are met, that a standard API is provided to be used for retrieval of essential security data from content (i.e. encapsulated with content) or via external networks (e.g. the Internet) and where resources can be accessed from the **ECI Host** environment in a standardized way.

The presence and use of the VM allows to exchange CA/DRM clients at will and to support multiple simultaneous instances of such clients in **ECI Hosts** so that users and operators are not tied in to a particular content protection provider and that they can use security solutions of different types to suit differing content types. For **Content Protection system** providers, it ensures the availability of a known execution platform that does not require specific integration with any and every vendor of **ECI Host** devices.

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 http://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.

The following referenced documents are necessary for the application of the present document.

[1] ETSI GS ECI 001-3: "Embedded Common Interface (ECI) for exchangeable CA/DRM solutions; Part 3: CA/DRM Container, Loader, Interfaces, Revocation".

[2] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, version 1.2", TIS Committee, 1995.

NOTE: Available at https://refspecs.linuxfoundation.org/elf/elf.pdf.

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] ISO/IEC 9899: "Information technology -- Programming Languages -- C", ISO/IEC JTC1/SC22 WG14.

[i.2] ETSI GR ECI 004: "Embedded Common Interface (ECI) for exchangeable CA/DRM solutions; Guidelines for the implementation of ECI".

3 Definitions and abbreviations

3.1 Definitions

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

bytecode: code of **ECI Client** (typically comprising a Conditional Access kernel or Digital Rights Management client) that is executed by the VM

content protection system: system that uses cryptographic techniques to manage access to digital content

NOTE: Typically, a **content protection system** is either a conditional access system or a digital rights management system.

Customer Premises Equipment (CPE): customer device that provides **ECI** specified decryption and encryption functions

ECI (Embedded CI): architecture and the system specified in the ETSI ISG "Embedded CI", which allows the development and implementation of software-based swappable **ECI Clients** in customer premises equipment (**CPE**) and thus provides interoperability of **CPE** devices with respect to **ECI**

ECI Client (Embedded CI Client): implementation of a CA/DRM client which is compliant with the ECI specifications

ECI Host: hardware and software system of a **CPE**, which covers **ECI** related functionalities and has interfaces to an **ECI Client**

ecosystem: content and system environment in which the Virtual Machine described in the present document exists

NOTE: It takes into account the wider perspective of content preparation, delivery, authorization, etc. and is not limited to a specific device or implementation.

interface specification: wrapper document that describes the extension, restrictions or any other modifications to the present document that are required to meet the specific needs of a wider **ecosystem** in which the VM is required to operate

native code: programmatic code written in the native executable instruction set of the **ECI Host** processor

sandbox: application execution environment limiting application access to only those resources defined by the **sandbox** API

VM Instance: instantiation of VM established by an **ECI Host** that appears to an **ECI Client** as an execution environment to run in

3.2 Abbreviations

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

API	Application Programming Interface
CA	Conditional Access
CI	Common Interface
CP	Content Protection
CPU	Central Processing Unit
DRM	Digital Rights Management
ELF	Executable and Linkable Format
EPG	Electronic Programme Guide
ID	Identification/Identity/Identifier
OS	Operating System
PC	Program Counter
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Computer
VM	Virtual Machine

4 Conceptual principles

4.1 The Virtual Machine as a CPU

In essence, the Virtual Machine (VM) comprises a virtual CPU with its own code and data memory and a set of system interfaces that provide access to hardware features of the **ECI Host** machine. The emulated CPU executes code in the manner of a virtual 32-bit CPU, and the code it executes is called **bytecode** in the present document. Since the VM is a simulation of a general purpose RISC processor it is able to execute a variety of applications.

4.2 Characteristics of the Virtual Machine

The VM shall provide a single-process, single-threaded environment.

The interface to the **ECI Host** hardware and other functions is provided in the form of a standard library of calls, termed SYSCALLs. The SYSCALL instruction is one of the customized instructions of the VM and it is generally executed after preparing the parameters required by the library routine (i.e. passed in "registers" of the VM).

All interaction between the **ECI Client** and the **ECI Host** is achieved through this operation. No **interrupt** architecture is defined and, once started, the **ECI Client** runs to completion. Therefore, there is no opportunity to invoke calls into the VM. Whilst restricting flexibility to a certain extent, this is outweighed by the enhanced control of the VM execution (ensuring robustness of operation), the avoidance of race conditions, interference with time-critical operations, etc.

As a consequence, the only means of passing data or messages to the **ECI Client** executing in the VM is on the basis of requests issued by the **ECI Client** by invoking the appropriate SYSCALLs.

4.3 Isolation of individual **ECI Clients**

The **ECI Client** executes in a Virtual Machine, which exists as an application running in the firmware of the **ECI Host**. It shall be possible to invoke multiple instances of the Virtual Machine, each potentially running a different **ECI Client**. This places three fundamental requirements on the **ECI Host** operating environment:

1) The Operating System shall allocate sufficient resource to each **VM Instance** such that the performance requirements laid out in [i.2] are met by all instances running simultaneously.

2) The libraries defined in clause 6 and annex C shall be fully re-entrant or implemented separately for each instance of the VM.

3) The Operating System and VM shall ensure no information can be exchanged between running **ECI Clients** and the outside world, including other **ECI Clients** by means other than those explicitly specified for such purpose as part of the SYSCALL interface. This among others implies that all memory mapped into the data space of a **VM Instance** is wiped from its previous content beforehand, and any attempts to use exceptional conditions in the VM to trigger unspecified behaviour shall be prevented. This also implies that there is no means for an **ECI Client** to change its **bytecode**. It specifically implies that the **ECI Host** and VM shall make all required checks to prevent an **ECI Client** inducing unintended behaviour in the **ECI Host** or VM implementations that may for instance lead directly or indirectly lead to the **ECI Client** being able to manipulate (hack) the **ECI Host**.

4.4 Specifying the Virtual Machine

In subsequent clauses of the present document, the following are explicitly detailed regarding the VM itself:

1) The technical architecture of the VM.

2) The instruction set of the VM.

3) The **ECI Host** interface.

4.5 **ECI Client** loader

In order to execute the **ECI Client**, the **bytecode** shall first be loaded into the code space of the VM memory and the data space initialized. In clause 7, the present document addresses some specific aspects of the format of the **ECI Client** container and initialization of the VM.

5 The Virtual Machine

5.1 Execution environment

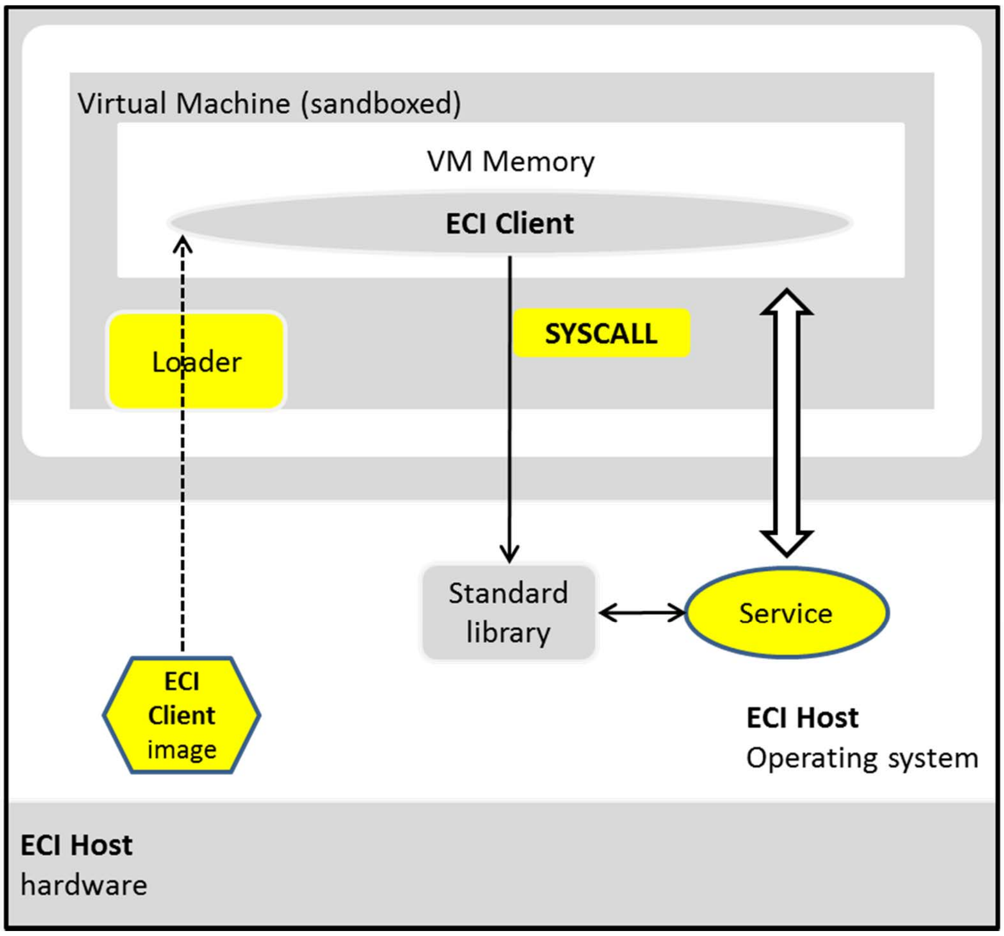

Figure 1: VM Host environment

As depicted in Figure 1, the VM shall be executed in a sandboxed environment that ensures isolation from the **ECI Host**'s operating system, other Virtual Machine instances and any other applications executing in the **ECI Host**.

The VM comprises a native application of the **ECI Host**, with associated memory, and interface library and a loader for installing the **bytecode** forming an **ECI Client**. The interface library provides the **ECI Client** with access to features of the **ECI Host** operating system and hardware and also to other applications that may be executing in the **ECI Host** and with which the **ECI Client** may need to interact. A typical example would be interaction with an Electronic Programme Guide (EPG) application that would require authorization status for specific content for display to the user.

5.2 Virtual Machine Architecture

5.2.1 CPU architecture

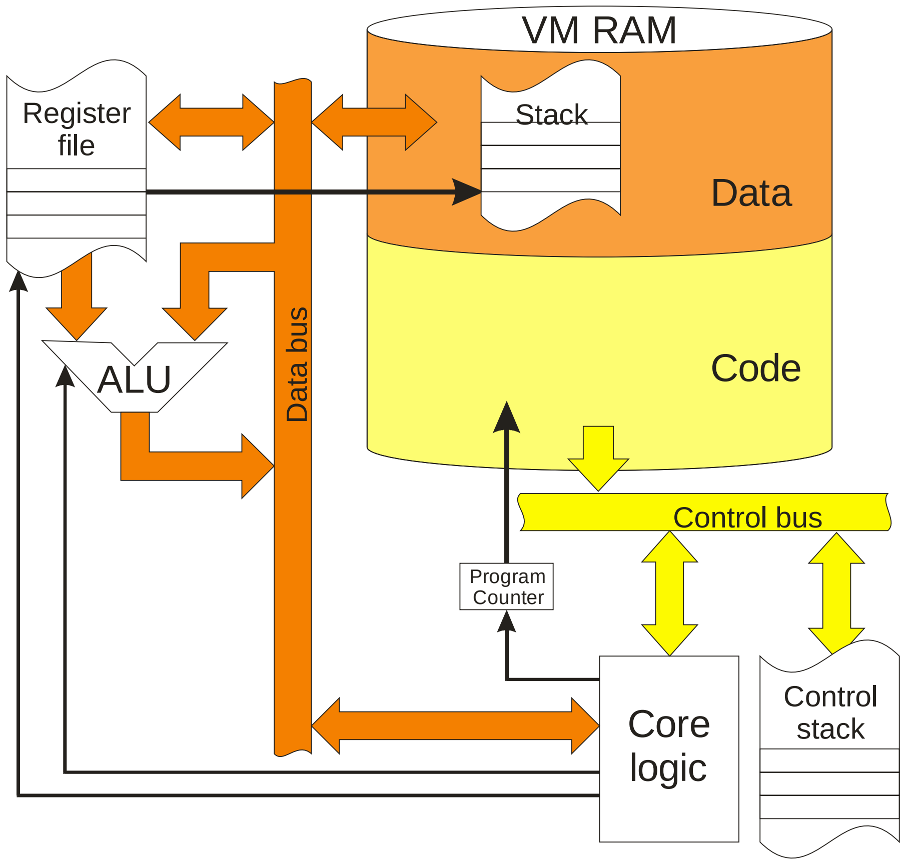

Figure 2: Virtual processor architecture

Figure 2 shows architecture of the virtual machine CPU. The VM is a register machine with the following characteristics.

- A register file with general purpose registers of 32 bits. The registers are organized in register windows. Each register window contains 32 registers. The last 16 registers of each window overlap with the first 16 registers of the next window. Two of these registers in each window serve as stack pointer and frame pointer. The total number of registers in the register file is REGISTER_FILE_SIZE, specified in annex A.
- A Harvard CPU architecture. Data is stored in a 32-bit flat memory space. Code is stored in a read-only, non-addressable memory space.
- A separate control stack keeps track of return addresses. The contents of this stack are inaccessible to the **bytecode** or external applications. The stack can store up to CONTROL_STACK_SIZE return addresses (see annex A).
- Load and store instructions for signed and unsigned byte, half-word and word data types, which are 8, 16 and 32 bits respectively.
- An instruction set with many data processing instructions tailored for the application domain.
- Native byte ordering for efficient load and store, independent of endianness. Natural alignment (alignment = size) is used for the basic types to make the **bytecode** maximally portable. In other words, the memory address of a half-word is always even, and the address of a word is always a multiple of four.
- A System Call instruction (SYSCALL) which can be used to implement system services. This also allows the VM to be extended with built-in functions, e.g. to perform frequently occurring data processing natively.

- Paged memory supporting a fragmented memory space. It allows mapping of native memory into the VM's memory space.

5.2.2 Registers

In each register window, 32 registers are visible, R0 through R31. Two registers are reserved for special treatment. R0 is the Frame Pointer and R16 is the Stack Pointer. The use of these registers is further detailed below.

At entry of a function, the ENTER instruction shifts the register window up by sixteen registers. This turns the old stack pointer into the new frame pointer, and makes a new stack pointer and fifteen more registers available. The new stack pointer is initialized by subtracting the frame size supplied by the ENTER instruction from the frame pointer.

The RETURN instruction reverses this process. It shifts the window down by sixteen registers, thus restoring the old frame pointer and stack pointer.

Since the original R0 through R15 cannot be reached from the called routine, they are automatically callee-saved. Since the return address is saved on a separate control stack, there is no data stack used for callee-saved registers and return addresses.

The true number of registers is limited, so there is a maximum on the call depth of an **ECI Client** (CONTROL_STACK_SIZE). Exceeding this depth will abort the VM program. The number of registers and the corresponding depth of the control stack can be specified when creating the VM process.

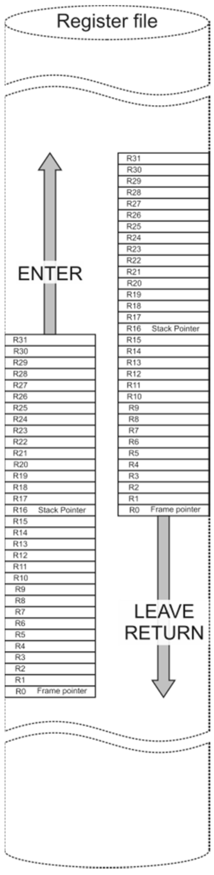

Figure 3: Register file architecture

5.2.3 Data space

The base data address of the VM defined as DATA_BASE_ADDRESS (see annex A) shall be 0x1000000 (16 Mbyte). The smallest address above the addressable memory that is not addressable is DATA_BASE_ADDRESS + ADDRESSABLE_DATA_SIZE (see annex A). The base address of the stack shall be defined by the VM implementation, but shall be towards the high end of the address space. The VM may reserve a maximum of VM_RESEVED_SIZE (see annex A) for private purposes in the address space of the **ECI Client** "below" the bottom of the stack (at a higher address). At VM initialization the stack pointer shall point to the first free stack location. The **ECI Client** can assume that the top of the (empty) heap at initialization is equal to the size of the initialized data + size of the uninitialized data segments, both rounded up to multiple of 4.

The data memory layout is sketched in Figure 4.

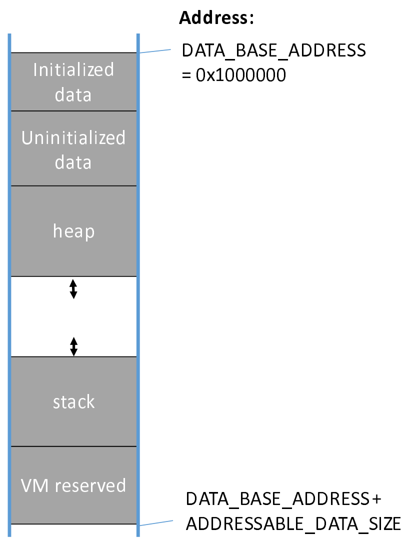

Figure 4: VM data memory layout

At **ECI Client** initialization, the **ECI Client** loader shall load the initialized and uninitialized data segments starting at address DATA_BASE_ADDRESS. All bytes of the uninitialized data segment shall be set to zero. The initialized data segment is not write protected.

NOTE 1: The stack size is initially restricted. Since local data structures defined in c-functions are typically allocated to the stack the stack segment should be set by the **ECI Client** to an appropriate size in case large local variables are used in the c-code.

NOTE 2: The **ECI Host** may map message buffers in the VM reserved memory below the base stack address.

Future VM versions might reserve more addressable memory for **ECI Clients**; i.e. might have a larger ADDRESSABLE_DATA_SIZE. For backward compatibility purposes **ECI Clients** shall not depend on the specific value or value range of the stack pointer presently defined, but simply use the stack pointer as passed on initialization.

The **ECI Client** Loader shall not load any image files that do not adhere to the above memory layout convention for the initialized and uninitialized data segments.

5.2.4 Code space

Code cannot be directly accessed by the program. The program may obtain 32-bit opaque references to static code objects (e.g. routine entry point, jump target) called *code references* (see MOVF instruction). *code references* may only be used with indirect control flow instructions (JMPR and CALLR). *code references* are not pointers to code memory space, and no pointer arithmetic shall take place with them.

The start address of the code segment in the code address space shall be 0x00000000. The maximum size of the code segment is defined as CODE_SIZE (see annex A).

5.2.5 Stack

The stack is conventionally defined to be located in data memory, to contain words only, to grow toward lower addresses, and to have its tip (word that was pushed last) always pointed to by the R16 register (*stack pointer*) of the current register window.

R0, the frame pointer, is used as a pointer into the callee's stack so that parameters or other data pushed onto the stack may be accessed by a called routine (see clause 5.2.8).

5.2.6 Endianness

Multi-byte data (half-words and words) are represented in system memory in little-endian format. The **ECI Client** software shall use little-endian.

5.2.7 Exceptions

The VM CPU does not issue any exception during execution. If an instruction operates under conditions outside those outlined in the present document (e.g. unaligned access to a half-word or word in memory, access to any memory address which has no corresponding memory, a branch to an unknown code reference), the behaviour is undefined. The VM may choose to terminate the kernel. The VM shall ensure that under no circumstances may an **ECI Client** operating outside the present document gain access to unauthorized data or to influence any other application.

5.2.8 Calling convention

The calling conventions pass the first seven scalar parameters (pointers and integers) in R17 through R23. The callee will see these as R1 through R7.

Scalar parameters beyond the seventh are passed on the stack by the caller in a right-to-left order. Because of the register window mechanism, the callee will always find the eighth parameter (if present) pointed to by R0. R0 is therefore the *frame pointer*. Structure parameters are always passed on the stack, or by reference. Pointers always refer to the VM memory space.

NOTE: All SYSCALLs pass any structures and arrays by reference only. This approach should be used for other calls, too.

The callee leaves the return value in R1, which will be seen as R17 by the caller. Types smaller than 32 bits are passed (and returned) as 32 bit values.

Structure return is implemented by passing an implicit first parameter which is a pointer to the memory area where the return type is expected to be stored (passed by reference). The callee writes its result to the location to which this parameter points. This return pointer is treated like a normal argument (passed in R17 → R1), which implies that the regular arguments of a function, which returns a structure, shift to other calling convention registers (R18..R23 → R2..R7) or via the stack.

5.3 Virtual Machine instruction set

5.3.1 Notation

The following notation is used:

```
rx        Register x.
uimm5     5 bit unsigned immediate.
uimms9    9 bit unsigned immediate. Always a multiple of two.
uimms10   10 bit unsigned immediate. Always a multiple of four.
simm11    11 bit signed immediate.
simm16    16 bit signed immediate.
uimm16    16 bit unsigned immediate.
pcr16     16 bit signed PC-relative
pcr24     24 bit signed PC-relative
imm32     32 bit immediate.
low8(x)   The least significant 8 bits of x.
low16(x)  The least significant 16 bits of x.
```

The functional descriptions use C-semantics on 32 bit integer types. The ability of the operation to support signed or unsigned data types is indicated as comments. Memory access is given by MEM1(), MEM2() and MEM4(), accessing 1, 2 or 4 bytes of memory, respectively. The operand of these is an offset into the data segment. When relevant, MEM is prefixed by U for unsigned operations or S for sign-extended operations.

5.3.2 Arithmetic Instructions

5.3.2.1 Register operands

```
ADD     r1,r2,rd        ; rd = r1 + r2;
SUB     r1,r2,rd        ; rd = r1 - r2;
OR      r1,r2,rd        ; rd = r1 | r2;
AND     r1,r2,rd        ; rd = r1 & r2;
XOR     r1,r2,rd        ; rd = r1 ^ r2;
SRA     r1,r2,rd        ; rd = r1 >> r2;     signed shift right
SRL     r1,r2,rd        ; rd = r1 >> r2;     logic shift right
SLL     r1,r2,rd        ; rd = r1 << r2;
MUL     r1,r2,rd        ; rd = r1 * r2;
SDIV    r1,r2,rd        ; rd = r1 / r2;      signed divide
SMOD    r1,r2,rd        ; rd = r1 % r2;      signed remainder
UDIV    r1,r2,rd        ; rd = r1 / r2;      unsigned divide
UMOD    r1,r2,rd        ; rd = r1 % r2;      unsigned remainder
EQ      r1,r2,rd        ; rd = r1 == r2;
NE      r1,r2,rd        ; rd = r1 != r2;
LT      r1,r2,rd        ; rd = r1 < r2;      signed less than
GE      r1,r2,rd        ; rd = r1 >= r2;     signed greater or equal
LTU     r1,r2,rd        ; rd = r1 < r2;      unsigned less than
GEU     r1,r2,rd        ; rd = r1 >= r2;     unsigned greater or equal
NOT     r1,rd           ; rd = ~r1;
NEG     r1,rd           ; rd = -r1;
ABS     r1,rd           ; rd = abs(r1);
MOV     r1,rd           ; rd = r1;
EXTB    r1,rd           ; rd = (int8_t) r1;   sign-extend from 8 bits
EXTH    r1,rd           ; rd = (int16_t) r1;  sign-extend from 16 bits
ZEXTB   r1,rd           ; rd = (uint8_t) r1;  zero-extend from 8 bits
ZEXTH   r1,rd           ; rd = (uint16_t) r1; zero-extend from 16 bits
MASKHI  r1,rd           ; rd = ~(-1) >> r1;   logic shift right
```

5.3.2.2 Register, immediate

```
ADDI    r1,imm32,rd        ; rd = r1 + imm32;
RSUBI   r1,imm32,rd        ; rd = imm32 - r1;
ORI     r1,imm32,rd        ; rd = r1 | imm32;
NORI    r1,imm32,rd        ; rd = ~(r1 | imm32);
ANDI    r1,imm32,rd        ; rd = r1 & imm32;
NANDI   r1,imm32,rd        ; rd = ~(r1 & imm32);
XORI    r1,imm32,rd        ; rd = r1 ^ imm32;
XNORI   r1,imm32,rd        ; rd = ~(r1 ^ imm32);
SRAI    r1,uimm5,rd        ; rd = r1 >> uimm5;     signed
SRLI    r1,uimm5,rd        ; rd = r1 >> uimm5;     logic
SLLI    r1,uimm5,rd        ; rd = r1 << uimm5;
MULI    r1,imm32,rd        ; rd = r1 * imm32;
MACI    r1,imm32,rd        ; rd += r1 * imm32;
SMODI   r1,imm32,rd        ; rd = r1 % imm32;      signed
SDIVI   r1,imm32,rd        ; rd = r1 / imm32;      signed
UMODI   r1,imm32,rd        ; rd = r1 % imm32;      unsigned
UDIVI   r1,imm32,rd        ; rd = r1 / imm32;      unsigned
EQI     r1,imm32,rd        ; rd = r1 == imm32;
NEI     r1,imm32,rd        ; rd = r1 != imm32;
LTI     r1,imm32,rd        ; rd = r1 < imm32;      signed
GTI     r1,imm32,rd        ; rd = r1 > imm32;      signed
GEI     r1,imm32,rd        ; rd = r1 >= imm32;     signed
LEI     r1,imm32,rd        ; rd = r1 <= imm32;     signed
LTUI    r1,imm32,rd        ; rd = r1 < imm32;      unsigned
GTUI    r1,imm32,rd        ; rd = r1 > imm32;      unsigned
GEUI    r1,imm32,rd        ; rd = r1 >= imm32;     unsigned
LEUI    r1,imm32,rd        ; rd = r1 <= imm32;     unsigned
ADDMXI  r1,imm32,rd        ; rd = (r1 + imm32) % 0x7fffffff;
MOVC    simm16,rd          ; rd = simm16;
MOVI    imm32,rd           ; rd = imm32;
MOVF    caddr,rd           ; rd = caddr;           load code reference
CLR     rd                 ; rd = 0;
```

```
INC     rd               ; rd = rd + 1;
DEC     rd               ; rd = rd – 1;
```

The signed divide and remainder operations follow the C99 definition: Division truncates the mathematical result toward zero; the remainder respects the relation:

$$\frac{a}{b}\times b + a\%b = a$$

Where % represents the remainder function, or modulus.

The right operand of the shift instructions shall be in range of [0, 31], otherwise the behaviour is undefined. The signed shift right copies the original most-significant bit into the vacated positions. Arithmetically, this corresponds to division with a power of two, rounding the mathematical result to minus infinity (*floor* rounding).

5.3.3 Short Forms

Many occurrences of the three operand instructions use one of the operands also as the result. Since these can be coded more compactly, special opcodes for these are available:

```
ADD2    r1,rd                         ; rd += r1;
SUB2    r1,rd                         ; rd -= r1;
MUL2    r1,rd                         ; rd *= r1;
AND2    r1,rd                         ; rd &= r1;
OR2     r1,rd                         ; rd |= r1;
XOR2    r1,rd                         ; rd ^= r1;
XNOR2   r1,rd                         ; rd = ~(rd ^ r1);
NE2     r1,rd                         ; rd = r1 != rd;
EQ2     r1,rd                         ; rd = r1 == rd;
SLL2    r1,rd                         ; rd <<= r1;
SRA2    r1,rd                         ; rd >>= r1;          signed
SRL2    r1,rd                         ; rd >>= r1;          logical
```

Bitwise immediate operations test or modify a single bit. Those immediates can be coded using 5 bits giving the bit position.

```
ANDB   r1,uimm5,rd                    ; rd = r1 & (1 << uimm5);
ORB    r1,uimm5,rd                    ; rd = r1 | (1 << uimm5);
XORB   r1,uimm5,rd                    ; rd = r1 ^ (1 << uimm5);
TESTB  r1,uimm5,rd                    ; rd = (r1 >> uimm5) & 1;
TESTBC r1,uimm5,rd                    ; rd = ! ((r1 >> uimm5) & 1);
```

Many comparisons are against zero. This saves an immediate operand and will be cheaper to emulate.

```
EQZ     r1,rd                         ; rd = r1 == 0;
NEZ     r1,rd                         ; rd = r1 != 0;
LTZ     r1,rd                         ; rd = r1 < 0;
GTZ     r1,rd                         ; rd = r1 > 0;
LEZ     r1,rd                         ; rd = r1 <= 0;
GEZ     r1,rd                         ; rd = r1 >= 0;
```

Unsigned versions of these do not make sense. They are either true or false or can be expressed using `EQZ` or `NEZ`.

5.3.4 Control Flow

5.3.4.1 Common rules

Control flow instructions with direct operands code their targets relative to the end address of the instruction. In register-based control flow, the register holds a function pointer index.

5.3.4.2 Unconditional Branches and Function Calls

```
JMP     pcr24              ; goto PC+pcr24;
JMPR    rd                 ; goto rd (shall be code reference);
CALL    pcr24              ; push PC; goto PC+pcr24;
CALLR   rd                 ; push program counter; goto rd (code reference);
ENTER   uimm16             ; shift register file by 16 (new r0 is old r16);
                           ; r16 = r0 - 4 * uimm16;
ENTER0                     ; equivalent to ENTER 0
ENTERC  uimms10            ; equivalent to ENTER uimms10
LEAVE                      ; unshift register file
RETURN                     ; unshift register file;
                           ; goto popped program counter;
RETURNL                    ; goto popped program counter;

SWITCH  r1,uimm16          ; goto PC + MIN(r1, uimm16)
                         ; advance to r1th CASE statement below
CASE    pcr24              ; goto PC + pcr24
                         ; add a case in the previous SWITCH. The first
                           ; entry is case value zero, each next one adds
                           ; one to the case value.
```

5.3.4.3 Conditional Branches

```
JEQ     r1,r2,pcr16            ; if (r1 == r2) goto PC+pcr16;
JNE     r1,r2,pcr16            ; if (r1 != r2) goto PC+pcr16;
JLT     r1,r2,pcr16            ; if (r1 < r2) goto PC+pcr16;
JGE     r1,r2,pcr16            ; if (r1 >= r2) goto PC+pcr16;
JLTU    r1,r2,pcr16            ; if ((unsigned)r1 < (unsigned)r2) goto PC+pcr16;
JGEU    r1,r2,pcr16            ; if ((unsigned)r1 >= (unsigned)r2) goto PC+pcr16;

JEQC    r1,simm11,pcr16        ; if (r1 == simm11) goto PC+pcr16;
JNEC    r1,simm11,pcr16        ; if (r1 != simm11) goto PC+pcr16;
JLTC    r1,simm11,pcr16        ; if (r1 < simm11) goto PC+pcr16;
JGEC    r1,simm11,pcr16        ; if (r1 >= simm11) goto PC+pcr16;
JLTUC   r1,uimm11,pcr16        ; if ((unsigned) r1 < uimm11) goto PC+pcr16;
JGEUC   r1,uimm11,pcr16        ; if ((unsigned) r1 >= uimm11) goto PC+pcr16;
JGTC    r1,simm11,pcr16        ; if (r1 > simm11) goto PC+pcr16;
JLEC    r1,simm11,pcr16        ; if (r1 <= simm11) goto PC+pcr16;
JGTUC   r1,uimm11,pcr16        ; if ((unsigned) r1 > uimm11) goto PC+pcr16;
JLEUC   r1,uimm11,pcr16        ; if ((unsigned) r1 <= uimm11) goto PC+pcr16;
```

5.3.4.4 Conditional Branches Based on Memory Comparisons with Constant

```
JWEQC   r1,simm11,pcr16    ; if (MEM4(r1) == simm11) goto PC+pcr16;
JWNEC   r1,simm11,pcr16    ; if (MEM4(r1) != simm11) goto PC+pcr16;
```

These read a word from memory and compare it with a constant.

5.3.4.5 Far Conditional Branches

For each of the conditional branches described above, there is a *far* version, which has a 24 bit offset. The assembler should choose the shortest version that fits.

5.3.5 Load and Store instructions

5.3.5.1 Register + offset

```
LDSBI    r1,imm32,rd                ; rd = SMEM1(r1 + imm32);
LDUBI    r1,imm32,rd                ; rd = UMEM1(r1 + imm32);
LDSHI    r1,imm32,rd                ; rd = SMEM2(r1 + imm32);
LDUHI    r1,imm32,rd                ; rd = UMEM2(r1 + imm32);
LDWI     r1,imm32,rd                ; rd = MEM4 (r1 + imm32);

STBI     rd,r1,imm32                ; MEM1(r1 + imm32) = low8(rd);
STHI     rd,r1,imm32                ; MEM2(r1 + imm32) = low16(rd);
STWI     rd,r1,imm32                ; MEM4(r1 + imm32) = rd;
```

5.3.5.2 Register + short offset

```
LDSBC    r1,uimm8,rd                          ; rd = SMEM1(r1 + uimm8);
LDUBC    r1,uimm8,rd                          ; rd = UMEM1(r1 + uimm8);
LDSHC    r1,uimms9,rd                         ; rd = SMEM2(r1 + uimms9);
LDUHC    r1,uimms9,rd                         ; rd = UMEM2(r1 + uimms9);
LDWC     r1,uimms10,rd                        ; rd = MEM4 (r1 + uimms10);

STBC     rd,r1,uimm8                          ; MEM1(r1 + uimm8) = low8(rd);
STHC     rd,r1,uimms9                         ; MEM2(r1 + uimms9) = low16(rd);
STWC     rd,r1,uimms10                        ; MEM4(r1 + uimms10) = rd;
```

5.3.5.3 Register Indexed

```
LDUB     r1,r2,rd                             ; rd = UMEM1(r1 + r2);
LDSB     r1,r2,rd                             ; rd = SMEM1(r1 + r2);
LDUH     r1,r2,rd                             ; rd = UMEM2(r1 + 2 * r2);
LDSH     r1,r2,rd                             ; rd = SMEM2(r1 + 2 * r2);
LDW      r1,r2,rd                             ; rd = MEM4(r1 + 4 * r2);

STB      rd,r1,r2                             ; MEM1(r1 + r2) = rd;
STH      rd,r1,r2                             ; MEM2(r1 + 2 * r2) = rd;
STW      rd,r1,r2                             ; MEM4(r1 + 4 * r2) = rd;

LDW1     r1,r2,rd                                 ; rd = MEM4(r1 + r2);
STW1     rd,r1,r2                                 ; MEM4(r1 + r2) = rd;
```

5.3.5.4 Absolute indexed

```
LDSHAX   imm32,r1,rd                              ; rd = SMEM2(imm32 + 2 * r1);
LDUHAX   imm32,r1,rd                              ; rd = UMEM2(imm32 + 2 * r1);
LDWAX    imm32,r1,rd                              ; rd = MEM4(imm32 + 4 * r1);
STHAX    rd,imm32,r1                              ; MEM2(imm32 + 2 * r1)= rd;
STWAX    rd,imm32,r1                              ; MEM4(imm32 + 4 * r1)= rd;
```

It should be noted that no absolute indexed byte loads are needed. For instance, `LDSBAX` is equivalent to `LDSBI`.

5.3.5.5 Dedicated Stack Access

These are word loads and stores that use the frame pointer implicitly.

```
LDFP     simm16,r1                                ; r1 = MEM4(FP + simm16);
STFP     r1,simm16                            ; MEM4(FP + simm16) = r1;
```

5.3.5.6 Memory Transfer

A block copy instruction used for compiler-generated block copies.

```
COPY     r1,s:uimm32,r2,o:uimm32          ; copy s bytes from r1 to r2+o
```

5.3.6 Complex Instructions

These are instructions that perform a combination of operations, usually with immediate operands. In this summary each operand designated as `i1`, `i2`, etc. is a 32 bit immediate (imm32).

```
ADDANDI2    r1,i1,i2,rd                       ; rd = (r1 + i1) & i2;
ADDMULI2    r1,i1,i2,rd                       ; rd = (r1 + i1) * i2;
ADDORI2 r1,i1,i2,rd                           ; rd = (r1 + i1) | i2;
ADDXORI2    r1,i1,i2,rd                       ; rd = (r1 + i1) ^ i2;

MULADDI2    r1,i1,i2,rd                       ; rd = (r1 * i1) + i2;
MULANDI2    r1,i1,i2,rd                       ; rd = (r1 * i1) & i2;
MULORI2 r1,i1,i2,rd                           ; rd = (r1 * i1) | i2;
MULXORI2    r1,i1,i2rd                        ; rd = (r1 * i1) ^ i2;

RSUBANDI2   r1,i1,i2,rd                       ; rd = (i1 – r1) & i2;
RSUBORI2    r1,i1,i2,rd                       ; rd = (i1 – r1) | i2;
RSUBXORI2   r1,i1,i2,rd                       ; rd = (i1 – r1) ^ i2;
```

```
ORADDI2 r1,i1,i2,rd                         ; rd = (r1 | i1) + i2;
ORMULI2 r1,i1,i2,rd                         ; rd = (r1 | i1) * i2;

SLLADDI2    r1,s1:uimm5,i2,rd               ; rd = (r1 << s1) + i2;
SLLANDI2    r1,s1:uimm5,i2,rd               ; rd = (r1 << s1) & i2;
SLLORI2 r1,s1:uimm5,i2,rd                   ; rd = (r1 << s1) | i2;
SLLRSUBI2   r1,s1:uimm5,i2,rd               ; rd = i2 - (r1 << s1);

ANDSLLI2    r1,i1,s2:uimm5,rd               ; rd = (r1 & i1) << s2;

MAMI3       r1,i1,i2,i3,rd                  ; rd = ((r1 * i1) & i2) * i3;
MPMI3       r1,i1,i2,i3,rd                  ; rd = ((r1 * i1) + i2) * i3;
MOMI3       r1,i1,i2,i3,rd                  ; rd = ((r1 * i1) | i2) * i3;

MPAI3       r1,i1,i2,i3,rd                  ; rd = ((r1 * i1) + i2) & i3;
MPOI3       r1,i1,i2,i3,rd                  ; rd = ((r1 * i1) + i2) | i3;
RORI3       r1,i1,i2,i3,rd                  ; rd = i3 - ((i1 – r1) | i2);
AMPI3       r1,i1,i2,i3,rd                  ; rd = ((r1 & i1) * i2) + i3;

LPAI3       r1,s1:uimm5,i2,i3,rd            ; rd = ((r1 << s1) + i2) & i3;

MPMPI4  r1,i1,i2,i3,i4,rd                   ; rd = (((r1 * i1) + i2) * i3) + i4;
MPOMI4  r1,i1,i2,i3,i4,rd                   ; rd = (((r1 * i1) + i2) | i3) * i4;
```

5.3.7 Miscellaneous

5.3.7.1 System Calls

A variety of services are implemented by System Calls.

```
SYSCALL  uimm16                             ; system service uimm16
```

A minimal set of POSIX System Calls is implemented that are mapped directly to the underlying OS. These are used for testing purposes. Some others perform more dedicated VM services, like `malloc()`and `free()`. More application-specific services may be added.

5.3.7.2 Pseudo Instructions

Some operations can be expressed in terms of other ones. The following pseudo opcodes are available:

```
SUBI    r1,imm32,rd    = ADDI  r1,-imm32,rd
GT      r1,r2,rd       = LT    r2,r1,rd
LE      r1,r2,rd       = GE    r2,r1,rd
GTU     r1,r2,rd       = LTU   r2,r1,rd
LEU     r1,r2,rd       = GEU   r2,r1,rd
```

6 Interface between the **ECI Client** and the **ECI Host**

6.1 General principles

System Calls arise when the SYSCALL instruction is executed. The instruction contains an immediate operand that identifies the System Call. System Calls are effectively calls to a standard library, passing the parameters as described in clause 5.2.8.

The first 7 parameters (words or pointers) are passed in registers R1..R8. They are all sign extended to 32-bit values if the actual parameter type is an 8 or 16 bit scalar. Return values (words or pointers) shall be placed in R1.

Unless otherwise stated, all memory addresses refer to the VM memory space.

For future compatibility reasons, the **ECI Client** shall clear to zero all registers R1..R8 not used for passing parameters. The content of all registers may be trashed by the library function.

The mandatory library System Calls that all compliant implementations shall support are listed below. The format used provides:

- The SYSCALL ID used as the immediate operand (SYSCALL imm32).
- A description of the library function.
- A declaration in C syntax.
- A description of the parameters and return value.
- Any additional notes.

Parameters and return values are typed using the following convention:

- **uint*nn*** represents an unsigned integer of *nn* bits (*nn* being one of 8, 16 or 32). Values of less than 32 bits shall be zero-extended to 32 bits when placing them in the registers.
- **int*nn*** represents a signed integer. Values of less than 32 bits shall be sign-extended when placing them in the registers.
- **void *** represents a generic pointer.
- **[u]int*nn* *** represents a pointer to one value of type [u]int*nn* or an array of them.
- **struct struct_type *** refers to a pointer to a structure (or an array of structures) in memory - structures are always passed by reference using this convention.

6.2 Error value

Most SYSCALLs return a negative word to indicate an error condition was detected. Table 1 lists the error values.

Table 1: Error values

value	Symbolic name	Meaning
-49	EPERM	A call was made to a non-existent SYSCALL or CLIB function.
-50	EINVAL	One of the parameters is incorrect.
-51	ERRSYSCALLMSGQUEUE	Number of messages sent the **ECI Host** exceeds its buffering capacity.
-52	ERRHEAPSIZE	An inappropriate value for heap size was requested.
-53	ERRSTACKSIZE	An inappropriate value for stack size was requested.

6.3 SYS_EXIT

SYSCALL ID:	0x0001
Description:	Terminates the VM, providing a reason code.
Declaration:	void SYS_EXIT(uint32 reason)
Operands:	The **reason** for termination
Returns:	nothing
Notes:	**reason** takes one of the values listed in Table 2.

Table 2: SYS_EXIT reason values

reason	Meaning
0	Normal termination
0x00000001..0x7FFFFFFF	Error condition, **ECI Client** provider specific
0x80000000..0xFFFFFFFF	Reserved for future use

6.4 SYS_PUTMSG

SYSCALL ID:	0x0003
Description:	Sends an asynchronous message (request or response).
Declaration:	int32 SYS_PUTMSG(MessageBuffer *msg_buffer)
Operands:	**msg_buffer** is a pointer to a message buffer block.
Returns:	The id of the message as assigned by the **ECI Host** (non-negative 16 bit value) or any of the error values below (negative): **ERRSYSCALLMSGQUEUE (**Table 1).
Notes:	The call shall not block in normal **ECI Host** operating conditions. The format of MessageBuffer is defined in [1]. The msg_buffer content is copied by the **ECI Host** and can be reused immediately by the **ECI Client** following the return of the SYSCALL.

6.5 SYS_GETMSG

SYSCALL ID:	0x0004
Description:	Retrieves the next message (be it a request or a result) from the **ECI Host**. The SYSCALL blocks if no message is available.
Declaration:	(MessageBuffer *) SYS_GETMSG()
Operands:	none.
Returns:	The pointer to the buffer containing the next message from the **ECI Host** or any of the error values listed in Table 1.
Notes:	The call will block in case the **ECI Host** has no messages queued for the **ECI Client**. The format of MessageBuffer is defined in [1]. The message buffer content will not be changed by the **ECI Host** until the next SYS_GETMSG SYSCALL. **ECI Clients** that wish to have access to message data after the next SYS_GETMSG call need to copy this data.

6.6 SYS_HEAPSIZE

SYSCALL ID:	0x0100
Description:	A request to **ECI Host** to change heap size to the provided parameter.
Declaration:	int32 SYS_HEAPSIZE(uint32 heapsize)
Operands:	**heapsize**: size to set the heap of the **ECI Client** to. It shall be non-negative, a multiple of 4 and not cause an overrun of the heap in the stack segment.
Returns:	The memory location offset in bytes from DATA_BASE_ADDRESS that is the lowest non-heap memory address in addressable memory, or any error value (negative) below: **ERRHEAPSIZE** (Table 1).
Notes:	The call will block in case the **ECI Host** has no messages queued for the **ECI Client**. At **ECI Client** initialization SYS_HEAPSIZE(0) will return the offset of the start of the heap segment (zero size at that time).

6.7 SYS_STACKSIZE

SYSCALL ID:	0x0200
Description:	A request to **ECI Host** to change stack size to the provided parameter.
Declaration:	int32 SYS_STACKSIZE(uint32 stacksize)
Operands:	**stacksize**: size to set the heap of the **ECI Client** to; it shall be non-negative and a multiple of 4 and not cause an overrun of the stack in the heap segment.
Returns:	The memory location offset in bytes from DATA_BASE_ADDRESS that is the lowest stack memory address in addressable memory, or any error value (negative) below: **ERRSTACKSIZE** (Table 1).
Notes:	The call will block in case the **ECI Host** has no messages queued for the **ECI Client**.

6.8 SYS_SYNCCALL

SYSCALL ID:	0x1000
Description:	the **ECI Client** sends a synchronous message to the **ECI Host** and suspends execution till the return of the System Call.
Declaration:	int32 SYS_SYNCCALL(uint32 tag, p1, p2, p3, …, pn)
Operands:	**tag**: same definition as the MsgTag field of the MessageBuffer structure. The MsgFlags field shall be set to zero and shall be ignored by the **ECI Host**. **p1…pn**: parameters of the synchronous call. For get-messages with a result larger than a 32-bit entity **p1** is the start address of the memory location where the result shall be returned, and **p2.. pn** are the parameters of the set message. All regular parameters including structs and arrays are passed by reference.
Returns:	For get-messages returning a result fitting in 32-bit the result is returned. Otherwise there is no return result. All errors are ignored; erroneous parameter configurations simply produce no result and/or have no effect. Call messages return a status code as defined by their specific semantics. Results can be returned at the location of pointer parameters to the SYSCALL as defined by the specific message semantics.
Notes:	This SYSCALL will not block.

6.9 SYS_CLIB

SYSCALL ID:	0x0300
Description:	This SYSCALL acts as an API to allow standard C library functions to be used by the **ECI Client**. The set of functions supported is detailed in annex C.
Declaration:	SYS_CLIB(uint32 clibfunc, etc.)
Operands:	**clibfunc** identifies the C library function to be called, as described in annex C. All other operands are defined in the list of C function calls.
Returns:	A returned value as detailed in the library in annex C or any of the error values listed in Table 1.
Notes:	As different C library functions take different numbers and types of parameters, these are not explicitly described here. The annex details the format of all operands. All operands to SYS_CLIB are scalar values, or pointers to non-scalar values in the VM memory. Since some C library functions may take non-scalar parameters, the VM shall make the conversion from parameters passed by reference to parameters passed by value before passing the execution to the library.

7 **bytecode** lifecycle

7.1 Introduction

The VM is implemented as a part of the **ECI Host** firmware. It is dynamically loaded/executed by the **ECI Host** operating system when an **ECI Client** needs to execute. Multiple instances can be made available for different **ECI Clients**, if they are required to be simultaneously available.

The **ECI Client** is written in the instruction set of the VM as described above. It is prepared by a CP system vendor (CA provider or DRM operator) and made available to the **ECI Host** as a logical code image. Locally, it is transformed to suit the specific design of the **ECI Host** and its operating environment and loaded into the VM when required. It executes within the VM until it is deliberately terminated (or an error condition occurs) and then the execution of the **ECI Client** is halted and the VM terminates.

7.2 Loading a new **ECI Client** into the VM

VM acts as an intermediate host for an externally provided **ECI Client**, exactly as if the **ECI Client** were a native application executing on the **ECI Host** device. The only difference is that the **ECI Client** is installed by the **ECI Host** device Operating System into the VM, rather than as a native application.

In order to load the **ECI Client**, the VM sub-system first creates a virtual processor context. For loading purposes, this entails allocating the VM memory and installing the code and data segment contents into it as if they were native applications, but where the code and initialized data provided in the ELF [2] file (see annex D for details) are transferred to the memory allocated for the VM.

Since the code segment is not accessible from the program, the implementation may choose to carry out any form of pre-processing on the code (e.g. optimization) at load time. In fact, the present document purely describes the format of the program in the image. The internal representation is fully implementation specific. The only condition is that all code references remain usable by the program with the same semantics.

Alternatively, the **ECI Client** image can be pre-processed when it is first retrieved for the **ECI Host** device and stored in a form that is ready to be loaded on demand. This is a more efficient manner of retaining and launching **ECI Clients** if they are regularly unloaded and reloaded.

7.3 Initialization of the VM

The general CPU context of the VM needs to be created - that is the register file, the control stack, the data and stack areas, and the Program Counter, plus any control/status logic and flags. These are not detailed in the present document, as they are implementation dependent.

The register file is set up so that R0 is located at the start of the register file space. All registers are set to 0. Thus the Frame Pointer and Stack Pointer registers (R0 and R16, respectively) in the first window are set such that when the first word is pushed onto the stack, the Stack Pointer is pre-decremented to -4 (0xFFFFFFFC) and the word is stored there.

The Program Counter is initialized to the start of the code segment unless *e_entry* member of ELF header [2] in the **ECI Client** image file has a non-zero value, in which case the Program Counter is set to the value (virtual address) specified by *e_entry* member. Control is then handed over to the execution component of the VM, termed "The Central Run Loop".

7.4 The Central Run Loop

The essence of the VM is in the central execution loop, which reads and translates each sequential instruction into an appropriate set of actions on registers, VM memory and/or in calls to the library. The loop executes instructions until an exception occurs. Program termination may be part of normal execution practice, for instance if the program executes the `SYS_EXIT` System Call, or it may arise as the result of an error situation, for instance if the control stack overflows.

If the "The Central Run Loop" is terminated, then the VM is shut down and will need to be re-instantiated if the **ECI Client** is required at any point in the future.

Annex A (normative): VM System resources

The following parameters are used in the present document to define performance of the VM. The recommended values for the parameters can be found in [i.2].

- REGISTER_FILE_SIZE
- CONTROL_STACK_SIZE = REGISTER_FILE_SIZE/16
- DATA_BASE_ADDRESS
- ADDRESSABLE_DATA_SIZE
- VM_RESERVED_SIZE
- CODE_SIZE
- DEFAULT_STACK_SIZE
- MIN_RAM

Annex B (normative): Op codes for the VM

The coding below specifies how the instructions are coded in the binary image. The overview below shows the different formats. The summary line presents a comma separated list of the fields that make up the instruction. These are either explicit bits, or a field name followed by a field width indicator. Different opcodes in the same formats are enumerated with the corresponding pattern that occupies the 'op' field. The bits are listed in big-endian order.

As an example, `SUB R3,R5,R17` is coded as:

```
1011 00001 00011 00101 10001
```

or, in nibbles:

```
1011 0000 1000 1100 1011 0001
```

or, in bytes:

```
0xB0, 0x8C, 0xB1
```

Each instruction name is followed by a number, which is the defined opcode number of that instruction. This number shall be the same for all future versions of the VM's instruction set.

Fields in opcodes shall not span more than four bytes. Consequently, 32 bit fields shall start at a byte boundary. No field exceeds 32 bits.

```
0, op:5, r1:5, rd:5
    00000       MOV 16
    00001       ADD2    17
    00010       SUB2    18
    00011       MUL2    19
    00100       AND2    20
    00101       OR2 21
    00110       XOR2    22
    00111       SLL2    23
    01000       SRL2    24
    01001       SRA2    25
    01010       NE2 26
    01011       EQ2 27
    01100       NEZ 28
    01101       EQZ 29
    01110       LTZ 30
    01111       GEZ 31
    10000       GTZ 32
    10001       LEZ 33
    10010       EXTB    34
    10011       EXTH    35
    10100       ZEXTB   36
    10101       ZEXTH   37
    10110       ABS 38
    10111       NEG 39
    11000       NOT 40
    11001       XNOR2   41
    11010       MASKHI  42

100, op:3, r1:5, rd:5, imm:32
    000         ADDI    136
    001         RSUBI   137
    010         ANDI    138
    011         ORI 139
    100         XORI    140
    101         MULI    141
    110         MACI    142
    111         ADDMXI  143

101000, op:2
    00          ENTER0  0
    01          RETURN  1
    10          RETURNL 2
    11          LEAVE   3
```

```
10100100, op:3, rd:5
    000         INC 8
    001         DEC 9
    010         JMPR    10
    011         CALLR   11
    100         CLR 12

1010010100000, op:4, r1:5, r2:5, rd:5
    0000        SDIV    80
    0001        SMOD    81
    0010        UDIV    82
    0011        UMOD    83
    0100        TESTBC  84

101001010001, op:4, r1:5, imm:11, pcr:24
    0000        JFNEC   560
    0001        JFEQC   561
    0010        JFLTC   562
    0011        JFGEC   563
    0100        JFGTC   564
    0101        JFLEC   565
    0110        JFLTUC  566
    0111        JFGEUC  567
    1000        JFLEUC  568
    1001        JFGTUC  569
    1010        JFWNEC  570
    1011        JFWEQC  571

10101000, op:3, r1:5, imm:16
    000         STFP    96
    001         LDFP    97
    010         MOVC    98
    011         SWITCH  99

1011, op:5, r1:5, r2:5, rd:5
    00000       ADD 48
    00001       SUB 49
    00010       MUL 50
    00011       AND 51
    00100       OR  52
    00101       XOR 53
    00110       SLL 54
    00111       SRA 55
    01000       SRL 56
    01001       SLLI    57
    01010       SRAI    58
    01011       SRLI    59
    01100       NE      60
    01101       EQ      61
    01110       LT      62
    01111       GE      63
    10000       LTU 64
    10001       GEU 65
    10010       ANDB    66
    10011       ORB 67
    10100       XORB    68
    10101       LDSB    69
    10110       LDUB    70
    10111       LDSH    71
    11000       LDUH    72
    11001       LDW 73
    11010       LDW1    74
    11011       STB 75
    11100       STH 76
    11101       STW 77
    11110       STW1    78
    11111       TESTB   79

110000, op:2, imm:24
    00          JMP 104
    01          CALL    105
    10          CASE    106

110001000, op:2, rd:5, imm:32
    00          MOVI    108
    01          MOVF    109
```

```
110001001, op:5, r1:5, rd:5, imm:32
    00000       NANDI   144
    00001       NORI    145
    00010       XNORI   146
    00011       NEI 147
    00100       EQI 148
    00101       LTI 149
    00110       GEI 150
    00111       GTI 151
    01000       LEI 152
    01001       LTUI    153
    01010       GEUI    154
    01011       GTUI    155
    01100       LEUI    156
    01101       SMODI   157
    01110       SDIVI   158
    01111       UMODI   159
    10000       UDIVI   160
    10001       STBI    161
    10010       STHI    162
    10011       STWI    163
    10100       LDSBI   164
    10101       LDUBI   165
    10110       LDSHI   166
    10111       LDUHI   167
    11000       LDWI    168
    11001       LDSHAX  169
    11010       LDUHAX  170
    11011       LDWAX   171
    11100       STHAX   172
    11101       STWAX   173

11001000000, op:3, r1:5, rd:5, imm:24
    000         JFNE    576
    001         JFEQ    577
    010         JFLT    578
    011         JFGE    579
    100         JFLTU   580
    101         JFGEU   581

11001000110, op:3, r1:5, r2:5, imm:16
    000         JNE 120
    001         JEQ 121
    010         JLT 122
    011         JGE 123
    100         JLTU    124
    101         JGEU    125

11001000111000, r1:5, r2:5, s:32, o:32
                COPY    112

11001001000, op:3, r1:5, r2:5, imm:8
    000         STBC    128
    001         STHC    129
    010         STWC    130
    011         LDSBC   131
    100         LDUBC   132
    101         LDSHC   133
    110         LDUHC   134
    111         LDWC    135

11001100000000000, op:5, r1:5, rd:5, imm1:32, imm2:32
    00000       ADDANDI2    200
    00001       ADDMULI2    201
    00010       ADDORI2 202
    00011       ADDXORI2    203
    00100       MULADDI2    204
    00101       MULANDI2    205
    00110       MULORI2 206
    00111       MULXORI2    207
    01000       RSUBANDI2   208
    01001       RSUBORI2    209
    01010       RSUBXORI2   210
    01011       ORADDI2 211
    01100       ORMULI2 212
```

```
11001100000000010000, op:5, r1:5, imm1:5, rd:5, imm2:32
    00000       SLLADDI2    232
    00001       SLLANDI2    233
    00010       SLLORI2 234
    00011       SLLRSUBI2   235
    00100       ANDSLLI2    236

11001100000000010001, op:5, r1:5, imm1:5, rd:5, imm2:32, imm3:32
    00000       LPAI3   392

110011000000001, op:7, r1:5, rd:5, imm1:32, imm2:32, imm3:32
    0000000     MAMI3   264
    0000001     MPMI3   265
    0000010     MOMI3   266
    0000011     MPAI3   267
    0000100     MPOI3   268
    0000101     RORI3   269
    0000110     AMPI3   270

110011000000010, op:7, r1:5, rd:5, imm1:32, imm2:32, imm3:32, imm4:32
    0000000     MPMPI4  424
    0000001     MPOMI4  425

1101, op:4, r1:5, imm:11, imm:16
    0000         JNEC   1
84
    0001         JEQC   185
    0010         JLTC   186
    0011         JGEC   187
    0100         JGTC   188
    0101         JLEC   189
    0110         JLTUC  190
    0111         JGEUC  191
    1000         JLEUC  192
    1001         JGTUC  193
    1010         JWNEC  194
    1011         JWEQC  195

11100000, uimm:8 ENTERC 5

1110001, op:1, uimm:16
    0            ENTER      6
    1            SYSCALL    7
```

Annex C (normative): Standard C library routines

C.1 Introduction

This annex details a set of standard C99 library routines [i.1] that shall be available for use by the **ECI Client**. For each function, the details of the operands passed by the **ECI Client** are defined and the return value.

Note that "string" means a sequence of non-zero bytes terminated by a zero byte.

The functions detailed below are shown as standard C library calls. In all cases, the first parameter will go into R2 (as R1 will contain the function ID, clibfunc). The declaration will assume all values are passed as scalar values or pointers to non-scalar values. If a library function calls for a non-scalar parameter to be passed by value, then the SYSCALL will pass it by reference and the VM will be required to convert the parameter as required by the library.

Return values are always scalar values or pointers returned in R1.

NOTE: The value selected for clibfunc is made up as follows:

((clibfunc >> 8) & 0x000000FF) =	The sub-chapter number of the C standard chapter dealing with library functions, coded as binary coded decimal. (For C99, the chapter is 7 and the <string.h> library is in sub-chapter 21.)
((clibfunc >> 4) & 0x0000000F) =	The function type in the library - the number following the sub-chapter number.
(clibfunc & 0x0000000F) =	The function number of a particular type in the library - the number following the function type number.

For example, memmove() is described under the heading 7.21.2.2 in [i.1]. Therefore, clibfunc is coded as 0x00002122.

C.2 memmove

clibfunc:	0x00002122
Description:	Copy n bytes from the memory pointed by s2 into the memory pointer by s1. Memory may overlap.
Declaration:	uint8 * memmove(uint8 * s1, uint8 * s2, uint32 n)
Returns:	s1

C.3 strcpy

clibfunc:	0x00002123
Description:	Copy the string (including terminating character) pointed by s2 into the memory pointed by s1. Results are undefined if memory areas overlap.
Declaration:	uint8 * strcpy(uint8 * s1, uint8 * s2)
Returns:	s1

C.4 strncpy

clibfunc:	0x00002124
Description:	As for strcpy(), but at most n bytes are copied. If the length of s2 is greater than n, then a null byte will be appended (at s1[n]).
Declaration:	uint8 * strncpy(uint8 * s1, uint8 * s2, uint32 n)
Returns:	s1

C.5 strcat

clibfunc:	0x00002131
Description:	Append a copy of the string pointed by s2 (including terminating character) at the end of the string pointed by s1. Results are undefined if memory areas overlap.
Declaration:	uint8 * strcat(uint8 * s1, uint8 * s2)
Returns:	s1

C.6 strncat

clibfunc:	0x00002132
Description:	Append a copy of the string pointed by s2 (including terminating character) at the end of the string pointed by s1, but at most n bytes are copied. If the length of s2 is greater than n, then a null byte will be appended (at n+1 bytes after the last non-null byte of the original s1). Results are undefined if memory areas overlap.
Declaration:	uint8 * strncat(uint8 * s1, uint8 * s2, uint32 n)
Returns:	s1

C.7 memcmp

clibfunc:	0x00002141
Description:	Compare the first n bytes pointed by s1 with the first n bytes pointed by s2.
Declaration:	uint32 memcmp(uint8 *s1, uint8 *s2, uint32 n)
Returns:	R1==0 if they all match, , otherwise R1 depends on the first position from the left for which values do not match. R1>0 if the byte of s1 at that position is greater than the byte of s2. R1<0 if the byte of s1 at that position is greater than the byte of s2.

C.8 strcmp

clibfunc:	0x00002142
Description:	Compare the strings pointed to by s1 and s2.
Declaration:	uint32 strcmp(uint8 * s1, uint8 * s2)

Returns:	R1==0 if they match, otherwise R1 depends on the first position from the left for which values do not match. R1>0 if the byte of s1 at that position is greater than the byte of s2. R1<0 if the byte of s1 at that position is greater than the byte of s2.

C.9 strncmp

clibfunc:	0x00002144
Description:	Compare the strings pointed to by s1 and s2, but only up to n bytes.
Declaration:	uint32 strncmp(uint8 * s1, uint8 * s2, uint32 n)
Returns:	R1==0 if they match, otherwise R1 depends on the first position from the left for which values do not match. R1>0 if the byte of s1 at that position is greater than the byte of s2. R1<0 if the byte of s1 at that position is greater than the byte of s2.

C.10 memchr

clibfunc:	0x00002151
Description:	Find the first occurrence of the byte in c within the n bytes pointed to by s.
Declaration:	uint8 * memchr(uint8 *s, uint8 c, uint32 n)
Returns:	A pointer to the located byte, or 0 if no byte was found.

C.11 strchr

clibfunc:	0x00002152
Description:	Find the first occurrence of the byte in c within the string pointed to by s, up to and including the terminating (null) byte.
Declaration:	uint8 * strchr(uint8 * s, uint8 c)
Returns:	A pointer to the located byte, or 0 if no byte was found.

C.12 strcspn

clibfunc:	0x00002153
Description:	Compute the length of the maximum initial segment of the string pointed by s1 which consists entirely of bytes not belonging to the string pointed by s2.
Declaration:	uint32 strcspn(uint8 * s1, uint8 * s2)
Returns:	The length computed.

C.13 strpbrk

clibfunc:	0x00002154
Description:	Find the first occurrence in the string pointed by s1 of any byte in the string pointed by s2.
Declaration:	uint32 strpbrk(uint8 * s1, uint8 * s2)
Returns:	The location of the first byte fulfilling the condition, or 0 if no such bytes are found.

C.14 strrchr

clibfunc:	0x00002155
Description:	Find the last occurrence of the byte in c within the string pointed to by s, up to and including the terminating (null) byte.
Declaration:	uint8 * strrchr(uint8 * s, uint8 c)
Returns:	A pointer to the located byte, or 0 if no byte was found.

C.15 strspn

clibfunc:	0x00002156
Description:	Compute the length of the maximum initial segment of the string pointed by s1 which consists entirely of bytes belonging to the string pointed by s2.
Declaration:	uint32 strspn(uint8 * s1, uint8 * s2)
Returns:	The computed value.

C.16 strstr

clibfunc:	0x00002157
Description:	Find the first occurrence of the string pointed by s1 (terminating byte excluded) in the string pointed by s2.
Declaration:	uint8 strstr(uint8 * s1, uint8 * s2)
Returns:	A pointer to the located position, or 0 if it was not found.

C.17 memset

clibfunc:	0x00002161
Description:	Copy the least significant byte of c into the memory pointed by s n times.
Declaration:	uint8 * memset(uint8 * s, uint8 c, uint32 n)
Returns:	s

Annex D (normative): ECI Client File Format

The **ECI Client** image file shall conform to ELF object file format specification [2]. This annex describes the specific information necessary to comply with the VM specification. Since the VM supports 32-bit architecture and little-endian, ELF file identification in *e_ident* [2] shall use the values in Table D.1.

Table D.1: ECI-compliant *e_ident* settings

Name	Value
e_ident[EI_CLASS]	ELFCLASS32
e_ident[EI_DATA]	ELFDATA2LSB

Table D.2 lists values that shall be used for some ELF header members.

Table D.2: ECI-compliant settings for ELF header members

Name	Value
e_type	ET_EXEC
e_machine	ET_NONE
e_version	EV_CURRENT

The loader shall reject any **ECI Client** image file with values that are different from the ones presented in this annex.

Annex E (informative): Authors & contributors

The following people have contributed to the present document:

Rapporteur:
dr. Dmitri Jarnikov, Irdeto

Other contributors:
Marnix Vlot, Vodafone Kabel Deutschland

Annex F (informative): Change History

Date	Version	Information about changes
2016-10-20	V0.0.1	First draft
2016-11-01	V0.0.2	Updated after WEBECI10
2016-11-15	V0.0.3	Updated after ECI13
2016-12-01	V0.0.4	Updated after additional reviews
2016-12-01	V0.0.5	Update for approval
2016-12-15	V0.0.6	Updated after WEBECI11. Text is approved
2017-01-23	V0.0.7	Editorial changes based on multiple reviews

History

Document history		
V1.1.1	July 2017	Publication