INTERNATIONAL STANDARD

ISO/IEC 13522-3

First edition
1997-05-15

Information technology — Coding of multimedia and hypermedia information —

Part 3:
MHEG script interchange representation

Technologies de l'information — Codage de l'information multimédia et hypermédia —

Partie 3: Représentation d'interéchange script MHEG

Reference number
ISO/IEC 13522-3:1997(E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialised system for worldwide standardisation. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organisation to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organisations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13522-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 13522 consists of the following parts, under the general title *Information technology - Coding of multimedia and hypermedia information*:

- *Part 1: MHEG object representation — Base notation (ASN.1)*
- *Part 3: MHEG script interchange representation*
- *Part 4: MHEG registration procedure*
- *Part 5: Support for base-level interactive applications*
- *Part 6: Support for enhanced interactive applications*

Annexes A to G form an integral part of this part of ISO/IEC 13522. Annexes H to K are for information only.

This page intentionally left blank

Information technology — Coding of multimedia and hypermedia information —

Part 3:
MHEG script interchange representation

1 Scope

1.1 Context of the scope

ISO/IEC 13522 specifies the coded representation of multimedia/hypermedia information objects (MHEG objects) for interchange as final form units within or across services and applications, by any means of interchange including local area networks, wide area telecommunication or broadcast networks, storage media, etc.

MHEG objects are usually produced by computer tools taking as a source form multimedia applications designed using multimedia scripting languages. In this context, one of the MHEG object classes, the script class, is intended to complement the other MHEG classes in expressing the functionality commonly supported by scripting languages. Script objects express more powerful control mechanisms and describe more complex relationships among MHEG objects than can be expressed by MHEG action and link objects alone. Furthermore, script objects express access and interaction with external services provided by the run-time environment.

Other parts of ISO/IEC 13522 define the coded representation for script objects in an open manner so that script objects may encapsulate either standardised or proprietary script code. Script objects encapsulate scripts that may be encoded in any encoding format as registered according to ISO/IEC 13522-4.

1.2 Scope of this part of ISO/IEC 13522

The scope of this part of ISO/IEC 13522 is to extend the coded representation of the MHEG script object class defined by another part of ISO/IEC 13522, including ISO/IEC 13522-1 and ISO/IEC 13522-5.

This part of ISO/IEC 13522 specifies the MHEG script interchange representation (MHEG-SIR) for the contents of script objects, i.e. the encoding of the script data component of the MHEG script class.

MHEG engines are system or application components that handle, interpret and present MHEG objects. This part of ISO/IEC 13522 also specifies the semantics of interchanged scripts. These semantics are defined in terms of minimum requirements on the behaviour of MHEG engines that support the interpretation of interchanged scripts.

This part of ISO/IEC 13522 is applicable to all applications that interchange multimedia and hypermedia information.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 13522. At the time of publication, the editions indicated were valid. All standards are subject

to revision, and parties to agreements based on this part of ISO/IEC 13522 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of ISO and IEC maintain registers of currently valid International Standards.

[1] ISO/IEC 8824-1:1995|ITU-T Recommendation X.680 (1994): *Information technology — Abstract Notation One (ASN.1): Specification of basic notation.*

[2] ISO/IEC 8825-1:1995|ITU-T Recommendation X.690 (1994): *Information technology — ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

[3] ISO/IEC 9646:1992-1995, *Information technology — Open Systems Interconnection — Conformance testing methodology and framework* (all parts).

[4] ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

[5] ISO/IEC 13522-1:1997, *Information technology — Coding of multimedia and hypermedia information — Part 1: MHEG object representation — Base notation (ASN.1).*

[6] ISO/IEC 13522-4:1996, *Information technology — Coding of multimedia and hypermedia information — Part 4: MHEG registration procedure.*

[7] ISO/IEC 13522-5:1997, *Information technology — Coding of multimedia and hypermedia information — Part 5: Support for base-level interactive applications.*

[8] ISO/IEC 14750:—[1)], *Information technology — Open Distributed Processing — Interface Definition Language.*

[9] IEEE 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

3 Definitions

For the purposes of this part of ISO/IEC 13522, the definitions given in ISO/IEC 8824-1 [1], ISO/IEC 8825-1 [2] and the following definitions apply.

3.1 application programming interface (API): Boundary across which a software application uses facilities of programming languages to invoke software services. These facilities may include procedures or operations, shared data objects and resolution of identifiers.

3.2 attribute: (1) MHEG attribute (see ISO/IEC 13522-1 [5]);
(2) IDL attribute (q.v.).

3.3 conforming MHEG-3 engine: MHEG-3 engine whose implementation conforms to the provisions of this part of ISO/IEC 13522.

3.4 conforming MHEG-3 interchanged script: Interchanged script that conforms to the provisions of this part of ISO/IEC 13522.

3.5 conforming MHEG-3 object: MHEG script object whose coded representation conforms to the provisions of this part of ISO/IEC 13522.

1) To be published.

3.6 frame: Record of elements on the call stack that define an execution context; one such record is pushed onto the call stack everytime a routine is called, to memorize the current execution context; one is popped from the call stack when the routine is returned from, to restore the execution context at the time of calling.

3.7 hypermedia (adj.): Featuring access monomedia and multimedia information by interaction with explicit links.

3.8 interchanged script: The coded representation of the "script data" attribute of an MHEG script object.

3.9 interface definition language (IDL): Formal notation that is used to specify types and objects through the definition of the interface that they provide, as defined by ISO/IEC 14750-1 [8].

3.10 IDL attribute: Named, typed association between an object and a value; it is declared as part of an IDL interface; it is made visible to clients as a pair of operations: an accessor (get) and a modifier (set); if it is read-only, it only provides an accessor.

3.11 IDL exception: Message that can be raised when an exceptional condition occurs during the performance of the request to an IDL operation; it is defined in an IDL module and may have members, which are returned to the caller together with the message identifier.

3.12 IDL instance: Object that provides the operations, signatures and semantics specified by an IDL interface; its creation and management is implementation-specific.

3.13 IDL interface: Description, using IDL, of a set of operations that a client may request of an IDL object.

3.14 IDL object: Identifiable, encapsulated entity that provides one or more services which can be requested by a client.

3.15 IDL operation: Service that can be requested and is provided by an IDL object; it is defined within an IDL interface by a name, a signature which defines the type of its parameters and return value, and the list of exceptions that its invocation may raise.

3.16 mh-script: Internal representation, within an MHEG engine, of an "available" MHEG script object.

3.17 MHEG action: Operation that applies to MHEG objects and consists of sequential and/or parallel combinations of MHEG elementary actions.

3.18 MHEG action object: MHEG object that describes MHEG actions.

3.19 MHEG application: Application that involves the interchange of MHEG objects within itself or with another application.

3.20 MHEG conforming object: Information object whose coded representation conforms to the provisions of another part of ISO/IEC 13522.

3.21 MHEG elementary action: One of the basic operations applying to MHEG objects; it maps one MHEG API primitive.

3.22 MHEG engine: Process or set of processes able to interpret MHEG objects.

3.23 MHEG entity: Any MHEG object, rt-object, content data, script data, socket, channel or other construction defined by ISO/IEC 13522.

3.24 MHEG link: MHEG object that defines spatio-temporal relationships among MHEG objects expressed in terms of trigger conditions and actions.

3.25 MHEG object: Coded representation of an instance of an MHEG object class.

3.26 MHEG script class: MHEG class defining a structure to interchange script data in a specified encoded form.

3.27 MHEG script object: The coded representation of an instance of an MHEG script class.

3.28 MHEG API: The API provided by an MHEG engine to MHEG applications for the manipulation of MHEG objects.

3.29 MHEG-3 (adj.): Applies to entities that conform to the provisions of this part of ISO/IEC 13522.

3.30 MHEG-3 application: MHEG application that interchanges scripts within itself and/or with other applications as the "script data" component of MHEG script objects, according to the representation specified by this part of ISO/IEC 13522.

3.31 MHEG-3 engine: MHEG engine that processes and interprets MHEG-SIR interchanged scripts.

3.32 MHEG-3 profile: Profile of this part of ISO/IEC 13522.

3.33 MHEG-SIR: (1) The script interchange representation defined by this part of ISO/IEC 13522; (2) (adj.) Applies to an entity defined as part of this Script Interchange Representation.

3.34 (MHEG-SIR) call stack: Stack that is associated with each running rt-script by the MHEG-SIR virtual machine and that contains a call frame for each active function invocation.

3.35 MHEG-SIR code: Encoded sequence of MHEG-SIR instructions.

3.36 (MHEG-SIR) constant: Static, typed, named value which is declared within an interchanged script and whose value is globally accessible and unchanged throughout the execution of the script.

3.37 (MHEG-SIR) constructed type: Type described as a combination of other types using one of the following constructors: sequence, string, array, union, structure.

3.38 (MHEG-SIR) data identifier: Integer that uniquely identifies the name of a data element of an interchanged script (constant, global variable, dynamic variable, local variable).

3.39 (MHEG-SIR) exception: Message triggered during the invocation of a service.

3.40 (MHEG-SIR) function: Named code sequence whose execution may be invoked by an interchanged script; it may be a routine, a predefined function or a service.

3.41 (MHEG-SIR) function identifier: Integer that uniquely identifies a function within an interchanged script.

3.42 (MHEG-SIR) global variable: Variable with global scope.

3.43 (MHEG-SIR) instruction: Elementary unit of code of an MHEG-SIR interchanged script; it consists of an op-code followed by zero or more operands.

3.44 (MHEG-SIR) instruction execution unit: Within an MHEG-SIR script interpreter, virtual processing unit that executes an MHEG-SIR instruction.

3.45 MHEG-SIR interchanged script: Interchanged script coded according to MHEG-SIR.

3.46 (MHEG-SIR) local variable: Variable with local scope within a routine.

3.47 (MHEG-SIR) message: Event that may be received by the script interpreter during the execution of the script; it may be either predefined (MHEG API exception, MHEG-3 API operation and exception, internal exception) or declared within an interchanged script (exception provided by an external interface).

3.48 (MHEG-SIR) message identifier: Integer that uniquely identifies a message within an interchanged script.

3.49 (MHEG-SIR) message queue: Queue that is associated with each running rt-script by the MHEG-SIR virtual machine and that contains the messages targeted at the rt-script.

3.50 (MHEG-SIR) object reference: MHEG-SIR value that represents an IDL instance and that is passed as the parameter of an external call to request a service from this instance.

3.51 (MHEG-SIR) operand: Parameter of an instruction; it is encoded next to the instruction's op-code.

3.52 (MHEG-SIR) package: Set of external functions that are provided by a module of the run-time environment and that are accessible to an rt-script and declared within an interchanged script; it is composed of services and exceptions.

3.53 (MHEG-SIR) parameter: Piece of data exchanged with a function call, a message or an instruction.

3.54 (MHEG-SIR) parameter stack: Stack that is associated with each running rt-script by the MHEG-SIR virtual machine and that is used to provide parameters to and retrieve results of instructions.

3.55 (MHEG-SIR) predefined type: A type whose description and identifier are predefined by this part of ISO/IEC 13522 and thus need not be declared within interchanged scripts; it may be either a primitive type or a constructed type.

3.56 (MHEG-SIR) primitive type: Basic predefined type, as opposed to constructed type.

3.57 (MHEG-SIR) routine: Function that is declared within an interchanged script together with the virtual machine code that defines its semantics.

3.58 (MHEG-SIR) rt-script execution unit: Within an MHEG-SIR script interpreter, virtual processing unit that executes script code.

3.59 (MHEG-SIR) script interpreter: The part of an MHEG-3 engine, that handles and interprets interchanged scripts.

3.60 (MHEG-SIR) service: External function that is declared within an interchanged script and whose implementation is made accessible to an rt-script by the run-time environment on the execution platform.

3.61 (MHEG-SIR) variable: Within the MHEG-SIR virtual machine, named, typed memory unit whose value may be changed at any time when its scope is active and whose most recent value may be read.

3.62 (MHEG-SIR) virtual machine: Abstract description of the memory units and instruction execution engine of an MHEG-SIR script interpreter.

3.63 multimedia (adj.): That handles several types of representation media.

3.64 multimedia and hypermedia application: Application that features presentation of multimedia information to the user and interactive navigation across this information by the user.

3.65 multimedia application: Application that features presentation of multimedia information to the user.

3.66 platform mapping specification: Specification of how MHEG-3 engine implementations shall map IDL specifications to run-time environment components on one type of platform.

3.67 queue: Collection of elements that are inserted and removed in first-in first-out (FIFO) order.

3.68 rt-script: Run-time instance (or copy) of an mh-script, created by an MHEG engine.

3.69 scope: Context of reference for a variable; if it is global, the variable may be referenced by any script instruction; if it is local, the variable may only be referenced in the local execution context.

3.70 scripting language: Programming language intended for easy and rapid design of applications by non-professional programmers.

3.71 script interchange representation (SIR): Coded representation used by an application to interchange scripts for the purpose of implementing dynamic behaviour.

3.72 stack: Collection of elements that are inserted (pushed) and removed (popped) in last-in first-out (LIFO) order.

4 Abbreviations

For the purposes of this part of ISO/IEC 13522, the following abbreviations apply.

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CORBA	Common Object Request Broker Architecture
CS	Call Stack
CT	Constant Table
DER	Distinguished Encoding Rules
DID	Data IDentifier
DT	Data Table
EBNF	Extended Backus-Naur Form
ER	Error Register
ETR	ETSI Technical Report
FID	Function IDentifier
FIFO	First In First Out
FP	Frame Pointer
FR	Function Register
GT	Global variable definition Table
HT	Handler Definition Table
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IP	Instruction Pointer
IR	Instruction Register
ISO	International Organisation For Standardisation
ITU-T	International Telecommunication Union, Telecommunication standardisation sector
JTC	Joint Technical Committee
LIFO	Last In First Out
LT	Local variable Table
MHEG	Multimedia and Hypermedia information coding Experts Group
MID	Message IDentifier
MPEG/DSM-CC	Moving Picture Experts Group - Digital Storage Media Command and Control
MQ	Message Queue
PID	Package IDentifier
PS	Parameter Stack
PT	Package definition Table

QP	Queue Pointer
rt	Run-time
RT	Routine definition Table
SIR	Script Interchange Representation
SP	Stack Pointer
ST	Service definition Table
TID	Type IDentifier
TLV	Type-Length-Value
TT	Type definition Table
VT	Variable Table
XT	eXception definition Table

5 Conformance

This part of ISO/IEC 13522 defines conformance requirements

- on information objects, i.e. MHEG script objects;
- on implementations, i.e. MHEG engine implementations.

5.1 Information object conformance

A conforming MHEG-3 script object shall meet all of the following criteria:

1) its coded representation shall conform to the provisions of another part of ISO/IEC 13522;
2) its coded representation shall encapsulate a conforming MHEG-3 interchanged script.

The information object conformance is evaluated on the information objects that are interchanged in the purpose of their execution on a terminal.

5.1.1 Profiles

This part of ISO/IEC 13522 defines no profiles.

NOTE 1: However, MHEG-3 profiles may be defined by other standards or by other parts of ISO/IEC 13522. In accordance with the profile definition framework, standardised MHEG-3 profiles should be at least as constraining; information objects claiming conformance to such profiles should at least conform to this part of ISO/IEC 13522.

An MHEG-3 profile should define all of the following:

- a profile of the MHEG-SIR virtual machine defined by this part of ISO/IEC 13522;
- a profile of IDL, together with its mapping to MHEG-SIR, for the expression of interface between scripts and the external environment;
- an API for the manipulation of MHEG objects defined by another part of ISO/IEC 13522, together with a mapping of this interface to MHEG-SIR.

NOTE 2: According to ISO recommendations, MHEG-3 profiles should ensure upward compatibility of the ASN.1 encoding, so that interchanged scripts conforming to an MHEG-3 profile also conform to this part of ISO/IEC 13522.

5.1.2 Encoding

A conforming MHEG-3 interchanged script shall be encoded according to the encoding rules defined by Annex B.

5.1.3 Syntax

A conforming MHEG-3 interchanged script shall conform to the ASN.1 syntax defined by Annex A.

5.1.4 Semantics

A conforming MHEG-3 interchanged script shall only include semantically valid declarations and instruction sequences as defined by Clauses 12 and 13.

5.2 Implementation conformance

An implementation of this part of ISO/IEC 13522 is an MHEG-3 engine.

A conforming MHEG-3 engine shall support the interpretation of conforming MHEG-3 script objects.

This part of ISO/IEC 13522 defines the semantics of MHEG-3 interchanged scripts. This implies conformance requirements not on information objects, but on the behaviour of MHEG-3 engines.

NOTE 1: Although a conforming script might not realise the semantics implied by its designer, the way conforming engines behave in interpreting this script is predictable.

NOTE 2: This part of ISO/IEC 13522 does not consider conformance for a system, an engine or a process as far as it is not related to the interpretation of interchanged scripts.

5.2.1 Conformance requirements

A conforming MHEG-3 engine shall meet all of the following criteria:

1) it shall parse and interpret conforming MHEG-3 interchanged scripts according to the virtual machine behaviour defined in this part of ISO/IEC 13522 (see Clause 9);

2) it shall support communication with the run-time environment and with MHEG objects according to the IDL mapping behaviour defined in this part of ISO/IEC 13522 (see Clauses 10, 11 and 14);

3) it shall provide the MHEG-3 API defined in this part of ISO/IEC 13522 (see Clause 15 and Annex F);

4) for the purpose of manipulation of MHEG objects by interchanged scripts, it shall support an MHEG API and its mapping according to the framework defined in this part of ISO/IEC 13522 (see Annex E);

5) for the purpose of communication with the run-time environment, it shall support a platform mapping specification according to the framework defined in this part of ISO/IEC 13522 (see Annex D);

6) it may provide additional functions or facilities not required by this part of ISO/IEC 13522 or by the platform mapping specification. Each such non-standard extension shall be identified as such in the system documentation.

5.2.2 Conformance documentation

A conformance document with the following information shall be available for an implementation claiming conformance to this part of ISO/IEC 13522. The conformance document shall meet all of the following criteria:

1) it shall list all the mandatory features required by this part of ISO/IEC 13522, with reference to the appropriate Clauses and subclauses;

2) it shall either include the platform mapping specification to which the implementation conforms or reference a registered platform mapping specification in an unambiguous way;

3) it shall contain a statement that indicates the full names, numbers, and dates of the standards that apply;

4) it shall state which of the optional features defined in this part of ISO/IEC 13522 and in the platform mapping specification are supported by the implementation;

5) it shall describe the behaviour of the implementation for all implementation-defined features defined in this part of ISO/IEC 13522 and in the platform mapping specification. This requirement shall be met by listing these features and by providing either a specific reference to the system documentation or full syntax and semantics of these features. The conformance document may specify the behaviour of the implementation for those features where this part of ISO/IEC 13522 or the platform mapping specification states that implementations may vary or where features are identified as undefined or unspecified.

No specifications other than those specified by this part of ISO/IEC 13522 and the platform mapping specification shall be present in the conformance document.

5.3 Application conformance

An application of this part of ISO/IEC 13522 (called MHEG-3 application) is an MHEG application that interchanges scripts within itself and/or with other applications as the "script data" component of MHEG script objects according to the encoded representation specified by this part of ISO/IEC 13522.

5.4 Test Methods

Any measurement of conformance to this part of ISO/IEC 13522 shall be performed using test methods that conform to ISO/IEC 9646 [3].

6 Overview

This part of ISO/IEC 13522 extends the provisions of other parts of ISO/IEC 13522 so that MHEG objects and applications support functionality of multimedia scripting languages in a standard way. Considering the functionality supported by other parts of ISO/IEC 13522, these extensions are divided in two main topics:

- data processing operations (see subclause 6.2);
- access to external data and functions (see subclause 6.3).

For the support of both topics, this part of ISO/IEC 13522 specifies

- complete and detailed provisions for the encoding of interchanged scripts;
- the required behaviour of a script interpreter.

6.1 Description methodology

For the description of these provisions, this International Standard|Recommendation follows a methodology that considers four description levels:

- level a): informal text description;
- level b): precise description of semantics;
- level c): formal description of syntax;
- level d): formal description of encoding.

These levels are used in the following Clauses as follows:

- level a): Clauses 8 to 11;
- level b): Clauses 12 to 15;
- level c): Annexes A, E, F, G;
- level d): Annexes B, C.

NOTE: Informative Annexes H and J also use level c) description.

6.2 Data processing operations

To deal with data processing operations, MHEG-SIR defines the structure of interchanged scripts that consist of data declarations and function declarations, the latter encapsulating sequences of instructions.

Clause 8 defines the elements of the MHEG-SIR virtual machine code.

Clause 9 specifies the MHEG-SIR virtual machine, i.e. a model of how MHEG-SIR script interpreters shall perform interpretation of MHEG-SIR script code. This virtual machine is used afterwards to describe the semantics of MHEG-SIR instructions. Clause 9 states requirements on the functionality that script interpreters shall provide; however, it does not specify how to implement this functionality.

Clause 12 defines the declarations of MHEG-SIR interchanged scripts. It specifies their structure, i.e. the way they shall be represented, and their semantics, i.e. the way they shall be interpreted by MHEG-SIR script interpreters. The semantics are specified using the virtual machine formalism introduced in Clause 9.

Clause 13 defines the MHEG-SIR instructions. It specifies their structure, i.e. the way they shall be represented, and their semantics, i.e. the way they shall be interpreted by MHEG-SIR script interpreters. These semantics are specified using the virtual machine formalism introduced in Clause 9.

Annex A formally defines the precise syntax of interchanged scripts using the ASN.1 notation.

Annex B formally defines the encoding of interchanged scripts.

Annex C lists the predefined elements of MHEG-SIR and defines their encoding.

Annex G formally defines the instantiation of this part of ISO/IEC 13522 to ISO/IEC 13522-1 and ISO/IEC 13522-5, i.e. the MHEG objects in these parts to which MHEG-SIR applies, and the way it applies to them.

6.3 Access to external data and functions

To deal with access to external data and functions, MHEG-SIR uses IDL to describe interfaces in an abstract, language-independent way and thus unify the way external data and functions are viewed by script interpreters.

In the MHEG-SIR context, IDL is used to separate clearly the way (MHEG-SIR specific) the use of external data or functions is expressed by interchanged scripts from the way (at least platform-dependent, and maybe application-dependent) these data or functions are provided by the external environment. MHEG-SIR thus defines how the interfaces are used, while the application is responsible for defining how they are provided.

To allow script interpreters to manipulate MHEG entities and exchange information with them, MHEG-3 engines provide script interpreters with access to the MHEG entities (data) and invocation of the MHEG actions (functions) through an MHEG API defined using IDL. The MHEG types and actions are predefined in MHEG-SIR to achieve compact coding and efficient interpretation of MHEG object manipulation.

To allow script interpreters to co-operate with the run-time environment, the run-time environment provides access to its data and functions according to a platform mapping specification of IDL. This specification describes how IDL operations need be provided on a particular platform so that MHEG-3 engines be able to use them as external services.

NOTE: Packages may be provided in the form of libraries, device drivers, operating system components, processes, telecommunication services, etc.

Clause 7 describes assumptions on the structure of MHEG-3 engines and their relationships with their environment.

Clause 10 describes the general mechanisms used to access to external data and functions provided by the run-time environment.

Clause 11 describes the general mechanisms used to manipulate MHEG objects.

Clause 14 specifies the IDL mapping for MHEG-SIR, i.e. the mechanisms used by the MHEG-SIR representation to describe IDL packages and invoke IDL operations.

Clause 15 specifies the structure and semantics of the MHEG-3 API, i.e. the set of operations that may be used to manipulate scripts.

Annex D specifies the IDL platform mapping specification form, i.e. the template for the document that need be filled in and registered for each platform type, to specify the platform-specific provisions that services provided by the run-time environment on this platform shall fulfil, and to which MHEG-3 engines shall conform so that they be able to co-operate with services provided by the run-time environment on this platform and therefore to interpret scripts that call upon such services.

Annex E specifies the framework that shall be used to define an MHEG API using IDL and the procedure that shall be followed to map it to MHEG-SIR.

Annex F defines the precise syntax of the MHEG-3 API using the IDL notation.

7 MHEG/MHEG-3 relationship

This Clause introduces general assumptions about MHEG-3 engines, which are used afterwards to describe the performance of a script interpreter and its relationships with its external environment.

MHEG-3 engines shall provide the functionality described hereafter in some way, in order to behave as expected as far as interpretation of interchanged scripts is concerned.

However, there is no requirement on MHEG-3 engines to implement this functionality as described.

NOTE: For instance, the MHEG-3 engine functional components described thereafter need not correspond to actual (e.g. software) components of MHEG-3 engine implementations.

7.1 MHEG entities

MHEG-3 engines handle MHEG entities: MHEG objects, mh-objects, rt-objects, interchanged MHEG objects, sockets, channels.

NOTE: MHEG entities are described in more detail in Annex K.

7.2 Functional entities

MHEG-3 engines may be viewed as consisting of the following functional components:

- MHEG object parser: parses interchanged MHEG objects and transforms them into mh-objects under control of the mh-object manager;
- mh-object manager: controls the life cycle and allows access to all mh-objects;
- rt-object manager: controls the life cycle and allows access to all rt-objects;

- reference resolver: transforms an MHEG reference into a usable identifier or handle;
- link handler: watches active links and triggers the corresponding actions when their conditions become true;
- action interpreter: interprets MHEG elementary actions;
- script interpreter: parses MHEG-SIR interchanged scripts and interprets rt-scripts; provides access to the run-time environment;
- presentation agent: interface with the presentation environment; orders presentation of rt-contents; receives user selections and modifications;
- access agent: interface with the communication environment; provides access to interchanged MHEG objects and to content data.

7.3 MHEG-SIR script interpreter

Within an MHEG-3 engine, the script interpreter shall be responsible for the following:

- parsing interchanged scripts (provided by the MHEG object parser)
- preparing the appropriate data structures for further execution of rt-scripts;
- executing script code;
- realising the default effect of MHEG actions targeted at mh-scripts or rt-scripts;
- invoking the appropriate handler (in the script program) for these MHEG actions;
- forwarding MHEG elementary actions invoked by the script program to the action interpreter;
- managing interchange with the run-time environment (locating and loading packages, invoking services, receiving messages, passing data) using the appropriate platform-specific communication mechanisms.

8 Elements of MHEG-SIR

This Clause describes the main elements of MHEG-SIR and how interchanged scripts shall use them.

The entities that are declared and manipulated by MHEG-SIR interchanged scripts are

- data types;
- data;
- functions;
- messages.

These concepts are defined in the following subclauses; however, the detailed structure of their declarations is specified in Clause 12.

8.1 Data types

Data types are used to describe the structure of

- the script's own data (constants and variables);
- the parameters and return values of the script's routines;
- the parameters and return values of external functions;
- the parameters of messages handled by scripts.

As scripts need adapt themselves to the signature of functions that may be provided by the external environment, MHEG-SIR defines a wide range of types corresponding to the IDL data types.

The encoding of data type definitions in an interchanged script is defined by Annex A. This part of ISO/IEC 13522 imposes no requirement on the way MHEG-3 engines represent these data types.

The MHEG-SIR uses two kinds of data types:

- predefined types (see subclause 8.1.1);
- declared types (see subclause 8.1.2).

All types may be referenced in a unique, unambiguous way by their type identifier.

8.1.1 Predefined types

Predefined types may be either primitive or constructed types.

Predefined types have predefined type identifiers and therefore need not be declared by interchanged scripts. The list of predefined types and their identifiers is given in Annex C.

8.1.1.1 Primitive types

The primitive types correspond to the IDL primitive types. This is the list of MHEG-SIR primitive types:

- `void`;
- `octet`;
- `short`;
- `long`;
- `unsigned short`;
- `unsigned long`;
- `float`;
- `double`;
- `boolean`;
- `character`;
- `data identifier`;
- `object reference`.

For easier reference, primitive types have individual letter codes as indicated by Table 1:

Table 1: Letter codes of primitive types

Type	Letter code
`octet`	O
`short`	S
`long`	L
`unsigned short`	W (as Word)
`unsigned long`	U
`float`	F
`double`	D
`boolean`	B
`character`	C
`data identifier`	I (as Identifier)
`object reference`	R (as Reference)

8.1.1.1.1 `void` type

The `void` type shall only be used to express the type of return value of a function. Functions whose type of return value is `void` do not return any data. An interchanged script shall have no constants or variables of `void` type. The `void` type shall not be used in the definition of constructed types.

8.1.1.1.2 `octet` type

Data whose type is `octet` shall take a numeric value within the range [0 .. 255]. `Octet` variables without explicit initial value shall be initialised to 0.

8.1.1.1.3 `short` type

Data whose type is `short` shall take a signed integer value within the range [-32 768 .. 32 767]. `Short` variables without explicit initial value shall be initialised to 0.

8.1.1.1.4 `long` type

Data whose type is `long` shall take a signed integer value within the range [-2 147 483 648 .. 2 147 483 647]. `Long` variables without explicit initial value shall be initialised to 0.

8.1.1.1.5 `unsigned short` type

Data whose type is `unsigned short` shall take an unsigned integer value within the range [0 .. 65 535]. `Unsigned short` variables without explicit initial value shall be initialised to 0.

8.1.1.1.6 `unsigned long` type

Data whose type is `unsigned long` shall take an unsigned integer value within the range [0 .. 4 294 967 295]. `Unsigned long` variables without explicit initial value shall be initialised to 0.

8.1.1.1.7 `float` type

Data whose type is `float` shall take a single-precision floating point value within the range specified by IEEE 754 [9]. `Float` variables without explicit initial value shall be initialised to 0.

8.1.1.1.8 `double` type

Data whose type is `double` shall take a double-precision floating point value within the range specified by IEEE 754 [9]. `Double` variables without explicit initial value shall be initialised to 0.

8.1.1.1.9 `boolean` type

Data whose type is `boolean` shall have either 'true' or 'false' as their value. `Boolean` variables without explicit initial value shall be initialised to 'false'.

8.1.1.1.10 `character` type

Data whose type is `character` shall take a character value within the `BMPString` character set as defined by the Basic Multilingual Plane of ISO/IEC 10646-1 [4]. `Character` variables without explicit initial value shall have an undefined initial value.

Conforming MHEG-3 engines may state that they only adopt a restricted set of characters, e.g. based on the standard collections of Annex A of ISO/IEC 10646-1 [4]. In this case, they shall document these adopted subsets and the level of implementation in the conformance document.

8.1.1.1.11 `data identifier` type

Data whose type is `data identifier` shall take an unsigned integer value within the range [0 .. 65 535]. This value is used to identify a constant, global variable, dynamic variable, local variable or routine parameter of the script, as defined by subclause 8.6.2 below. There shall be no constants of `data identifier` type. `Data identifier` variables without explicit initial value shall have an undefined initial value.

8.1.1.1.12 `object reference` type

Data whose type is `object reference` shall take as value a handle that references an IDL object to which services or predefined functions apply. Encoding of object references is defined by the platform mapping specification. There shall be no constants of `object reference` type. The `object reference` type shall not be used in the definition of constructed types. `Object reference` variables without explicit initial value shall have an undefined initial value.

Object references are used as the implicit first parameter of all external calls to specify the object to which the call applies. Object reference values shall be provided by the external environment, as an output parameter or return value of an **external call** (`XCALL`) instruction. The **get object reference** (`GETOR`) instruction is used to get a first object reference on the root object of a given package. The **null** object reference is used to refer to the original object of the MHEG-3 API (instance of `ScriptInterpreter`).

8.1.1.2 Predefined constructed types

To allow scripts to express manipulation of MHEG data more easily, the MHEG API data types are predefined.

Although they are not defined within interchanged scripts, predefined constructed types, like declared constructed types, can be expressed using type constructors and type identifiers, as described in subclause 8.1.2. Only predefined type identifiers shall be used to express the structure of predefined constructed types.

8.1.2 Declared constructed types

Constructed types shall be defined using one constructor and one or several type identifiers identifying either a declared or predefined type.

The constructor of a constructed type shall be one of the following:

- `sequence` (see subclause 8.1.2.1);
- `string` (see subclause 8.1.2.2);
- `array` (see subclause 8.1.2.3);
- `structure` (see subclause 8.1.2.4);
- `union` (see subclause 8.1.2.5).

Declared types are defined within interchanged scripts.

MHEG-SIR types shall not be redefined in an interchanged script. The structure of a declared type shall not match that of a predefined type or that of another declared type.

There shall not be more than 16 384 types declared in an interchanged script.

8.1.2.1 `sequence` types

`Sequence` types shall be defined by

- their size (optional);
- their element type.

The size shall be an unsigned short value. It represents the maximum number of elements of the sequence. If the type definition specifies no size, the number of elements may be any size up to the maximum. `Sequence` types with an explicit size are called bounded `sequence` types.

The maximum size of any `sequence` type is 65 535 elements.

The element type may be any primitive, constructed or predefined type except `void` and `object reference`. The element type shall be referenced using its type identifier. `Sequence` type definitions shall not lead to infinite recursion.

NOTE: As a consequence, the type identifier of the `sequence` may be nested within the type definition only below a `union` constructor.

Data whose type is a defined `sequence` type shall take as their value an ordered list of zero or more values of the element type.

Variables of a `sequence` type without explicit initial value shall be initialised to a null list (sequence of zero element).

8.1.2.2 `string` types

`String` types are semantically equivalent to `sequence` types whose element type is `character`.

NOTE: To optimise their handling, string values may be implemented in a different way than sequences of character would. Therefore, strings and sequences of character remain distinct, although semantically equivalent, types.

`String` types shall be defined by their size (optional).

The size shall be an unsigned short value. It represents the maximum number of elements of the string. If the type definition specifies no size, the number of elements may be any size up to the maximum. `String` types with an explicit size are called bounded `string` types.

The maximum size of any `string` type is 65 535 characters.

Data whose type is a defined `string` type shall take as their value a string of zero or more characters.

Variables of a `string` type without explicit initial value shall be initialised to a null string (sequence of zero character).

8.1.2.3 `array` types

`Array` types shall be defined by

- their size;
- their element type.

The size shall be an unsigned short value. It represents the exact number of elements in the array.

The element type may be any primitive, constructed or predefined type except `void` and `object reference`. The element type shall be referenced using its type identifier. `Array` type definitions shall not lead to infinite recursion.

NOTE: As a consequence, the type identifier of the `array` may be nested within the type definition only below a `union` constructor.

Data whose type is a defined `array` type shall take as their value an ordered list of values of the element type, the length of the list being specified by the size of the array.

Variables of an `array` type without explicit initial value shall be initialised to a list of elements whose initial value is determined by the element type.

8.1.2.4 `structure` types

`Structure` types shall be defined by an ordered list of 1 to 256 element types.

The element types may be any primitive, constructed or predefined type except `void` and `object reference`. The element types shall be referenced using their type identifiers. `Structure` type definitions shall not lead to infinite recursion.

NOTE: As a consequence, the type identifier of the `structure` may be nested within the type definition only below a `union` constructor.

Data whose type is a defined `structure` type shall take as their value an ordered list of values of the element type that corresponds to their rank in the type definition.

Variables of a `structure` type without explicit initial value shall be initialised to a list of elements whose initial value is determined by their element type.

8.1.2.5 `union` types

`Union` types shall be defined by an ordered list of element types.

There shall not be more than 256 choices (element types) in a `union` type.

The element types may be any primitive, constructed or predefined type except `void` and `object reference`. The element types shall be referenced using their type identifiers.

Data whose type is a defined `union` type shall take as their value

- an integer which represents the index (starting at 0) in the choice list;
- a value of the element type whose rank in the type definition is the above index.

Variables of a `union` type without explicit initial value shall have an undefined initial value.

8.2 Data

The MHEG-SIR defines three kinds of data:

- immediate values (see subclause 8.2.1);
- constants (see subclause 8.2.2);
- variables (see subclause 8.2.3).

All data used by an interchanged script are of a definite data type, either predefined or declared.

Two data values shall be equal if and only if

- they are of the same type, i.e. they have the same type identifier;
- if they are of a primitive type then they are identical;
- if they are of a `structure`, `sequence` or `array` type then every element of one list is equal to the element of the same rank in the other list;
- if they are of a `union` type then their tags are identical and their values are equal to each other.

As a consequence,

- values of a `string` type shall not be compared with values of a `sequence` of `character` type;

- values of a bounded `sequence` type shall not be compared with values of another bounded `sequence` type or with values of an unbounded `sequence` type, since they have different type identifiers;
- values of a bounded `string` type shall not be compared with values of another bounded `string` type or with values of an unbounded `string` type, since they have different type identifiers.

All variables and constants are referenced in a unique, unambiguous way by their data identifier.

8.2.1 Immediate values

Immediate values are data that are not declared within the interchanged script, and may therefore only be used "immediately", i.e. as they are encountered. An immediate value may be encountered in an interchanged script

- as a constant value;
- as the initial value of a variable;
- as the operand of a **push immediate** (`PUSHI`) instruction.

Besides, immediate values are used in the course of the script execution through the parameter stack, as parameters for instructions or functions.

Unless the context restricts it otherwise, immediate values may be of any type except `void`.

The encoding of data values in an interchanged script is defined by Annex A. This part of ISO/IEC 13522 imposes no requirement on the way MHEG-3 engines represent data values of a particular type.

8.2.2 Constants

Constants shall be declared within the interchanged script and defined by

- a data type;
- a value of this data type.

Constants may be of any type except

- `object reference`;
- `data identifier`;
- `void`.

Constants have a global scope and may be referenced using their data identifier throughout the interchanged script.

There shall not be more than 4 096 constants declared in an interchanged script.

8.2.3 Variables

Variables shall be declared within the interchanged script and defined by

- a data type;
- optionally, a value of this data type, to be taken as the initial value for this constant.

Variables may be of any type except `void`.

Variables are referenced using their data identifier. A reference to a variable may be used with either of the following semantics:

- "right-hand" semantics: the same as if the value of this variable was provided instead;
- "left-hand" semantics: states that this variable has to be assigned a data value.

In the latter case, the value to be assigned to the variable may be an immediate value (including a computed value), the value of a constant or the value of a variable (including the future value of a function's output parameter).

The MHEG-SIR defines three kinds of variables:

- global variables;
- local variables;
- dynamic variables.

8.2.3.1 Global variables

Global variables have a global scope which covers the entire interchanged script. They may be referenced using their data identifier from any routine or variable. They may be assigned a new value at any time during execution of the rt-script.

There shall not be more than 28 672 global variables declared in an interchanged script.

8.2.3.2 Local variables

Local variables have a lexical scope which is restricted to the execution of the code of the routine to which they belong. They may be referenced using their data identifier only within the code of this routine.

There are two kinds of local variables:

- local variables that are declared within the routine declaration as part of the local variable declaration;
- actual parameters of the routine, whether passed by value or by reference, which are declared within the routine declaration as part of the routine signature.

There shall not be more than 256 local variables declared in each routine of an interchanged script.

8.2.3.3 Dynamic variables

Dynamic variables have a dynamic scope which extends from the time when they are created using an **allocate** (ALLOC) instruction up to the time when they are released using a **free** (FREE) instruction. At creation, they are given a data identifier by the script interpreter. They may be referenced using their data identifier at any time during the execution of the script. However, as the data identifier of a dynamic variable is only known at run-time, it can only be used as a parameter stack or a variable value, not as the operand of an instruction.

There shall not be more than 32 512 dynamic variables used at a given time during execution of an rt-script.

8.3 Functions

The MHEG-SIR defines three kinds of functions:

- routines (see subclause 8.3.1);
- services (see subclause 8.3.2);
- predefined functions (see subclause 8.3.3).

All functions shall have a signature (or prototype) which consists of

- a type of return value;

- an ordered list of formal parameters defined by their type and passing mode.

All functions are referenced in a unique, unambiguous way using their function identifier.

Functions shall be either **synchronous** or **asynchronous**. When a synchronous function is called, the caller waits for the completion of the function execution and may therefore retrieve its result. When an asynchronous function is called, the caller only waits for an acknowledgement of reception of the request; it then resumes execution without waiting for the completion of the function.

As a consequence, asynchronous functions shall not have output parameters or a return value. Routines are always synchronous.

8.3.1 Routines

Routines are internal functions of interchanged scripts.

Routines shall be declared within the interchanged script. Routines shall consist of

- a signature;
- local variables;
- program code.

There shall not be more than 4 096 routines declared in an interchanged script.

Execution of a routine may be triggered

- by an explicit **call** (`CALL`) instruction, with the routine's function identifier being the operand of the instruction;
- upon reception of an exception during an **external call** (`XCALL`) instruction, where the message identifier of the received exception is mapped to the routine's function identifier by the handler definition table;
- upon examining the queue of received messages, either when no routine is executing or upon encountering a **yield** (`YIELD`) instruction, where the message identifier of the received message is mapped to the routine's function identifier by the handler definition table or is an MHEG-3 API `run` operation targeted at the routine.

Parameters may be passed to routines using either of the following modes:

- by **value**: a value of the parameter type is passed to the routine;
- by **reference**: a data identifier referencing a global variable, dynamic variable or constant whose type is the same as the parameter type is passed to the routine.

In both cases, the value of the passed parameter becomes the value of the local variable whose index corresponds to the parameter's index. The local variable corresponding to a parameter passed by reference shall be of the `data identifier` type.

Data identifiers to local variables shall not be passed by reference.

8.3.2 Services

Services are external functions provided by the run-time environment, that an interchanged script may invoke.

Services shall be declared within the interchanged script, as part of a package declaration, by

- their signature;

- their IDL global operation name.

There shall not be more than 256 services declared in each package of an interchanged script.

There shall not be more than 192 packages declared in an interchanged script.

A service may be called by an **external call** (XCALL) instruction.

Parameters may be passed to services using one of the following modes:

- **in**: a data identifier referencing a variable or constant whose type is the same as the parameter type is passed to the service;
- **inout**: a data identifier referencing a variable whose type is the same as the parameter type is passed to the service; upon returning, the variable is updated with its new value;
- **out**: same as **inout**, however the value of the variable is not used by the service.

8.3.3 Predefined functions

Predefined functions correspond to the operations of the MHEG-3 engine's interface.

Predefined functions have predefined function identifiers and therefore shall not be declared within an interchanged script.

As this part of ISO/IEC 13522 is not specifically linked to another part of ISO/IEC 13522, the MHEG API operations used to manipulate MHEG objects are not explicitly defined. However, this part of ISO/IEC 13522 specifies the procedure that shall be used to define an MHEG API and to specify the mapping of operations of this MHEG API to predefined function identifiers. This is described in Annex E.

In addition, this part of ISO/IEC 13522 defines the MHEG-3 API, i.e. the interface that MHEG-3 engines shall provide for the manipulation of scripts. This interface is described in Clause 15 and Annex F. This interface may be used from within scripts and is therefore mapped to predefined function identifiers. The list of these predefined functions and their identifiers is given in Annex C.

Predefined functions may be called and passed parameters to using the same mechanisms as with services.

8.4 Messages

The MHEG-SIR defines two kinds of messages:

- package exceptions (see subclause 8.4.1);
- predefined messages (see subclause 8.4.2).

All messages shall have a signature (or prototype) which consists of an ordered list of formal parameters (members) defined by their type.

All messages are referenced in a unique, unambiguous way by their message identifier.

8.4.1 Package exceptions

Package exceptions are sent to an rt-script by the run-time environment as a consequence of the invocation of a service by this rt-script.

Package exceptions shall be declared within the interchanged script, as part of a package declaration, by

- their signature;
- their IDL global exception name.

There shall not be more than 256 exceptions declared in each package of an interchanged script.

8.4.2 Predefined messages

Predefined messages sent to an rt-script may be one of the following:

- an exception of the MHEG-3 engine interface (i.e. the MHEG API), raised by the MHEG-3 engine as a consequence of the invocation of a predefined function by the rt-script;
- the consequence of invocation of an operation of the MHEG-3 API targeted at the rt-script;
- the `InstructionExecutionError` exception, which is raised as the consequence of an error occurring in the execution of an instruction of the rt-script.

Predefined messages have predefined message identifiers and therefore shall not be declared within an interchanged script.

As this part of ISO/IEC 13522 is not specifically linked to another part of ISO/IEC 13522, the MHEG API exceptions are not explicitly defined. However, this part of ISO/IEC 13522 specifies the procedure that shall be used to define an MHEG API and to specify the mapping of exceptions of this MHEG API to predefined message identifiers. This is described in Annex E.

Annex C specifies how the `InstructionExecutionError` and the messages resulting from MHEG-3 API operations shall be mapped to predefined message identifiers.

8.5 Instructions

The program code part of routines consists of a sequence of instructions. Unlike the rest of an interchanged script, which is handled upon preparation of the script, instructions need only be dealt with after creation of an rt-script, when the routine to which they belong is activated.

An instruction shall consist of one op-code (operation code) followed by zero or more operands. The number, type and encoding of operands is fully determined by the op-code.

As a rule, operands complete the instruction, whereas parameter values are taken from the parameter stack.

The performance of the instruction execution unit is described in Clause 9, whereas the precise semantics of each instruction are described in Clause 13.

8.6 Identifiers

Identifiers are used to reference MHEG-SIR entities (i.e. types, data, functions and messages) in an unambiguous way, throughout interchanged scripts.

8.6.1 Type identifiers

Type identifiers (TIDs) shall be encoded on two bytes as follows:

- primitive types and predefined types shall have predefined TIDs as defined by Annex C;
- declared types whose index (starting at 0) in the type declaration table is X shall have (X + 4000h) as TID.

Hence

- TIDs between 0 and 3FFFh shall reference predefined types;
- TIDs between 4000h and 7FFFh shall reference declared types.

8.6.2 Data identifiers

Data identifiers (DIDs) shall be encoded on two bytes as follows:

- constants whose index (starting at 0) in the constant declaration table is X shall have X as DID;
- global variables whose index (starting at 0) in the global variable declaration table is X shall have (X + 1000h) as DID;
- local variables whose index (starting at 0) in the local variable declaration table is X shall have (X + 8000h) as DID;
- dynamic variables shall have DIDs starting at 8100h. The procedure for allocating data identifiers to dynamic variables is not specified; MHEG-3 engines may therefore have different allocation schemes.

Hence

- DIDs between 0 and 0FFFh shall reference constants;
- DIDs between 1000h and 7FFFh shall reference global variables;
- DIDs between 8000h and 80FFh shall reference local variables;
- DIDs between 8100h and FFFFh shall reference dynamic variables.

8.6.3 Function identifiers

Function identifiers (FIDs) shall be encoded on two bytes as follows:

- routines whose index (starting at 0) in the routine declaration table is X shall have X as FID;
- predefined functions whose index (starting at 0) in the predefined function table is X shall have (X + 1000h) as FID;
- services whose index (starting at 0) in a package declaration is X and whose package index in the package declaration table is Y (starting at 0) shall have (((Y+64) << 8) + X) as FID.

Hence

- FIDs between 0 and 0FFFh shall reference routines;
- FIDs between 1000h and 3FFFh shall reference predefined functions;
- FIDs between 4000h and FFFFh shall reference services.

8.6.4 Message identifiers

Message identifiers (MIDs) shall be encoded on two bytes as follows:

- predefined messages whose index (starting at 0) in the predefined message table is X shall have X as MID;
- exceptions whose index (starting at 0) in a package declaration table is X and whose package index (starting at 0) in the package declaration table is Y shall have (((Y+64) << 8) + X) as MID.

Hence

- MIDs between 0 and 3FFFh shall reference predefined messages;
- MIDs between 4000h and FFFFh shall reference package exceptions.

9 The MHEG-SIR virtual machine

This Clause presents the MHEG-SIR virtual machine, i.e. the execution model for the MHEG-SIR code.

9.1 Structure of the MHEG-SIR virtual machine

The MHEG-SIR virtual machine is a set of logical, abstract components. The description of the MHEG-SIR virtual machine is intended for clarification of the operational semantics of the MHEG-SIR code.

An MHEG-3 engine shall have the same interpretation behaviour for MHEG-SIR code as the described virtual machine. It shall interpret MHEG-SIR declarations and instructions so as to produce similar external effects in all respects.

However, this implies no requirements on the technology or organisation that may actually be used to implement an MHEG-3 engine. An actual script interpreter need not be designed as described by the virtual machine, as long as it provides equivalent functionality.

The MHEG-SIR virtual machine consists of

- memory areas (see subclause 9.3);
- processing units (see subclause 9.5).

Some memory areas are associated with an mh-script and so shared by all the rt-scripts created from it. Other memory areas are associated with each rt-script.

Processing units only apply to one rt-script. However, an MHEG-3 engine may run several rt-scripts at the same time. In this case, it shall maintain a separate run-time context for each active rt-script.

NOTE: In other terms, the MHEG-SIR virtual machine is single-threaded. Multi-threaded applications can be achieved by associating each thread with a separate rt-script.

9.2 Structures and notations

9.2.1 Table

A table `T` consists of an array of homogeneous entries `T[i]` that may be accessed via their index `i`. These entries have the same structure, but not necessarily the same size. Entries consist of one or several fields `fld`. Some entries may be void. Indices are MHEG-SIR identifiers, i.e. consecutive numeric values taken in a given range, not necessarily starting at 0 for a given table. The underlying access mechanism (sequential indexing, direct access, hashcoding...) is not specified. The notation uses the following primitives to express manipulation of a table `T`:

- `T[i]` to access entry `i`;
- `T[i] = VAL` to assign value `VAL` to entry `i`;
- `T[i].fld` to access field `fld` of entry `i`;
- `T[i].fld = VAL` to assign value `VAL` to field `fld` of entry `i`.

9.2.2 Stack

A stack consists of an array of homogeneous elements. Elements are inserted on the top of the stack. Only the top element (last inserted) may be accessed at any time. When it is removed from the stack, it is lost, and the next element becomes the top of the stack. The notation uses the following primitives to express manipulation of the call stack `CS`:

- `CS.push(F)`: inserts frame `F` on the top of the stack, increments the frame pointer register (FP);
- `CS.pop()`: decrements FP, removes the top-of-stack frame, then returns it;
- `CS[FP]`: returns the value of the top-of-stack frame.

In the same way as for tables, the "`.`" notation is used to access stack element fields.

9.2.3 Parameter stack

The parameter stack is a special case because it is a byte (untyped) stack used to store typed values. The notation uses the following primitives to express manipulation of the parameter stack `PS`, where `tid` is the type identifier of a primitive type, as indicated by Table 1:

- `PS.push(VAL)`: inserts value `VAL` on the top of the stack;
- `PS.pop(tid)`: removes the top-of-stack value, whose type identifier is `tid`, then returns it;
- `PS[SP](tid)`: returns the value of the top-of-stack value, whose type identifier is `tid`.

9.2.4 Queue

A queue consists of an array of homogeneous elements. Elements are inserted at the end of the queue. Only the start element (first inserted) may be accessed at any time. When it is removed from the queue, it is lost, and the next element becomes the start of the queue. The notation uses the following primitives to express manipulation of the message queue `MQ`:

- `MQ.insert(M)`: inserts message value `M` at the end of the queue;
- `MQ.remove()`: increments the queue pointer register (QP), removes the element at the start of the queue, and returns it;
- `MQ[QP]`: returns the value of the element at the start of the queue.

In the same way as for tables, the "`.`" notation is used to access queue element fields.

9.2.5 Data representation

The representation of the structures and data is implementation-dependent. Although script interpreters may represent each value of a data type with a minimum number of bytes, they are not required to do so. Table 2 states this minimum number:

Table 2: Minimum number of bytes to represent values

Type	Minimum number of bytes to represent a value of the type
octet	1
short	2
long	4
unsigned short	2
unsigned long	4
float	4
double	8
boolean	1
character	2 (1 for restricted character sets)
data identifier	2
object reference	implementation-dependent
string	character size x (string length+1)
sequence	size of element type x sequence length (actual number of elements) + 2
array	size of element type x array size
structure	sum of the sizes of the element types
union	size of the "biggest" element type + 1
type identifier	2
function identifier	2
message identifier	2
package identifier	1

NOTE: The notation makes no distinction between fixed-length values and variable-length values. Script interpreters may store variable-length values on the heap. `VT[i].val` is used to access the value of a variable even though it could actually be stored in the variable table as a handle to the heap.

When `VAL` is a value of a constructed type, access to its elements is noted as follows:

- `VAL.tag`: tag of a union;
- `VAL.val`: value of a union;
- `VAL[n]`: value of the nth element of a sequence, string, structure or array;
- `VAL.lg`: actual length of a sequence or string.

Execution semantics are expressed using a C-like syntax. Expressions within single quotes indicate the corresponding value, e.g. 'void' indicates TID value 0.

9.3 Memory areas

In the MHEG-SIR virtual machine, memory areas are used to hold all the necessary information used to interpret a particular interchanged script.

Memory areas may be associated with either an mh-script (see subclause 9.3.1) or an rt-script (see subclause 9.3.2).

9.3.1 Mh-script memory areas

Mh-script memory areas should be completely filled at load-time, i.e. upon initialisation of an mh-script. They shall be accessible for use by all rt-scripts created from this mh-script. Mh-script memory areas shall not be modified at run-time until the mh-script is destroyed, unless otherwise specified (e.g. for the package definition table). Mh-script memory areas comprise

- data areas (see subclause 9.3.1.1);
- code areas (see subclause 9.3.1.2).

9.3.1.1 Data areas

Data areas are used to store the definitions and values of the script's global data. Data areas comprise

- the type definition table (TT) (see subclause 9.3.1.1.1);
- the constant table (CT) (see subclause 9.3.1.1.2);
- the global variable definition table (GT) (see subclause 9.3.1.1.3).

9.3.1.1.1 Type definition table

The type definition table maps all the script's defined types, represented by type identifiers, to their description:

- `TT[TID].val`: description of the type.

NOTE: The representation used for the type description is not specified; however, it should allow to check easily whether a value belongs to a type.

9.3.1.1.2 Constant table

The constant table maps all the script's constants, represented by data identifiers, to their type and value:

- `CT[DID].TID`: type of the constant (expressed as a type identifier);
- `CT[DID].val`: value of the constant (depending on its type).

9.3.1.1.3 Global variable definition table

The global variable definition table maps all the script's global variables, represented by data identifiers, to their type and initial value:

- `GT[DID].TID`: type of the global variable (expressed as a type identifier);
- `GT[DID].val`: initial value of the global variable (depending on its type).

9.3.1.2 Code areas

Code areas are used to store the addresses and program code of the script's functions. Code areas comprise

- the routine definition table (RT) (see subclause 9.3.2.1);
- the package definition table (PT) (see subclause 9.3.2.2);
- the service definition table (ST) (see subclause 9.3.2.3);
- the exception definition table (XT) (see subclause 9.3.2.4);
- the handler definition table (HT) (see subclause 9.3.2.5);
- the program code area, consisting of the sequence of instructions of each routine (see subclause 9.3.2.6).

9.3.1.2.1 Routine definition table

The routine definition table maps all the script's routines, represented by function identifiers, to their signature description, their local variable declaration and their program code:

a) `RT[FID].TID`: type of return value (expressed as a type identifier);
b) `RT[FID].nbp`: number of parameters;

c) `RT[FID].sig`: signature description, where
   1) `RT[FID].sig[i].TID` is the type (expressed as a type identifier) of the ith parameter;
   2) `RT[FID].sig[i].mod` is the passing mode (**value** or **reference**) of the ith parameter;
d) `RT[FID].LT`: declaration of the routine's local variables (whose `nbp` first elements are the actual parameters of the routine);
e) `RT[FID].IP`: pointer to the first instruction in the routine code.

Local variables used to hold parameters passed by **reference** shall have `data identifier` as their type, while local variable used to hold parameters passed by **value** shall have the same type as in the signature description for the corresponding parameter.

9.3.1.2.2 Package definition table

The package definition table maps all the script's defined packages, represented by package identifiers (PIDs) as declared by the MHEG-SIR package declaration table, to package names and additional information:

- `PT[PID].name`: name of the package;
- `PT[PID].nbf`: number of services in the package;
- `PT[PID].nbm`: number of exceptions defined by the package;
- `PT[PID].sts`: current status of the package (**unchecked**, **available**, **ready**, **opened**);
- `PT[PID].or`: initial object reference of the package.

A package is initially at **unchecked** status. It becomes **available** once the **package availability** procedure has been performed successfully. It then becomes **ready** once the **package load** procedure has been performed successfully. Finally, it is **opened** when there is a valid initial object reference to the package, stored in the `PT[PID].or` field, for use by further service invocations.

As an exception to the rule stated in subclause 9.3.1,

- The `PT[PID].sts` fields may be modified at run-time, each time the status of a package changes.
- The `PT[PID].or` fields may be modified at run-time, when a package is loaded.

9.3.1.2.3 Service definition table

The service definition table maps all the script's defined external services, represented by MHEG-SIR function identifiers, to their signature description and to their IDL global operation name:

a) `ST[FID].TID`: type of return value (expressed as a type identifier);
b) `ST[FID].syn`: calling mode (**synchronous**, **asynchronous**);
c) `ST[FID].nbp`: number of parameters;
d) `ST[FID].sig`: signature description, where
   1) `ST[FID].sig[i].TID` is the type (expressed as a type identifier) of the ith parameter;
   2) `ST[FID].sig[i].mod` is the passing mode (**in**, **inout** or **out**) of the ith parameter;
e) `ST[FID].name`: the IDL global name of the operation which the service invokes.

The IDL platform-specific mapping specification shall be used to map `ST[MID].name` to a platform-specific name.

9.3.1.2.4 Exception definition table

The exception definition table maps all the interchanged script's defined messages, represented by message identifiers, to their signature description and their IDL global exception name:

- `XT[MID].name`: the IDL global name of the exception which causes the message;
- `XT[MID].nbm`: number of members;

- `XT[MID].sig`: signature description, where `XT[MID].sig[i].TID` is the type (expressed as a type identifier) of the ith member.

The IDL platform-specific mapping specification is used to map `XT[MID].name` to a platform-specific name.

9.3.1.2.5 Handler definition table

The handler definition table maps messages, represented by message identifiers, to routines represented by function identifiers:

- `HT[MID].FID`: identifier of routine to invoke for handling the message.

If a message is mapped to a routine in the handler table, the signature of this routine need match the signature of this message. Matching between the signatures shall be checked at load-time and non-matching entries shall be rejected.

The handler definition table is used by the rt-script execution unit. When the rt-script execution unit removes a message from the message queue, it invokes the routine that corresponds to the message, with the message parameters as its parameters.

9.3.1.2.6 Program code area

An instruction consists of one 1-byte op-code followed by zero to three operand bytes. The op-code completely determines the number and length of its operands, according to the instructions table. Both op-codes and operands are coded in an optimised fashion so as to ease switching.

NOTE: A script interpreter (especially on 32-bit machines) may align instructions at load-time, i.e. insert padding bytes in order to represent each instruction on four bytes; this makes it easy to increment the instruction pointer. As a variant, a script interpreter may instead leave instructions packed, and determine the number of bytes to increment at run-time.

9.3.2 Rt-script memory areas

Rt-script memory areas are initialised upon creation of the rt-script and may be modified during its execution. Rt-script memory areas comprise

- dynamic memory areas (see subclause 9.3.2.1);
- registers (see subclause 9.3.2.2).

9.3.2.1 Dynamic memory areas

Dynamic memory areas are used to represent the data and the current execution context of the rt-script.

Dynamic memory areas comprise

- the variable table (VT) (see subclause 9.3.2.1.1);
- the call stack (CS) (see subclause 9.3.2.1.2);
- the parameter stack (PS) (see subclause 9.3.2.1.3);
- the message queue (MQ) (see subclause 9.3.2.1.4);
- the heap (see subclause 9.3.2.1.5).

9.3.2.1.1 Variable table

The variable table maps the rt-script's variables, represented by data identifiers, to their type and current value:

- VT[DID].TID: type of the variable (expressed as a type identifier);
- VT[DID].val: current value of the variable (depending on its type).

The variable table is initialised upon creation of the rt-script. It consists of two subtables:

- a copy of the global variable table associated with the mh-script;
- the local variable table of the currently executing routine.

VT[DID].val fields are modified every time a variable is assigned by the execution of a variable assignment instruction.

When the current routine changes (following execution of a CALL, RET or YIELD instruction), the local variable table is stored and replaced in the VT by the local variable table of the new routine. The first entries of a local variable table are the parameters passed to the function.

9.3.2.1.2 Call stack

The call stack is used to store the current invocation context.

The call stack is an array of call frames. Every frame shall correspond to the context at the time of invocation of an active function (routine, external function or MHEG action). Frames shall be stored on the CS in order of invocation. The top frame of the CS, if any, shall describe the execution context of the routine that called the currently executing function.

Each frame shall consist of the following elements:

- CS[i].FID: function identifier of the caller;
- CS[i].IP: pointer to the instruction to return to after the current function returns;
- CS[i].LT: local variable table of the caller (at invocation time);
- CS[i].SP: pointer to the top of the parameter stack (at invocation time).

The LT field of a call frame shall have the structure of a variable table:

- CS[i].LT[DID].TID: type identifier of the variable whose identifier is DID;
- CS[i].LT[DID].val: value of the variable.

The call stack is modified by certain control flow instructions. Initially the call stack shall be empty. When a function is invoked, a frame describing this call shall be pushed onto the call stack. When a function is returned from, this frame shall be popped from the call stack. The address of the top frame of the call stack shall be stored at all times in the FP register.

9.3.2.1.3 Parameter stack

The parameter stack is used to store the parameters and return values of instructions. The parameter stack is an array of data values. The type of the data value is determined by the operation sequence that pushes the value on the stack.

The parameter stack is used by the MHEG-SIR instruction execution unit. Initially the parameter stack shall be empty. It is modified by most instructions (arithmetic operators, logical operators, comparison operations, stack manipulation, variable assignment, conditional jumps, calls). When an instruction is executed, it shall pop its parameters from the parameter stack and push its return value back onto the parameter stack. The address of the top frame of the parameter stack shall be stored at all times in the stack pointer (SP) register.

9.3.2.1.4 Message queue

The message queue is used to buffer the messages that are received by the script interpreter. Each item in the queue shall consist of the following elements:

- `MQ[i].MID`: message identifier;
- `MQ[i].LT`: list of message parameters.

The LT field of a message queue item shall have the structure of a variable table:

- `MQ[i].LT[j].TID`: type identifier of the jth parameter;
- `MQ[i].LT[j].val`: value of the jth parameter.

Messages shall be inserted into the message queue by the script interpreter asynchronously as they are generated in the external environment. The message queue shall be processed by the rt-script execution unit when either of the following occurs:

- the rt-script is not running, i.e. there is no currently executing routine;
- a `YIELD` instruction is encountered.

The start of the message queue (next message to pop) shall be stored at all times in the QP (queue pointer) register. Initially the message queue shall be empty.

9.3.2.1.5 Heap

The heap is used to store dynamic variables, represented by data identifiers, as their type and current value:

- `VT[DID].TID`: type of the variable (expressed as a type identifier);
- `VT[DID].val`: current value of the variable (depending on its type).

Dynamic variables are referenced by handles of an opaque type, whose representation is not specified. Data identifiers are internally mapped to these handles so that dynamic variables be accessed in the same way as other variables.

`VT[DID].val` fields are modified every time a variable is assigned by the execution of a variable assignment instruction.

The application is responsible for explicit allocation and deallocation of dynamic variables using the `ALLOC` and `FREE` instructions.

NOTE: Script interpreters may also use the heap to store the values of global or local variables of a variable-length type. In this case, a heap handle is stored in the table instead of the data itself.

9.3.2.2 Registers

Registers hold specific states of the virtual machine and need be frequently modified during the execution of an rt-script.

The registers maintained by the MHEG-SIR virtual machine are

- the instruction pointer (IP) or program counter (see subclause 9.3.2.2.1);
- the frame pointer (FP) (see subclause 9.3.2.2.2);
- the stack pointer (SP) (see subclause 9.3.2.2.3);
- the queue pointer (QP) (see subclause 9.3.2.2.4);
- the instruction register (IR) (see subclause 9.3.2.2.5);
- the error register (ER) (see subclause 9.3.2.2.6);

- the function register (FR) (see subclause 9.3.2.2.7).

The representation of data held by pointer registers is not specified. All registers shall be initialised to a null value whose representation is not specified.

9.3.2.2.1 Instruction pointer register

The IP register points to the next instruction to be executed within a routine's program code. This register shall be modified by the rt-script execution unit and by the MHEG-SIR instruction execution unit as part of the execution of instructions.

9.3.2.2.2 Instruction register

The IR register holds the code for the instruction which is currently executing. This register shall be updated by the rt-script execution unit each time a new instruction is loaded, and accessed by the MHEG-SIR instruction execution unit.

NOTE: The IR need not be more than 4 bytes long, but its actual size is not specified.

9.3.2.2.3 Error register

The ER holds the code of the last error encountered during execution of an instruction. This register shall be updated by the MHEG-SIR instruction execution unit, every time it encounters an error. The null value indicates that up to the current time no error has been encountered during the execution of the rt-script.

The error codes are predefined. The error codes raised by each instruction are defined in Clause 13.

When an error is raised during execution of an instruction, ER shall be set to a non-null value and an `InstructionExecutionError` exception shall be raised. This results in the corresponding message being inserted into the message queue.

9.3.2.2.4 Stack pointer register

The SP register points to the top of the parameter stack. The value of this register shall be updated by the MHEG-SIR instruction execution unit as follows:

- it shall be incremented every time data is pushed onto the parameter stack;
- it shall be decremented every time data is popped off the parameter stack.

9.3.2.2.5 Frame pointer register

The FP register points to the top frame of the call stack. The value of this register shall be updated by the MHEG-SIR instruction execution unit as follows:

- it shall be incremented every time a function is called;
- it shall be decremented every time a function is returned from.

9.3.2.2.6 Queue pointer register

The QP register points to the next message to be removed from the message queue. The value of this register shall be decremented by the script interpreter every time a message is removed.

9.3.2.2.7 Function register

The FR register holds the FID of the currently executing function. The value of this register shall be updated by the script interpreter every time a function is called or returned from.

9.4 Script statuses

9.4.1 Mh-script statuses

The status of an mh-script shall be either **available** or **not available**.

9.4.1.1 Not available

The status of an mh-script shall be **not available** if it is in one of the following cases:

- mh-script initialisation (i.e. the effect of the MHEG-3 API `prepare` operation) has not been achieved on this mh-script;
- mh-script destruction (i.e. the effect of the MHEG-3 API `destroy` operation) has been requested on this mh-script.

9.4.1.2 Available

The status of an mh-script shall be **available** if mh-script initialisation has been successfully achieved on this mh-script and if mh-script destruction has not yet been requested.

This implies that

- the interchanged script has been parsed and the mh-script memory areas fully completed accordingly;
- the packages referenced in the mh-script are available and have been loaded according to the **package load** procedure.

9.4.2 Rt-script statuses

The status of an rt-script shall be one of the following: **not ready**, **ready**, **running, erroneous**.

9.4.2.1 Not ready

The status of an rt-script shall be **not ready** if it is in one of the following cases:

- rt-script initialisation (i.e. the effect of the MHEG-3 API `new` operation) has not been achieved on this rt-script;
- rt-script destruction (i.e. the effect of the MHEG-3 API `delete` operation) has been requested on this rt-script.

The status of an rt-script is initially **not ready**. Otherwise, it changes to **not ready** when a **delete** operation is invoked on this rt-script.

9.4.2.2 Ready

The status of an rt-script shall be **ready** if all of the following conditions are met:

- rt-script initialisation has been successfully achieved on this mh-script;
- rt-script destruction has not yet been requested on this rt-script;
- the IP register is set to 'null', i.e. there is no currently executing routine;
- the ER register is set to 'null'.

This implies that the calling stack, message queue and parameter stack are void.

However, the global variable values need not be the same as the initial values; once it has no more instruction to execute and no message in the message queue, an rt-script goes back to **ready** status.

The status of an rt-script changes

- from **not ready** to **ready** when a `new` operation is invoked on this rt-script;
- from **running** to **ready** when the rt-script execution unit no longer has instructions to execute or as the result of invoking a `stop` or `reinit` operation;
- from **erroneous** to **ready** as the result of invoking a `stop` or `reinit` operation.

9.4.2.3 Running

The status of an rt-script shall be **running** if all of the following conditions are met:

- rt-script initialisation has been achieved without error on this mh-script;
- rt-script destruction has not yet been requested on this rt-script;
- the IP register is not 'null', i.e. there is a currently executing routine;
- the ER register is set to 'null'.

The status of an rt-script changes from **ready** to **running** when there is a message in the message queue and the rt-script execution unit is activated. This may occur as the result of invoking a `run` operation.

9.4.2.4 Erroneous

The status of an rt-script shall be **erroneous** if the ER register is not 'null', i.e. if an error has occurred during the rt-script execution.

The status of an rt-script changes from **running** to **erroneous** when an instruction execution error is raised by the rt-script instruction execution unit.

9.5 Processing units

This subclause describes the MHEG-SIR virtual machine's flow of control and the semantics of instructions.

For the purposes of the virtual machine description, the script interpreter's main process is assumed to run in parallel with all active rt-script execution units. Scheduling of the different tasks is not specified.

9.5.1 Message reception

The script interpreter's main process receives and handles events. In the absence of any events, it is idle. Events received by the script interpreter may be

- MHEG-3 API operation invocations;
- messages corresponding to the occurrence of a exception raised as the result of invoking either a service or predefined function.

9.5.1.1 MHEG-3 API operations

MHEG-3 API operations may be invoked by an rt-script execution unit, by another component of the MHEG-3 engine or by external processes outside the MHEG-3 engine.

When an MHEG-3 API operation is invoked, the main process shall proceed as specified by the semantics of the MHEG-3 API described in Clause 15.

9.5.1.2 External exception

When a message coming from either the action interpreter (MHEG API exception) or the run-time environment is targeted at an rt-script, then if this message actually corresponds to an exception raised by the MHEG API or the run-time environment as a consequence of the invocation of an operation resulting from

an XCALL instruction by this rt-script, the main process shall parse the exception's parameters and construct an message structure consisting of the message identifier of the exception followed by its actual members (starting with the object reference of the originating object). Then

- if the exception results from the invocation of a currently executing synchronous operation, the main process shall request the rt-script execution unit to terminate the XCALL instruction (therefore popping its frame from the call stack) without looking for output parameters or a return value, then immediately afterwards to trigger the routine corresponding to the exception's message identifier, with the exception's members as its actual parameters; the effect shall be the same as if this routine had been invoked by a CALL instruction;

- if the exception results from the invocation of a previously terminated synchronous operation (whether successfully or not), the main process shall ignore the exception;

- if the exception results from the invocation of an asynchronous operation, the main process shall insert the constructed message into the message queue of the target rt-script.

9.5.1.3 InstructionExecutionError exception

When the internal InstructionExecutionError exception is raised by the execution of an instruction, the main process shall construct a message structure consisting of the message identifier corresponding to this exception, followed by one member set to the value of the error register, then insert it into the message queue of the rt-script whose execution raised the exception.

9.5.1.4 MHEG-3 API exception

When an exception resulting from the invocation of an MHEG-3 API operation is returned to an rt-script, the main process shall construct a message structure consisting of the message identifier corresponding to the exception, followed by its members, then insert it into the message queue of the rt-script that invoked the operation.

9.5.2 Mh-script initialisation

When the MHEG-3 API prepare operation is invoked, the script interpreter shall access the stream or file using the provided system identifier and parse the script. The script interpreter shall then

- parse the declarations part and initialize the CT, GT, TT, RT, ST, PT, XT, HT and the RT[i].LT for each routine i; this includes the appropriate checks (handler verification, **package availability** procedure);
- parse the structure of the instructions part to fill in the program code area of each routine;
- perform the **package load** procedure, establishing static links with packages according to the platform mapping specification;
- put the mh-script to **available** status.

NOTE: The semantics of package loading need be defined by the platform mapping specification. The MHEG-3 engine may take the responsibility to optimise its resource management strategy, e.g. by unloading packages temporarily in order to release memory, or by loading packages only as rt-scripts are created or even as services are invoked.

9.5.3 Rt-script initialisation

When the MHEG-3 API new operation is invoked, the script interpreter shall create a context for the target rt-script, i.e. the script interpreter shall

- initialise the dynamic memory areas;
- initialise all registers to null values;

- create an rt-script execution unit for the rt-script;
- put the rt-script to **ready** status.

9.5.4 Rt-script execution unit

When activated and unless requested to stop the current rt-script, the rt-script execution unit shall perform as follows:

```
rt-script-execution-unit ()
{
FID fid = 'null';
if (IP == 'null')                                       // no next instruction
{
        while (fid == 'null')
        {
                if (QP == 'null') then exit;            // return
                fid= HT[MQ[QP].MID].FID;                // find handler for message
                if (fid != 'null') then                 // handler found
                {
                        CS.push({IP, FR, SP, 'null'});  // stack routine call
                        FR = fid;
                        IP = RT[FR].IP;                 // branch to start of routine
                }
                MQ.remove();                            // remove message
        }
}
// endif

while (IP != 'null'):
{
        IR = *IP++;                     // load next instruction and increment program counter
        instruction-execution-unit(); // call the MHEG-SIR instruction execution unit
}
// endwhile
return;                                 // return to script interpreter
}
```

9.5.5 MHEG-SIR instruction execution unit

When called by the rt-script execution unit, the MHEG-SIR instruction execution unit of an rt-script shall decode the op-code contained in the first byte of the IR, then interpret the instruction corresponding to this op-code as specified by Clause 13, then return.

The instruction execution unit pops from the parameter stack those parameters that are used to perform the instruction (if any). It pushes on the parameter stack those parameters that are the result of the instruction (if any).

Table 3 summarises the effects of the instructions on the various elements of the MHEG-SIR virtual machine as defined by this Clause.

10 Provisions for run-time environment access

This Clause describes the mechanisms defined by this part of ISO/IEC 13522 to make it possible for rt-scripts to access and interchange data with external functions provided by the run-time environment on the execution platform.

10.1 General model

The interface that external software available in the run-time environment provides need be declared in the interchanged script as part of its package declaration, so that the script interpreter know how to access this interface when the script invokes it.

A package declaration describes a set of services (i.e. external functions) by their signature, i.e. the type and passing mode of each parameter.

MHEG-SIR specifies how calling external functions, passing parameters, getting back return values and handling exceptions shall be expressed within interchanged scripts.

This part of ISO/IEC 13522 also specifies how these expressions shall be interpreted by MHEG-3 engines.

This part of ISO/IEC 13522 also deals with interchange (i.e. function call, parameters passing, return value retrieval and exception handling) between an MHEG-3 engine and the run-time environment. For this purpose, this part of ISO/IEC 13522 contains provisions for specifying how access to these functions should be provided to MHEG-3 engines by external software. Such a convention, called a platform mapping specification, is dependent on the run-time platform.

Platform mapping specifications conforming to the provisions of this part of ISO/IEC 13522 need to be registered to ensure the interoperability of run-time environment services with any compliant MHEG-3 engine on this platform. If a platform mapping specification exists for the platform, MHEG-3 engines shall conform to this platform mapping specification in order to access run-time environment services.

MHEG-3 engine implementations shall document in their conformance document the platform mapping specification(s) to which they conform.

NOTE: If existing software does not comply with the platform mapping specification and need be accessed from MHEG-SIR scripts, it may be embedded into an interface that translates its own interface conventions into those of the platform mapping specification.

10.2 Declaration of IDL interfaces

The interface of external software intended for use by an interchanged script may contain

- operation declarations;
- exception declarations;
- type declarations.

Types shall be declared in the type declaration of this interchanged script.

Operations and exceptions shall be declared in the package declaration of this interchanged script. This package declaration shall be assigned a package identifier and shall consist of

- the name of the package;
- a set of service descriptions;
- a set of exception descriptions.

Service descriptions shall be assigned a function identifier and shall consist of

- the name of the operation;
- the function signature, i.e. the type and passing mode of each parameter and the type of the return value.

Exception descriptions shall be assigned a message identifier and shall consist of

- the name of the exception;
- the exception signature, i.e. the type of each member.

Identifiers (package identifiers, type identifiers, function identifiers) are used by MHEG-SIR scripts to refer to types and functions; a function identifier for an external operation can be built from a package identifier and the index of the service declaration in this package, while a message identifier for an external exception can be built from a package identifier and the index of the exception declaration in this package.

Names (package names, operation names, exception names) shall be used by the script interpreter to link with the actual implementation of the external software.

An MHEG-SIR package declaration lies at the same abstraction level as an IDL specification. This part of ISO/IEC 13522 defines the rules for mapping an IDL specification into a package declaration. Clause 14 specifies

- how an IDL data type description shall be mapped to an MHEG-SIR data type description;
- how an IDL operation description shall be mapped to an MHEG-SIR service description;
- how an IDL exception description shall be mapped to an MHEG-SIR exception description.

10.3 Invocation of external operations in an MHEG-SIR program

A service described in a package declaration shall be invoked from an MHEG-SIR program as follows:

- variables of expected types corresponding to the return value (if any) and to each parameter shall be declared within the interchanged script (except the originating object's object reference, which shall be implicit);
- the program shall assign those variables which correspond to input or input/output parameters;
- the program shall push onto the stack the data identifiers of all these variables in right-to-left order (the identifier of the variable corresponding to the return value is pushed first, then the actual parameters, with the object reference (implicit parameter) of the target being pushed last);
- the program shall invoke the operation using an **external call** (XCALL) instruction with the function identifier of the invoked operation as operand;
- the program shall exploit the function results using the variables corresponding to the return value, the **inout** parameters and the **out** parameters.

10.4 Handling of external exceptions in an MHEG-SIR program

An exception described in a package declaration shall be handled by an MHEG-SIR program as follows:

- variables of expected types corresponding to each member shall be declared within the interchanged script (except the originating object's object reference, which shall be implicit);
- a routine whose parameters correspond to the exception members shall be declared within the routine declaration part of the interchanged script;
- the mapping between the identifiers of this handling routine and the exception shall be declared in the handler declaration part of the interchanged script.

10.5 Invocation of external operations by an MHEG-3 engine

When an interchanged script expresses invocation of an operation as described in subclause 10.3, the script interpreter shall behave as described by the semantics of the XCALL instruction in Clause 13. As part of this performance, it shall interpret the mechanisms described in subclause 10.3 in translating them into the run-time environment access mechanisms as defined by the platform mapping specifications.

NOTE: For instance, an MHEG-3 engine may translate a variable identifier pushed onto the stack as a service parameter into either a value or a real memory address to be passed to the external software that provides the service.

10.6 Handling of external exceptions by an MHEG-3 engine

When an exception is raised by an external service, this results in a message being transmitted to the MHEG-3 engine according to the run-time environment access mechanisms defined by the platform mapping specifications.

The script interpreter shall then behave as described in subclause 9.5.1.2.

10.7 Platform mapping specifications

A platform mapping specification shall contain all of the following:

- the description of the platform to which the specification applies;
- the **package availability** procedure, which MHEG-3 engines shall use to check the availability of a given package within the run-time environment;
- the **package load** procedure, which MHEG-3 engines shall use to make the operations of a given package accessible to an rt-script;
- the **package unload** procedure, which MHEG-3 engines shall use to unload a package;
- the **operation invocation** procedure, which MHEG-3 engines shall use to invoke a given operation;
- the **data encoding** rules, which MHEG-3 engines shall use to encode the value of **in** or **inout** parameters of an operation and to decode the value of **out** or **inout** parameters of an operation or exception members;
- the **parameter passing** procedures, which MHEG-3 engines shall use to pass **in**, **inout** and **out** parameters to an operation;
- the **return value retrieval** procedure, which MHEG-3 engines shall use to retrieve the return value of an operation;
- the **exception retrieval** procedure, which MHEG-3 engines shall use to retrieve exceptions raised by an operation.

The contents of a platform mapping specification are defined in Annex D.

11 Provisions for MHEG object manipulation

This Clause describes the mechanisms defined by this part of ISO/IEC 13522 to make it possible for rt-scripts to manipulate MHEG objects.

11.1 Invoking MHEG actions

MHEG-SIR is used to express invocation of MHEG actions as defined by the MHEG API.

The MHEG API is defined using IDL. The mapping from an IDL definition to an MHEG-SIR package declaration and type declaration is defined in Clause 14. However, the MHEG API package is considered as a predefined one. So its declaration shall not be included explicitly in interchanged scripts. The mapping mechanism is similar to the external function declaration mechanism described in Clause 10, except that the IDL types and operations defined by the MHEG API shall not be declared as part of the MHEG-SIR code, but are instead dealt with as predefined types and predefined external functions.

The mechanism used to invoke an MHEG action is similar to the invocation of a service provided by the run-time environment. An `XCALL` instruction is used. Types defined in the MHEG API package are referred to using a predefined type identifier. Functions described in the MHEG API package are referred to using a predefined function identifier.

11.1.1 Sending messages to other scripts

The MHEG-3 API package is considered as a predefined one. Within an interchanged script, messages may be efficiently targeted at other scripts using the predefined functions mapping MHEG-3 API operations. An rt-script can thus pass and receive parameters and call routines from another rt-script.

NOTE: This may be used to implement the concept of "library" or "utility" scripts. This may also be used to synchronise rt-scripts.

11.1.2 Exchange of information with MHEG objects

Exchange of information between an rt-script and other MHEG entities (including other rt-scripts) may be expressed using the MHEG API operations mapping the MHEG "set data" and "get data" actions. MHEG content objects embedding generic values may be used to constitute a shared memory area among MHEG objects.

Waiting for a signal from another object may be translated by a loop including a call to the MHEG API operation mapping the MHEG "get data" action until the expected value is retrieved.

Generating a signal may be translated by a call to the MHEG API operation corresponding to the "set data" MHEG action.

NOTE: As far as exchange of information among rt-scripts is concerned, use of the mechanism described in subclause 11.1.1 is recommended.

11.2 Receiving MHEG messages

MHEG-SIR is used to express handling of messages resulting from MHEG actions. These messages may be either of the following:

- MHEG-3 API `run` operations;
- MHEG API exceptions.

11.2.1 MHEG-3 API `run` operations

The MHEG "set parameters" and "run" actions that may be targeted to an rt-script should result in the MHEG-3 API `setParameter` and `run` operations. Invocation of the `run` operation results in a message being inserted into the rt-script's message queue with

- as message identifier, a predefined message identifier which is mapped to the routine identifier of the targeted routine;
- as members, the parameters previously set by the `setParameter` operation.

11.2.2 MHEG API exceptions

The MHEG API exceptions are considered as messages which are sent to the script interpreter as the result of invoking an MHEG API operation. These exceptions have predefined message identifiers. The script interpreter shall process these messages in the same way as it would process an exception coming from the run-time environment, as described in subclause 9.5.1.2.

12 MHEG-SIR declarations

This Clause defines the structure of interchanged scripts. This Clause also specifies the way the virtual machine deals with parsing of an interchanged script.

The following notations conventions are used:

- non-terminals are written as normal text;
- terminal types are written in uppercase;
- enumerated values are enclosed in single quotes;
- "`:=`" indicates a definition;
- "`|`" indicates a choice in a production;
- "`*`" indicates zero or more occurrences of the preceding type;
- "`+`" indicates one or more occurrences of the preceding type;
- "`?`" indicates zero or one occurrence of the preceding type (optional type).

NOTE: The complete grammar of interchanged scripts is described in Annex H.

An interchanged script shall consist of

- a sequence of type declarations;
- a sequence of constant declarations;
- a sequence of global variable declarations;
- a sequence of package declarations;
- a sequence of message handler declarations;
- a sequence of routine declarations.

```
InterchangedScript      ::=   TypeDeclaration*
                              ConstantDeclaration*
                              VariableDeclaration*
                              PackageDeclaration*
                              HandlerDeclaration*
                              RoutineDeclaration*
```

12.1 Type declaration

Type declarations are used to describe the types of the interchanged script.

A type declaration shall consist of

- a type identifier (optional);
- a type description.

```
TypeDeclaration         ::=   TypeIdentifier?
                              TypeDescription
```

12.1.1 Type identifier

Type identifiers are used to reference the type description throughout the interchanged script.

The type identifier shall be a positive integer within the range allowed for declared types. It shall correspond to the maximum number of predefined types incremented by the index (starting at 0) of the declaration in the type declarations part.

If the type identifier is not provided, it shall be computed by the script parser.

```
TypeIdentifier          ::=   INTEGER
```

12.1.2 Type description

Type descriptions describe the structure of a declared type.

The type description shall be one of the following:

- a string description;
- a sequence description;
- an array description;
- a structure description;
- a union description.

```
TypeDescription          ::=   SequenceDescription
                         |     StringDescription
                         |     ArrayDescription
                         |     StructureDescription
                         |     UnionDescription
```

12.1.2.1 String description

A string description shall consist of an integer (optional).

```
StringDescription        ::=   INTEGER?                    // String (max) size
```

The integer represents the maximum size of the string; if it is not provided, the string shall be unbounded.

12.1.2.2 Sequence description

A sequence description shall consist of

- an integer (optional);
- a type identifier.

```
SequenceDescription      ::=   INTEGER?                    // Sequence (max) size
                               TypeIdentifier
```

The integer represents the maximum size of the sequence; if it is not provided, the sequence shall be unbounded.

The type identifier represents the type of element of the sequence.

12.1.2.3 Array description

An array description shall consist of

- an integer;
- a type identifier.

```
ArrayDescription         ::=   INTEGER                     // Array size
                               TypeIdentifier
```

The integer represents the size of the array.

The type identifier represents the type of element of the array.

12.1.2.4 Structure description

A structure description shall consist of a sequence of type identifiers.

```
StructureDescription     ::=   TypeIdentifier+
```

Each type identifier represents the type of one of the fields of the structure.

12.1.2.5 Union description

A union description shall consist of a sequence of one or more type identifiers.

```
UnionDescription         ::=   TypeIdentifier+
```

Each type identifier represents the type of one of the choices of the union.

12.2 Constant declaration

Constant declarations are used to describe the types and values of the constants of the interchanged script.

A constant declaration shall consist of

- a data identifier (optional);
- a type identifier;
- a constant value.

```
ConstantDeclaration      ::=   DataIdentifier?
                               TypeIdentifier
                               ConstantValue
```

12.2.1 Data identifier

Data identifiers are used to reference data throughout the interchanged script.

The data identifier shall be a positive integer within the range allowed for constants. It shall correspond to the index (starting from 0) of the declaration in the constant declarations part.

If the data identifier is not provided, it shall be computed by the script parser.

```
DataIdentifier           ::=   INTEGER
```

12.2.2 Type identifier

The type identifier represents the type to which the value of the constant belongs.

12.2.3 Constant value

The constant value represents the value to which the constant corresponds throughout the script.

If the type of the constant is a primitive or `string` type, the constant value shall consist of an immediate value expressed in this type.

If the type of the constant is a `sequence` type, the constant value shall consist of a sequence of constant values, whose length is less or equal to the size of the `sequence` type and whose type is the element type of the sequence description.

If the type of the constant is an `array` type, the constant value shall consist of a sequence of constant values, whose length equal to the size of the `array` type and whose type is the element type of the array description.

If the type of the constant is a `structure` type, the constant value shall consist of a sequence of constant values, whose length is equal to the number of elements in the `structure` type; each of these values shall be of the same type as the corresponding element type in the structure description.

If the type of the constant is a `union` type, the constant value shall consist of an integer representing the index (starting from 0) of the choice in the `union` type and a constant value whose type is the type of element of the corresponding rank in the union description.

```
ConstantValue          ::=   BOOLEAN
                       |     OCTET
                       |     INTEGER            // all numeric types
                       |     REAL               // float or double
                       |     STRING             // character or string
                       |     DataIdentifier
                       |     ConstantValue*   // sequence, array or structure
                       |     UnionValue

UnionValue             ::=   INTEGER            // Tag index
                             ConstantValue
```

12.3 Global variable declaration

Global variable declarations are used to describe the types and initial values of the global variables of the interchanged script.

A global variable declaration shall consist of

- a data identifier (optional);
- a type identifier;
- a constant reference (optional).

```
VariableDeclaration    ::=   DataIdentifier?
                             TypeIdentifier
                             ConstantReference? // Initial value
```

12.3.1 Data identifier

Data identifiers are used to reference data throughout the interchanged script.

The data identifier shall be a positive integer within the range allowed for global variables. It shall correspond to the maximum number of constants incremented by the index (starting from 0) of the declaration in the global variable declarations part.

If the data identifier is not provided, it shall be computed by the script parser.

12.3.2 Type identifier

The type identifier represents the type to which the value of the global variable belongs.

12.3.3 Constant reference

The constant reference represents the initial value of the global variable.

The constant reference shall be one of the following:

- a data identifier referencing a constant;
- a constant value as described in subclause 12.2.3.

In any case, the value to which this constant reference refers shall be of the type of the global variable.

If the constant reference is not provided, the script interpreter shall assign the global variable a default value if its type allows for it. Otherwise, it shall remain undefined until assigned by an instruction.

```
ConstantReference      ::=   DataIdentifier
                       |     ConstantValue
```

12.4 Package declaration

Package declarations are used to describe the external services and exceptions used by the interchanged script.

A package declaration shall consist of

- a package identifier (optional);
- a string representing the package name;
- a sequence of service descriptions;
- a sequence of exception descriptions.

```
PackageDeclaration      ::=   PackageIdentifier?
                              VisibleString     // Package name
                              ServiceDescription*
                              ExceptionDescription*
```

12.4.1 Package identifier

Package identifiers are used to reference packages throughout the interchanged script.

The package identifier shall be a positive integer within the range allowed for packages. It shall correspond to the index (starting at 0) of the declaration in the package declarations part.

If the package identifier is not provided, it shall be computed by the script parser.

```
PackageIdentifier       ::=   INTEGER
```

12.4.2 Name

A package name is used by the script interpreter to access the package within the run-time environment, according to the **package availability** procedure described by the platform mapping specification.

12.4.3 Service description

Service descriptions describe external function prototypes.

A service description shall consist of

- a function identifier (optional);
- a string representing the operation name;
- a calling mode (optional);
- a type identifier (optional);
- a sequence of parameter descriptions.

```
ServiceDescription      ::=   FunctionIdentifier?
                              VisibleString?    // IDL global name
                              CallingMode?
                              TypeIdentifier?   // return value
                              ServiceParameterDescription*
```

12.4.3.1 Function identifier

Function identifiers are used to reference functions throughout the interchanged script.

The function identifier shall be a positive integer within the range allowed for services. It shall correspond to the maximum number of routines plus the maximum number of predefined functions plus the package

identifier multiplied by 256, incremented by the index (starting from 0) of the service in the package declaration.

If the function identifier is not provided, it shall be computed by the script parser.

```
FunctionIdentifier              ::=    INTEGER
```

12.4.3.2 Name

The operation name is used by the script interpreter to access the operation within the run-time environment, according to the **operation invocation** procedure described by the platform mapping specification.

12.4.3.3 Calling mode

The calling mode represents the way the operation shall be invoked.

The calling mode shall be either 'synchronous' or 'asynchronous'.

If the value is not specified, the calling mode shall be 'synchronous'.

```
CallingMode                     ::=    'SYNCHRONOUS' | 'ASYNCHRONOUS'
```

12.4.3.4 Type identifier

The type identifier represents the type of return value of the service.

If the type identifier is not specified, it shall be interpreted as a `void` type, i.e. the function shall have no return value.

If the calling mode of the operation is 'asynchronous', the type identifier shall be either 'void' or not specified.

12.4.3.5 Parameter description

Parameter descriptions are used to specify the type and passing mode of service parameters.

A parameter description shall consist of

- a passing mode;
- a type identifier.

```
ServiceParameterDescription ::= ServicePassingMode?
                                TypeIdentifier
```

12.4.3.5.1 Passing mode

The passing mode indicates whether the value of the parameter at the time of invocation of the service is used by the service (input parameter) and whether this parameter is modified by the service for use by its caller (output parameter).

The passing mode shall be one of the following: 'in', 'inout' or 'out'.

If the passing mode is not specified, it shall be interpreted as an **in** parameter.

NOTE: The object reference parameter is implicit, so it should not be specified as part of the declaration. It is dealt with as an **in** parameter.

If the calling mode of the operation is 'asynchronous', the passing mode shall be either 'in' or not specified.

```
ServicePassingMode      ::=   'IN' | 'OUT' | 'INOUT'
```

12.4.3.5.2 Type identifier

The type identifier represents the type of the considered service parameter.

12.4.4 Exception description

Exception descriptions describe prototypes of exceptions that may be raised during the execution of external functions.

An exception description shall consist of

- a message identifier (optional);
- a string representing the exception name;
- a sequence of type identifiers representing the members of the exception.

```
ExceptionDescription    ::=   MessageIdentifier?
                              VisibleString?     //IDL exception global name
                              TypeIdentifier*    //Parameter types
```

12.4.4.1 Message identifier

Message identifiers are used to reference messages throughout the interchanged script.

The message identifier shall be a positive integer within the range allowed for exceptions. It shall correspond to the maximum number of predefined messages plus the package identifier multiplied by 256, incremented by the index (starting at 0) of the exception in the package declaration.

If the message identifier is not provided, it shall be computed by the script parser.

```
MessageIdentifier       ::=   INTEGER
```

12.4.4.2 Name

An exception name is used by the script interpreter to retrieve the exception within the run-time environment, according to the **exception retrieval** procedure described by the platform mapping specification.

12.4.4.3 Parameter description

Each parameter of the message corresponds to one member of the exception. It is described by its type identifier.

12.5 Handler declaration

Handler declarations are used to associate a message with the function that handles it.

A handler declaration shall consist of

- a message identifier;
- a function identifier.

```
HandlerDeclaration      ::=   MessageIdentifier
                              FunctionIdentifier
```

12.5.1 Message identifier

The message identifier indicates the message to be handled.

The message identifier shall be a positive integer within the whole range allowed to messages, representing either a predefined message or an exception.

12.5.2 Function identifier

The function identifier indicates the function to be triggered when the message is removed from the message queue.

The function identifier shall be a positive integer within the whole range allowed to function, representing a routine, a predefined function or a service.

The description of the formal parameter types for the function shall be the same as for the message, so that the function may be called with the message actual parameters as its parameters. If signatures do not match, the handler shall be rejected by the script parser.

12.6 Routine declaration

Routine declarations are used to describe the structure and program code of the internal functions of the interchanged script.

A routine declaration shall consist of

- a function identifier (optional);
- a type identifier (optional);
- a sequence of parameter descriptions;
- a sequence of local variable declarations;
- MHEG-SIR program code.

```
RoutineDeclaration          ::=   FunctionIdentifier?
                                  TypeIdentifier?          // for return value
                                  RoutineParameterDescription*
                                  LocalVariableDeclaration*
                                  OCTET STRING             // program code
```

12.6.1 Function identifier

The function identifier shall be a positive integer within the range allowed for routines. It shall correspond to the index (starting from 0) of the routine in the routine declarations part.

If the function identifier is not provided, it shall be computed by the script parser.

12.6.2 Type identifier

The type identifier represents the type of return value of the routine.

If the type identifier is not specified, it shall be interpreted as a `void` type, i.e. the function shall have no return value.

12.6.3 Parameter description

Parameter descriptions are used to specify the type and passing mode of routine parameters.

A parameter description shall consist of

- a passing mode (optional);
- a type identifier.

```
RoutineParameterDescription ::= RoutinePassingMode?
                                TypeIdentifier
```

12.6.3.1 Passing mode

The passing mode indicates whether the parameter shall be passed to the routine using its **value** (input parameter) or a **reference** to the variable that holds its value (input/output parameter).

The passing mode shall be one of the following: 'value' or 'reference'.

```
RoutinePassingMode      ::= 'VALUE' | 'REFERENCE'
```

12.6.3.2 Type identifier

The type identifier represents the type of the considered routine parameter.

12.6.4 Local variable declaration

Local variable declarations are used to describe the types and initial values of variables whose scope is limited to one execution of a routine.

A local variable declaration shall have the same structure as a global variable declaration, as defined in subclause 12.3. It shall consist of

- a data identifier (optional);
- a type identifier;
- a constant reference (optional).

12.6.4.1 Data identifier

The data identifier shall be a positive integer within the range allowed for local variables. It shall correspond to the maximum number of constants plus the maximum number of global variables incremented by the index (starting from 0) of the declaration in the local variable declarations of the routine, incremented by the number of formal parameters of the routine.

If the data identifier is not provided, it shall be computed by the script parser.

12.6.4.2 Type identifier

The type identifier represents the type to which the value of the local variable belongs.

12.6.4.3 Constant reference

The constant reference represents the initial value of the local variable.

The constant reference shall be one of the following:

- a data identifier referencing a constant;
- a constant value as defined in subclause 12.2.3.

In any case, the value to which this constant reference refers shall be of the type of the local variable.

If the constant reference is not provided, the script interpreter shall assign the local variable a default value if its type allows for it. Otherwise, it shall remain undefined until assigned by an instruction.

12.6.5 Program code

The program code consists of the sequence of instructions of the routine, intended for execution by the script interpreter when the routine is triggered. The syntax and semantics of the MHEG-SIR instructions is described in Clause 13.

The last instruction of a routine shall be a `RET` instruction.

13 MHEG-SIR instructions

This Clause defines the semantics of the MHEG-SIR instructions.

13.1 Presentation methodology

Each instruction is defined in the corresponding subclause by a set of entries as follows:

Short description:	A brief description of the instruction's semantics.
Synopsis:	`Mnemonic Operand1 ... OperandN`
Operands:	A description of the types and semantics of each operand carried with the instruction (if any).
Stack:	A visual synopsis of the instruction's effect on the parameter stack, e.g. `..., Parameter1, Parameter2 ⇨ ..., Result`
Types:	A list of the types of parameters to which the instruction applies (if it is a template instruction).
Parameters:	A description of the semantics of each element of the parameter stack which is popped, pushed or otherwise effected by the instruction (if any).
Effect:	A textual specification of the interpretation semantics of the instruction.
Formal specification:	A formal specification of the interpretation semantics of the instruction using the notation described in this subclause.
Errors:	A list of the errors that may be raised during execution of the instruction.

13.1.1 Error conditions

The semantics of the instruction, as described by the formal specification, shall apply only if the operands are valid. Otherwise, an `InstructionExecutionError` exception shall be raised and the error register shall be set to `InvalidOperand`. The result of the instruction execution is unspecified.

When the parameter stack is looked up, i.e. on a `PS.pop` or a `PS[SP]` primitive, if the parameter stack does not hold enough parameters then an `InstructionExecutionError` exception shall be raised and the error register shall be set to `StackUnderflow`. The resulting state of the parameter stack is unspecified.

If the result of an arithmetic operation falls in a range that exceeds that of the target type, arithmetic operations shall raise an `InstructionExecutionError` exception and the error register shall be set to `ArithmeticOverflow` or `DivisionByZero`, as applicable.

If a identifier does not refer to a valid entity (type, data, function, message, package) then when its value in the corresponding table is accessed (e.g. using `DT[i]`), an `InstructionExecutionError` exception shall be raised and the error register shall be set to `InvalidIdentifier`.

If IP is set to an invalid pointer then an `InstructionExecutionError` exception shall be raised and the error register shall be set to `JumpOutOfRange`.

When a dynamic variable is allocated and allocation is impossible due to lack of memory space or data identifiers then the `new()` primitive shall raise an `InstructionExecutionError` exception and set the error register to `AllocationFailed`.

Triggering of the other error conditions is specified explicitly throughout subclause 13.3. The error code values are defined in Annex C.

13.1.2 Formal specification

The "formal specification" entry of an instruction description gives a concise formal notation of the effect that the instruction execution unit shall produce as it interprets the instruction; however, as this specification is expressed in terms of a sequence of operations, there may be other methods to lead to the same result, so this formal specification does not require the instruction execution unit to perform as expressed as long as the effect is the same.

The error cases described in subclause 13.1.1 are implicit and are not expressed in the formal specification. The other error cases are explicitly mentioned.

To specify the semantics of an instruction in a formal way, a C-like syntax is used. It uses the notations and concepts defined in Clauses 8 and 9, plus the following notations:

- data table (DT) notation (see subclause 13.1.3);
- template instruction notation (see subclause 13.1.4);
- primitives (see subclause 13.1.5).

13.1.3 Data table notation

The notation `DT(i)`, where `i` stands for a data identifier, corresponds to:

- the entry whose key is `i` in the constant table, if `i` is the data identifier of a constant;
- the entry whose key is `i` in the global variable table, if `i` is the data identifier of a global variable;
- the entry whose key is `i` in the local variable table of the currently executing routine, if `i` is the data identifier of a local variable;
- the dynamic variable whose handle is mapped to `i`, if `i` is the data identifier of a dynamic variable.

This macro may be expressed as follows:

```
#define DT(i) (i < 4096) ? CT[i] : VT[i])
```

13.1.4 Template instruction notation

A number of instructions (e.g. arithmetic and logical instruction) operate on values of a given type and produce a result with the same type. The `<T>` notation is used to express a template instruction. `<Mnemonic>_<T>` represents all instructions `<Mnemonic>` with `<T>` being replaced by the type letter of any primitive type to which the instruction is applicable, as described by the "Types" entry in the instruction description.

NOTE: Operations on mixed types should be handled by explicitly inserting type conversion instructions in the instruction sequence.

13.1.5 Primitives

The following primitive notations are used in the formal specification of the instructions:

- `DID new(tid)`: allocates a dynamic variable of the type identifier by `tid`;
- `void raise(exc)`: raises an `InstructionExecutionError` exception and sets ER to error code `exc`;
- `void delete(did)`: releases the dynamic variable identified by `did`;
- `int sizeof(tid)`: returns the size of values of the type identified by `tid`, expressed in the same units as the PS pointer addresses;
- `type(<T>)`: macro to be replaced by the C type name.

13.2 Classification of MHEG-SIR instructions

The MHEG-SIR instructions may be clustered into categories according to their effect on the control flow, on the variable tables or on the parameter stack, and according to the types of stack parameters that they accept:

a) instructions that affect the control flow:
   1) unconditional jump instructions: `JMP`, `LJMP`;
   2) conditional jump instructions: `JT`, `JF`, `LJT`, `LJF`;
   3) function invocations: `CALL`, `XCALL`;
   4) miscellaneous control flow instructions: `RET`, `YIELD`;

b) instructions that do not affect the control flow, but affect the value of variables:
   1) complex variable modifiers: `SET`, `SETC`;
   2) arithmetic operators on variables: `INC`, `DEC`;
   3) stack pop instructions: `POPR`, `POP`, `POPC`;
   4) memory management instructions: `ALLOC`, `FREE`;

c) instructions that do not affect the control flow or the variables, but affect the parameter stack:
   1) conversion instructions: `CVT`;
   2) arithmetic operators: `ADD`, `SUB`, `MUL`, `DIV`, `REM`, `NEG`;
   3) logical operators: `AND`, `OR`, `XOR`, `NOT`;
   4) logical shift operators: `SHIFT`;
   5) comparison operators: `EQ`, `GT`, `LT`, `EQR`;
   6) complex data accessors: `GET`, `GETC`;
   7) miscellaneous stack manipulation instructions: `PUSHI`, `PUSHR`, `PUSH`, `DUP`, `GETOR`;

d) instructions that have no effect: `NOP`.

NOTE: Most instructions only operate on primitive type values. Only the following instructions are used to manipulate constructed values: `EQR`, `GET`, `GETC`, `SET`, `SETC`, `ALLOC`, `FREE`, `CALL`, `XCALL`.

The effect of instructions is summarised in Table 3. The operations are listed in canonical order, i.e. by ascending op-code number. Some mnemonics represent template instructions and therefore have type suffixes.

Table 3: Synopsis of MHEG-SIR instructions and their effect

Mnemonics	Ref.	Opcode (hexa)	Op. size	Op. type	Parameter types	PS effect	VT effect	Control flow effect
NOP	14.3.1	00	0					
YIELD	14.3.2	02	0					x
RET	14.3.3	03	0			0|1 ⇨ 0|1		x
FREE	14.3.4	08	0			1 ⇨ 0	x	
NOT_<T>	14.3.5	10 – 13	0		BOWU	1 ⇨ 1		
OR_<T>	14.3.6	14 – 17	0		BOWU	2 ⇨ 1		
XOR_<T>	14.3.7	18 – 1B	0		BOWU	2 ⇨ 1		
AND_<T>	14.3.8	1C – 1F	0		BOWU	2 ⇨ 1		
EQR	14.3.9	20	0			2 ⇨ 1		
EQ_<T>	14.3.10	21 – 2B	0		OSLWUFDBCIR	2 ⇨ 1		
LT_<T>	14.3.11	30 – 37	0		COSLWUFD	2 ⇨ 1		
GT_<T>	14.3.12	38 – 3F	0		COSLWUFD	2 ⇨ 1		
ADD_<T>	14.3.13	40 – 47	0		OSLWUFD	2 ⇨ 1		
SUB_<T>	14.3.14	48 – 4F	0		OSLWUFD	2 ⇨ 1		
MUL_<T>	14.3.15	50 – 57	0		OSLWUFD	2 ⇨ 1		
DIV_<T>	14.3.16	58 – 5F	0		OSLWUFD	2 ⇨ 1		
NEG_<T>	14.3.17	62 – 67	0		SLFD	1 ⇨ 1		
REM_<T>	14.3.18	79 – 7D	0		OSLWU	2 ⇨ 1		
DUP_<T>	14.3.19	81 – 8B	0		OSLWUFDBCIR	1 ⇨ 2		
CVT_<TT>	14.3.20	94 – BE	0		OSLWUFDBC	1 ⇨ 1		
JT	14.3.21	C0	1	offset		1 ⇨ 0		x
JF	14.3.22	C1	1	offset		1 ⇨ 0		x
JMP	14.3.23	C2	1	offset				x
SHIFT_<T>	14.3.24	C5 – C7	1	offset	OWU	1 ⇨ 1		
GETOR	14.3.25	C9	1	PID		0 ⇨ 1		
LJT	14.3.26	D0	2	offset		1 ⇨ 0		x
LJF	14.3.27	D1	2	offset		1 ⇨ 0		x
LJMP	14.3.28	D2	2	offset				x
CALL	14.3.29	D4	2	FID		n ⇨ 0|1		x
XCALL	14.3.30	D6	2	FID		n ⇨ 0|1		x
PUSH	14.3.31	E0	2	DID		0 ⇨ 1		
PUSHR	14.3.32	E1	2	DID		0 ⇨ 1		
PUSHI	14.3.33	E3	2	value		0 ⇨ 1		
POP	14.3.34	E4	2	DID		1 ⇨ 0	x	
POPR	14.3.35	E5	2	DID		1 ⇨ 0	x	
POPC	14.3.36	E6	2	DID		1 ⇨ 0	x	
ALLOC	14.3.37	E8	2	TID		0 ⇨ 1	x	
INC	14.3.38	EA	2	DID		1 ⇨ 0	x	
DEC	14.3.39	EB	2	DID		1 ⇨ 0	x	
GET	14.3.40	F0	3	DID, idx		idx ⇨ 1		
GETC	14.3.41	F2	3	DID, idx		idx+1 ⇨ 0	x	
SET	14.3.42	F4	3	DID, idx		idx+1 ⇨ 0	x	
SETC	14.3.43	F6	3	DID, idx		idx+1 ⇨ 0	x	

13.3 Description of instructions

13.3.1 No operation

Short description: Do nothing.

Synopsis: `NOP`

Operands: None.

Types: Not applicable.

Parameters: None.

Stack: ... ⇨ ...

Effect: None.

Formal specification:
```
0;
```

Errors:

13.3.2 Yield

Short description: Handle pending messages.

Synopsis: `YIELD`

Operands: None.

Stack: ... ⇨ ...

Types: Not applicable.

Parameters: None.

Effect: If there is a pending message in the message queue, handle it by calling the corresponding routine.
Upon returning, iterate the process until the message queue is empty.

Formal specification:
```
while (QP != 'null')
{
   FID fid = HT[MQ[QP].MID].FID;
   if (fid == 'null') then raise('HandlerNotFound');
   else
   {
        CS.push({IP-1, FR, SP, LT});
        // IP-1: allows to re-iterate the YIELD instruction
        FR = fid;
        IP = RT[FR].IP;
        LT = MQ[QP].LT;
   }
   MQ.remove();
}
```

Errors: `HandlerNotFound`

13.3.3 Return

Short description: Return to caller.

Synopsis: `RET`

Operands: None.

Stack: `..., (Val) ⇨ ..., (Val)`

Types: Not applicable.

Parameters: If the current routine signature has a return value, `Val` shall be interpreted as of the type of this return value.
Otherwise, no stack parameter shall be considered.

Effect: Return to the calling routine. Pop the call stack and restore the context of the previous frame. If the current routine has a return value, check that there is a value of the same type on the top of the parameter stack.
If there is no calling function to return to, stop and go back to **ready** status.

Formal specification:

```
if (sizeof(RT[FR].TID) != (SP - CS[FP].SP))
        then raise('InvalidReturnValue');
IP = CS[FP].IP;
FR = CS[FP].FR;
LT = CS[FP].LT;
CS.pop();
```

Errors: `InvalidReturnValue`

13.3.4 Free

Short description: Release dynamic variable.

Synopsis: `FREE`

Operands: None

Stack: `..., Did ⇨ ...`

Types: Not applicable.

Parameters: `Did` shall be interpreted as a data identifier.

Effect: Check that `Did` is the data identifier of a dynamic variable. Release the dynamic memory associated with `Did`, and make the data identifier invalid.

Formal specification:

```
if (did < 8100h) then raise('InvalidParameter');
delete(VT[PS.pop('data identifier')]);
```

Errors:
`StackUnderflow`
`InvalidIdentifier`

13.3.5 Not

Short description: Logical negation.

Synopsis: `NOT_<T>`

Operands: None.

Stack: `..., Val ⇨ ..., Neg`

Types: `Boolean` or any unsigned integer (B, O, W, U).

Parameters: `Val` shall be interpreted as of type `<T>`.
`Neg` shall be of type `<T>`.

Effect: Replace the top element of the parameter stack by its logical negation if `<T>` is `B`, its bitwise negation otherwise (i.e. its complement-to-one):

```
Neg = ~Val
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
if (<T> == 'boolean') then PS.push(! buf);
else PS.push(~ buf);
```

Errors: `StackUnderflow`

13.3.6 Or

Short description: Logical disjunction.

Synopsis: `OR_<T>`

Operands: None.

Stack: `..., Val1, Val2 ⇨ ..., Disj`

Types: `Boolean` or any unsigned integer (B, O, W, U).

Parameters: `Val1` and `Val2` shall be interpreted as of type `<T>`.
`Disj` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their logical disjunction if `<T>` is `B`, their bitwise disjunction otherwise:

```
Disj = Val1 | Val2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
if (<T> == 'boolean') then buf = buf || PS.pop('boolean');
else buf |= PS.pop(<T>);
PS.push(buf);
```

Errors: `StackUnderflow`

13.3.7 Exclusive or

Short description: Logical exclusion.

Synopsis: `XOR_<T>`

Operands: None.

Stack: `..., Val1, Val2 ⇨ ..., Excl`

Types: `Boolean` or any unsigned integer (B, O, W, U).

Parameters: `Val1` and `Val2` shall be interpreted as of type `<T>`.
`Excl` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their logical exclusion if <T> is B, their bitwise exclusion otherwise:

```
Excl = Val1 ^ Val2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
if (<T> == 'boolean') then buf = (buf != PS.pop('boolean'));
else buf ^= PS.pop(<T>);
PS.push(buf);
```

Errors: StackUnderflow

13.3.8 And

Short description: Logical conjunction.

Synopsis: AND_<T>

Operands: None.

Stack: ..., Val1, Val2 ⇨ ..., Conj

Types: Boolean or any unsigned integer (B, O, W, U).

Parameters: Val1 and Val2 shall be interpreted as of type <T>.
Conj shall be of type <T>.

Effect: Replace the top two elements of the parameter stack by their logical conjunction if <T> is B, their bitwise conjunction otherwise:

```
Conj = Val1 & Val2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
if (<T> == 'boolean') then buf = buf && PS.pop('boolean');
else buf &= PS.pop(<T>);
PS.push(buf);
```

Errors: StackUnderflow

13.3.9 Equal reference

Short description: Compare constructed values.

Synopsis: EQR

Operands:

Stack: ..., Did1, Did2 ⇨ ..., Bool

Types: Not applicable.

Parameters: Did1 and Did2 shall be interpreted as of data identifier type.
Bool shall be of boolean type.

Effect: Check that Did1 and Did2 identify data of the same type.
Return 'true' if the data identified by Did1 and Did2 are equal (see subclause 8.2), 'false' otherwise:

```
Bool = (DT(Did1) == DT(Did2))
```

Formal specification:
```
DID did2 = PS.pop('data identifier');
DID did1 = PS.pop('data identifier');
if (DT(did1).tid != DT(did2).tid) then raise('TypeMismatch');
if (DT(did1).val == DT(did2).val) then PS.push('true');
else PS.push('false');
```

Errors: `TypeMismatch`
`StackUnderflow`
`InvalidIdentifier`

13.3.10 Equal

Short description: Equality.

Synopsis: `EQ_<T>`

Operands: None.

Stack: `..., Val1, Val2 ⇨ ..., Comp`

Types: Any primitive type except `void` (O, S, L, W, U, F, D, B, C, I, R)

Parameters: `Val1` and `Val2` shall be interpreted as of type `<T>`.
`Comp` shall be of `boolean` type.

Effect: Replace the top two elements of the parameter stack by 'true' if they are equal and 'false' otherwise:
```
Comp = (Val1 == Val2)
```

Formal specification:
```
type(<T>) buf = PS.pop(<T>);
if (buf == PS.pop<T>) then PS.push('true');
else PS.push('false');
```

Errors: `StackUnderflow`

13.3.11 Less than

Short description: Strict inferiority.

Synopsis: `LT_<T>`

Operands: None.

Stack: `..., Val1, Val2 ⇨ ..., Comp`

Types: `Character` or any numeric (C, O, S, L, W, U, F, D).

Parameters: `Val1` and `Val2` shall be interpreted as of type `<T>`.
`Comp` shall be of `boolean` type.

Effect: Replace the top two elements of the parameter stack by 'true' if the top element is greater than the next, and 'false' otherwise:
```
Comp = (Val1 < Val2)
```
To compare characters, the numeric order shall be used.

Formal specification:
```
type(<T>) buf = PS.pop(<T>);
if (PS.pop<T> < buf) then PS.push('true');
else PS.push('false');
```

Errors: `StackUnderflow`

13.3.12 Greater than

Short description: Strict superiority.

Synopsis: `GT_<T>`

Operands: None.

Stack: `..., Val1, Val2 ⇨ ..., Comp`

Types: `Character` or any numeric (C, O, S, L, W, U, F, D).

Parameters: `Val1` and `Val2` shall be interpreted as of type `<T>`.
`Comp` shall be of `boolean` type.

Effect: Replace the top two elements of the parameter stack by 'true' if the top element is less than the next, and 'false' otherwise:

```
Comp = (Val1 > Val2)
```

To compare characters, the numeric order shall be used.

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
if (PS.pop<T> > buf) then PS.push('true');
else PS.push('false');
```

Errors: `StackUnderflow`

13.3.13 Add

Short description: Arithmetic addition.

Synopsis: `ADD_<T>`

Operands: None.

Stack: `..., Num1, Num2 ⇨ ..., Sum`

Types: Any numeric (O, S, L, W, U, F, D).

Parameters: `Num1` and `Num2` shall be interpreted as of type `<T>`.
`Sum` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their sum:

```
Sum = Num1 + Num2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
buf += PS.pop(<T>);
PS.push(buf);
```

Errors: `StackUnderflow`
`ArithmeticOverflow`

13.3.14 Subtract

Short description: Arithmetic subtraction.

Synopsis: `SUB_<T>`

Operands: None.

Stack: `..., Num1, Num2 ⇨ ..., Diff`

Types: Any numeric (O, S, L, W, U, F, D).

Parameters: `Num1` and `Num2` shall be interpreted as of type `<T>`.
`Diff` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their difference:

```
Diff = Num1 - Num2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
buf = PS.pop(<T>) - buf;
PS.push(buf);
```

Errors: `StackUnderflow`
`ArithmeticOverflow`

13.3.15 Multiply

Short description: Arithmetic multiplication.

Synopsis: `MUL_<T>`

Operands: None.

Stack: `..., Num1, Num2 ⇨ ..., Prod`

Types: Any numeric (O, S, L, W, U, F, D).

Parameters: `Num1` and `Num2` shall be interpreted as of type `<T>`.
`Prod` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their product:

```
Prod = Num1 * Num2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
buf *= PS.pop(<T>);
PS.push(buf);
```

Errors: `StackUnderflow`
`ArithmeticOverflow`

13.3.16 Divide

Short description: Arithmetic division.

Synopsis: `DIV_<T>`

Operands: None.

Stack: `..., Num1, Num2 ⇨ ..., Quot`

Types: Any numeric (O, S, L, W, U, F, D).

Parameters: `Num1` and `Num2` shall be interpreted as of type `<T>`.
`Quot` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their quotient:

```
Quot = Num1 / Num2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
buf = PS.pop(<T>) / buf;
PS.push(buf);
```

Errors:

```
StackUnderflow
DivisionByZero
```

13.3.17 Negate

Short description: Sign change.

Synopsis:

```
NEG_<T>
```

Operands: None.

Stack:

```
..., Num ⇨ ..., Opp
```

Types: Any signed numeric (S, L, F, D).

Parameters: `Num` shall be interpreted as of type `<T>`.
`Opp` shall be of type `<T>`.

Effect: Replace the top element of the parameter stack by its opposite:

```
Opp = -Num1
```

Formal specification:

```
type(<T>) buf = PS.pop<T>;
PS.push(- buf);
```

Errors:

```
StackUnderflow
```

13.3.18 Remainder

Short description: Arithmetic remainder.

Synopsis:

```
REM_<T>
```

Operands: None.

Stack:

```
..., Num1, Num2 ⇨ ..., Rem
```

Types: Any integer (O, S, L, W, U).

Parameters: `Num1` and `Num2` shall be interpreted as of type `<T>`.
`Rem` shall be of type `<T>`.

Effect: Replace the top two elements of the parameter stack by their remainder:

```
Rem = Num1 % Num2
```

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
buf = PS.pop(<T>) % buf;
PS.push(buf);
```

Errors:

```
StackUnderflow
DivisionByZero
```

13.3.19 Duplicate

Short description: Duplicate value

Synopsis: `DUP_<T>`

Operands: None

Stack: `..., Val ⇨ ..., Val, Val`

Types: Any primitive type except `void` (O, S, L, W, U, F, D, B, C, I, R)

Parameters: `Val` shall be interpreted as of type `<T>`.

Effect: Duplicate the value on the top of stack.

Formal specification:
```
type(<T>) buf = PS[SP](<T>);
PS.push(buf);
```

Errors: `StackUnderflow`

13.3.20 Convert

Short description: Convert value

Synopsis: `CVT_<T1><T2>`

Operands: None

Stack: `..., Val ⇨ ..., Res`

Types: `Boolean`, `character` or any numeric (O, S, L, W, U, F, D, B, C); see allowed combinations in subclause 13.4.

Parameters: `Val` shall be interpreted as of type `<T1>` (source type).
`Res` shall be of type `<T2>` (destination type).

Effect: Replace the value on the top of stack by an equivalent value in the destination type. Conversion rules defined in subclause 13.4 apply.

Formal specification:
```
type(<T2>) buf = (type(<T2>)) (PS.pop(<T1>));
PS.push(buf);
```

Errors: `StackUnderflow`

13.3.21 Jump on true

Short description: "If" conditional short jump.

Synopsis: `JT Off`

Operands: `Off` shall be a one-byte signed offset (in complement-to-two notation) specifying the number of instructions to move forwards or backwards within the current routine.

Stack: `..., Test ⇨ ...`

Types: Not applicable.

Parameters: `Test` shall be interpreted as of `boolean` type.

Effect: If the top element of the stack is 'true' then
if `Off` is positive, jump `Off` instructions forwards;
if `Off` is negative, jump `-Off` instructions backwards.

Formal specification:
```
if (PS.pop('boolean'))  then IP += Off;
```

Errors: `StackUnderflow`
`JumpOutOfRange`

13.3.22 Jump on false

Short description: "Else" conditional short jump.

Synopsis: `JF Off`

Operands: `Off` shall be a one-byte signed offset (in complement-to-two notation) specifying the number of instructions to move forwards or backwards within the current routine.

Stack: `..., Test ⇨ ...`

Types: Not applicable.

Parameters: `Test` shall be interpreted as of `boolean` type.

Effect: If the top element of the stack is 'false' then
if `Off` is positive, jump `Off` instructions forwards;
if `Off` is negative, jump `-Off` instructions backwards.

Formal specification:
```
if ! (PS.pop('boolean'))  then IP += Off;
```

Errors: `StackUnderflow`
`JumpOutOfRange`

13.3.23 Jump

Short description: Unconditional short jump.

Synopsis: `JMP Off`

Operands: `Off` shall be a one-byte signed offset (in complement-to-two notation) specifying the number of instructions to move forwards or backwards within the current routine.

Stack: `... ⇨ ...`

Types: Not applicable.

Parameters: None.

Effect: If `Off` is positive, jump `Off` instructions forwards;
if `Off` is negative, jump `-Off` instructions backwards.

Formal specification: `IP += Off;`

Errors: `JumpOutOfRange`

13.3.24 Shift

Short description: Logical shift.

Synopsis: `SHIFT_<T> Off`

Operands: `Off` shall be a one-byte signed offset (in complement-to-two notation) specifying the number of bit places to shift the parameter leftwards or rightwards.

Stack: `..., Val ⇨ ..., Pwr`

Types: Any unsigned integer (O, W, U).

Parameters: `Val` shall be interpreted as of type `<T>`.
`Pwr` shall be of type `<T>`.

Effect: Replace the top element of the stack by its value shifted right `Off` bits if `Off` is positive, or left `-Off` bits if `Off` is negative. If `Off` is beyond range, the result is unspecified.

Formal specification:

```
type(<T>) buf = PS.pop(<T>);
if (Off >=0) then buf >>= Off;
else if (buf < 0)
        buf = -(-buf << -Off);
else buf <<= -Off;
PS.push(buf);
```

Errors: `StackUnderflow`
`ShiftOutOfRange`

13.3.25 Get object reference

Short description: Get initial object reference to package.

Synopsis: `GETOR Pid`

Operands: `Pid` shall be the one-byte representation of a package identifier specifying the package to access.

Stack: `... ⇨ ..., Obref`

Types: Not applicable.

Parameters: `Obref` shall be of `object reference` type.

Effect: Retrieve an object reference to the initial object of the package.

Formal specification:

```
if (PT[PID].sts = 'not available') then raise('BadPackageStatus');
PS.push(PT[PID].or);
```

Errors: `InvalidIdentifier`
`BadPackageStatus`

13.3.26 Long jump on true

Short description: "If" conditional long jump.

Synopsis: `LJT Off`

Operands: `Off` shall be a two-byte signed offset (in complement-to-two notation) specifying the number of instructions to move forwards or backwards within the current routine.

Stack: `..., Test ⇨ ...`

Types: Not applicable.

Parameters: `Test` shall be interpreted as of `boolean` type.

Effect: If the top element of the stack is 'true' then
if `Off` is positive, jump `Off` instructions forwards;
if `Off` is negative, jump `-Off` instructions backwards.

Formal specification:
```
if (PS.pop('boolean'))  then IP += Off;
```

Errors: `StackUnderflow`
`JumpOutOfRange`

13.3.27 Long jump on false

Short description: "Else" conditional long jump.

Synopsis: `LJF Off`

Operands: `Off` shall be a two-byte signed offset (in complement-to-two notation) specifying the number of instructions to move forwards or backwards within the current routine.

Stack: `..., Test ⇨ ...`

Types: Not applicable.

Parameters: `Test` shall be interpreted as of `boolean` type.

Effect: If the top element of the stack is 'false' then
if `Off` is positive, jump `Off` instructions forwards;
if `Off` is negative, jump `-Off` instructions backwards.

Formal specification:
```
if ! (PS.pop('boolean'))  then IP += Off;
```

Errors: `StackUnderflow`
`JumpOutOfRange`

13.3.28 Long jump

Short description: Unconditional long jump.

Synopsis: `LJMP Off`

Operands:	`Off` shall be a two-byte signed offset (in complement-to-two notation) specifying the number of instructions to move forwards or backwards within the current routine.
Stack:	`...` ⇨ `...`
Types:	Not applicable.
Parameters:	None.
Effect:	If `Off` is positive, jump `Off` instructions forwards; if `Off` is negative, jump `-Off` instructions backwards.
Formal specification:	`IP += Off;`
Errors:	`JumpOutOfRange`

13.3.29 Call

Short description:	Call routine.
Synopsis:	`CALL Fid`
Operands:	`Fid` shall be the two-byte representation of a function identifier specifying the routine to invoke.
Stack:	`..., ParN, ... , Par1` ⇨ `...`
Types:	Not applicable.
Parameters:	`Par1`, ..., `ParN` (where N is the number of parameters of the routine) are the actual parameters of the routine. They shall be interpreted as of the same type as the formal parameters of the routine when those are passed by value, and they shall be interpreted as of `data identifier` type and reference a variable of the same type of the formal parameters of the routine when those are passed by reference.
Effect:	Pop the top elements of the parameter stack and invoke the routine specified by Fid with these elements as actual parameters. For parameters passed by reference, check that the data identifier does not reference a local variable and points to data of the same type as in the signature. Push one frame onto the call stack with the current context. Initialise the local variable table for the routine. Set the instruction pointer to the first instruction of the routine.

Formal specification:

```
TID tid;
CS.push(IP, FR, SP, LT);
FR = Fid;
LT = RT[Fid].LT;
for (short i = 0; i < RT[Fid].nbp; i--;)
{
        switch (RT[Fid].sig[i].mod)
        {
        case 'value':
                tid = RT[Fid].sig[i].TID;
                break;
        case 'reference':
                tid == 'data identifier';
                if (8000h <= PS[SP](tid) < 8100h)
                then raise('InvalidParameter');
                if (RT[Fid].sig[i].TID != DT(PS[SP](tid)).TID)
                then raise('TypeMismatch');
                break;
        };
        LT[i+0x8000].val = PS.pop(tid);
};
IP = RT[Fid].IP;
```

Errors:

```
InvalidIdentifier
StackUnderflow
TypeMismatch
InvalidParameter
```

13.3.30 External call

Short description: Call external function.

Synopsis: `XCALL Fid`

Operands: `Fid` shall be the two-byte representation of a function identifier specifying the service or predefined function to invoke.

Stack: `..., ParN, ..., Par1 ⇨ ..., (Ret)`

Types: Not applicable.

Parameters: `Par1` shall be interpreted as of `object reference` type. It indicates the reference of the IDL instance to which to apply the operation.
`Par2, ..., ParN` (where N is the number of parameters of the function, plus 1) are the actual parameters of the function. Whatever the passing mode, they shall be interpreted as of `data identifier` type.
If the function has a return value type other than `void`, `Ret` shall be of this type.

Effect: Check that the parameters reference data of the same type as in the function's signature. Pop the top elements of the parameter stack and invoke the external function specified by Fid with these elements as actual parameters. Push one frame onto the call stack with the current context. Pass parameters to and invoke the external function. If the invocation is asynchronous, pop the call stack as soon as the request is acknowledged. If the invocation is synchronous, wait for completion of the request. If an exception is raised, activate the handler of the exception. Otherwise, retrieve the function's output parameters and return value, push the return value onto the parameter stack and pop the call stack.

Formal specification:

```
DID buf[ST[Fid].nbp];
Object obref = PS.pop('object reference');
for (short i = 0; i < ST[Fid].nbp; i++;)
{
        if (ST[Fid].sig[i].TID != DT(PS[SP]('data identifier').TID))
        then raise('TypeMismatch');
        buf[i] = PS.pop('data identifier');
};
CS.push(IP, FR, SP, LT);
FR = Fid;
LT[0].tid = 'object reference';
LT[0].val = obref;
for (short i = 1;i < ST[Fid].nbp; i++}
        LT[i].TID = 'data identifier';
        LT[i].val = buf[i];
}
short Pid = (Fid>>8)-64;
if (PT[Pid].sts != 'available') then raise('BadPackageStatus');
_open_package(PT[Pid].name);
// open a context to invoke the service
_pass_in_parameter(LT[0]);
// according to the platform mapping specification procedure
for (short i=1; i<ST[Fid].nbp; i++;)
        switch(ST[Fid].sig[i].mod)
        {
        case 'in': _pass_in_parameter (LT[i]);
        case 'out': _pass_out_parameter (LT[i]);
        case 'inout': _pass_inout_parameter (LT[i]);
        };
if (ST[Fid].mod == 'asynchronous') then
{
        _invoke_operation(PT[Pid].name, ST[Fid].name);
        {IP, FR, SP, LT} = CS.pop();
}
else
{
        result = _invoke_operation(PT[Pid].name, ST[Fid].name);
        if (result == 'ok')   then
        {
                _retrieve_out_parameter();
                {IP, FR, SP, LT} = CS.pop();
                if (ST[Fid].TID != 'void') then
                        PS.push(_retrieve_return_value());
        }
        else // the result is an exception formatted as a message
        {
                FR = HT[result.MID];
                LT = result.LT;
                IP = RT[FT].IP;
        }
}
_close_package(PT[Pid].name);
// close service invocation context
```

Errors:

```
InvalidIdentifier
StackUnderflow
TypeMismatch
BadPackageStatus
InvalidObjectReference
```

13.3.31 Push

Short description: Push data value.

Synopsis: `PUSH Did`

Operands: `Did` shall be the two-byte representation of a data identifier holding the value to push onto the stack.

Stack: `... ⇨ ..., Val`

Types: Not applicable.

Parameters: `Val` shall be of the same type as the constant or variable identified by `Did`.

Effect: Check that `Did` identifies a constant or variable of a primitive type. Push the value of the constant or variable whose data identifier is `Did` onto the parameter stack.

Formal specification:
```
if (DT(Did).tid > 'object reference') then raise('InvalidType');
PS.push(DT(Did).val);
```

Errors:
```
InvalidIdentifier
InvalidType
```

13.3.32 Push reference

Short description: Push data identifier.

Synopsis: `PUSHR Did`

Operands: `Did` shall be the two-byte representation of a data identifier to push onto the stack.

Stack: `... ⇨ ..., Val`

Types: Not applicable.

Parameters: `Val` shall be of `data identifier` type.

Effect: Push `Did` onto the parameter stack.

Formal specification:
```
PS.push(Did);
```

Errors: None.

13.3.33 Push immediate

Short description: Push short integer.

Synopsis: `PUSHI Int`

Operands: `Int` shall be the two-byte representation of a signed short integer value (in complement-to-two notation) specifying the value to push onto the stack.

Stack: `... ⇨ ..., Val`

Types: Not applicable.

Parameters: `Val` shall be of `short` type.

Effect: Push `Int` onto the parameter stack.

Formal specification:
```
PS.push(Int);
```

Errors: None.

13.3.34 Pop

Short description: Pop value and assign it to a variable.

Synopsis: `POP Did`

Operands: `Did` shall be the two-byte representation of the data identifier of the variable to which to assign the top element of the stack.

Stack: `..., Val ⇨ ...`

Types: Not applicable.

Parameters: `Val` shall be interpreted as of the type of the variable identified by `Did`.

Effect: Check that `Did` identifies a variable of a primitive type. Pop `Val` from the parameter stack into the variable identified by `Did`.

Formal specification:

```
TID tid = VT(Did).TID;
if (tid > 'object reference') then raise('InvalidType')
VT(Did).val = PS.pop(tid);
```

Errors:

```
InvalidIdentifier
StackUnderflow
InvalidType
```

13.3.35 Pop reference

Short description: Pop value and assign it to the variable referenced by a variable.

Synopsis: `POPR Did`

Operands: `Did` shall be the two-byte representation of the data identifier of a variable of `data identifier` type, whose value identifies the variable to which to assign the value of the top element of the stack.

Stack: `..., Val ⇨ ...`

Types: Not applicable.

Parameters: `Val` shall be interpreted as of the type of `VT(Did).val`.

Effect: Check that `Did` identifies a variable of `data identifier` type.
Pop `Val` from the parameter stack and assign it to the variable identified by `Did`.

Formal specification:

```
TID tid = type(VT(Did).val.TID);
if (tid != 'data identifier') then raise('InvalidType');
VT(VT(Did).val).val = PS.pop(tid);
```

Errors:

```
InvalidIdentifier
StackUnderflow
InvalidType
```

13.3.36 Pop contents

Short description: Pop variable and assign its value to a variable.

Synopsis: `POPC Did1`

Operands: `Did1` shall be the two-byte representation of the data identifier of the variable to which to assign the value of the data identified by the top element of the stack.

Stack: `..., Did2 ⇨ ...`

Types: Not applicable.

Parameters: `Did2` shall be interpreted as of `data identifier` type. The data identified by `Did2` shall be interpreted as of the type of the data identified by `Did1`.

Effect: Check that `Did1` and `Did2` identify data of the same type.
Pop `Did2` from the parameter stack and assigns the value of `Did2` to the variable identified by `Did1`.

Formal specification:

```
DID did2 = PS.pop('data identifier');
if (VT(Did1).TID != DT(did2).TID) then raise('TypeMismatch');
VT(Did1).val = DT(did2).val;
```

Errors:

```
InvalidIdentifier
StackUnderflow
TypeMismatch
```

13.3.37 Allocate

Short description: Create dynamic variable.

Synopsis: `ALLOC Tid`

Operands: `Tid` shall be the two-byte representation of a type identifier specifying the type of the dynamic variable to allocate.

Stack: `..., ⇨ ..., Did`

Types: Not applicable.

Parameters: `Did` shall be of `data identifier` type.

Effect: Generate a dynamic variable whose type is identified by `Tid`. Push its data identifier onto the parameter stack.

Formal specification:

```
DID did = new(Tid);
VT[did].val = 'null';  // default value for the type
VT[did].TID = Tid;
PS.push(did);
```

Errors:

```
AllocationFailed
InvalidIdentifier
```

13.3.38 Increment

Short description: Increment variable.

Synopsis: `INC Did`

Operands: `Did` shall be the two-byte representation of the data identifier of the variable which to increment.

Stack: `..., Val ⇨ ...,`

Types: Not applicable.

Parameters: `Val` shall be interpreted as of the type of the variable identified by `Did`.

Effect: Check that `Did` identifies a variable of a numeric type. Pop the parameter stack and increment the value of the variable identified by `Did` by the popped value.

Formal specification:
```
TID tid = VT(Did).TID;
if (VT(Did).TID > 'double') raise('InvalidType');
VT(Did).val += PS.pop(<T>);
```

Errors:
```
InvalidIdentifier
StackUnderflow
InvalidType
ArithmeticOverflow
```

13.3.39 Decrement

Short description: Decrement variable.

Synopsis: `DEC Did`

Operands: `Did` shall be the two-byte representation of the data identifier of the variable which to decrement. The variable identified by `Did` shall be of a numeric type.

Stack: `..., Val ⇨ ...,`

Types: Not applicable.

Parameters: `Val` shall be interpreted as of the type of the variable identified by `Did`.

Effect: Check that `Did` identifies a variable of a numeric type. Pop the parameter stack and decrement the value of the variable identified by `Did` by the popped value.

Formal specification:
```
TID tid = VT(Did).TID;
if (VT(Did).TID > 'double') raise('InvalidType');
VT(Did).val -= PS.pop(<T>);
```

Errors:
```
InvalidIdentifier
StackUnderflow
InvalidType
ArithmeticOverflow
```

13.3.40 Get

Short description: Get value of element of data of constructed type.

Synopsis: `GET Did Lvl`

Operands: `Did` shall be the two-byte representation of the data identifier of the data to access.
`Lvl` shall be a one-byte unsigned quantity representing the number of nested levels to go to access the sought value.

Stack: `..., Idx(Lvl), ..., Idx(1) ⇨ ..., Val`

Types: Not applicable.

Parameters: `Idx(1), ... Idx(Lvl)` shall be interpreted as of `unsigned short` type.
Val shall be of the same type as the accessed element.

Effect: Replace a list of indices on the parameter stack by the value of the element addressed by the popped indices within the constructed type data identified by `Did`:

```
Val = DT(Did)[Idx(1),...,Idx(Lvl)]
```

Check that the accessed element is of a primitive type. If `Lvl` equals 0, perform as a `PUSH` instruction.

Formal specification:

```
void *buf = &DT(Did);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise ('InvalidIndex');
        buf = &buf->.val[idx];
}
if (buf->TID > 'object reference') then raise('InvalidType');
PS.push(buf->val);
```

Errors:

```
InvalidIdentifier
InvalidLevel
StackUnderflow
InvalidIndex
InvalidType
```

13.3.41 Get contents

Short description: Set data contents to element of data of constructed type.

Synopsis:

```
GETC Did1 Lvl
```

Operands: `Did1` shall be the two-byte representation of the data identifier of the data to access.
`Lvl` shall be a one-byte unsigned quantity representing the number of nested levels to go to access the element to access.

Stack:

```
..., Did2, Idx(Lvl), ..., Idx(1) ⇨ ...,
```

Types: Not applicable.

Parameters: `Idx(1), ... Idx(Lvl)` shall be interpreted as of `unsigned short` type.

Effect: Pop a list of indices and a data identifier `Did2` from the parameter stack. Within the constructed type data identified by `Did1`, assign the variable identified by `Did2` to the element addressed by the popped list of indices:

```
VT(Did2).Val = DT(Did1)[Idx(1),...,Idx(Lvl)]
```

Check that the element to access is of the same type as the data identified by `Did2`.

Formal specification:

```
void *buf = &DT(Did1);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise ('InvalidIndex');
        buf = &buf->.val[idx];
}
DID did2 = PS.pop('data identifier');
if (VT(did2).TID != buf->TID) then raise('TypeMismatch');
VT(did2).val = buf->val;
```

Errors:

```
InvalidIdentifier
InvalidLevel
StackUnderflow
InvalidIndex
TypeMismatch
```

13.3.42 Set

Short description: Set element of variable of constructed type to value.

Synopsis: `SET Did Lvl`

Operands: `Did` shall be the two-byte representation of the data identifier of the variable to modify.
`Lvl` shall be a one-byte unsigned quantity representing the number of nested levels to go to access the element to modify.

Stack: `..., Val, Idx(Lvl), ..., Idx(1)` ⇨ `...,`

Types: Not applicable.

Parameters: `Idx(1), ... Idx(Lvl)` shall be interpreted as of `unsigned short` type. `Val` shall be interpreted as of the same type as the element to modify.

Effect: Pop a list of indices and a value from the parameter stack. Within the structured variable identified by `Did`, assign the element addressed by the popped list of indices to the popped value:

```
VT(Did)[Idx(1),...,Idx(Lvl)] = Val
```

Check that the element to modify is of a primitive type. If `Lvl` equals 0, perform as a `POP` instruction.

Formal specification:

```
void *buf = &VT(Did);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise ('InvalidIndex');
        buf = &buf->.val[idx];
}
if (buf->TID > 'object reference') then raise('InvalidType');
buf->val = PS.pop(buf->TID);
```

Errors:

```
Invalid Identifier
InvalidLevel
StackUnderflow
InvalidIndex
InvalidType
```

13.3.43 Set contents

Short description: Set element of variable of constructed type to data contents.

Synopsis: `SETC Did1 Lvl`

Operands: `Did1` shall be the two-byte representation of the data identifier of the variable to modify.
`Lvl` shall be a one-byte unsigned quantity representing the number of nested levels to go to access the element to modify.

Stack: `..., Did2, Idx(Lvl), ..., Idx(1)` ⇨ `...,`

Types: Not applicable.

Parameters: `Idx(1), ... Idx(Lvl)` shall be interpreted as of unsigned short type.
`Did2` shall be interpreted as of `data identifier` type.

Effect: Pop a list of indices and a data identifier from the parameter stack. Within the structured variable identified by `Did1`, assign the element addressed by the popped list of indices to the value identified by the popped data:

`VT(Did1)[Idx(1),...,Idx(Lvl)] = DT(Did2).`

Check that `Did2` identifies a data of the type of the element to modify. If `Lvl` equals 0, perform as a `POPC` instruction.

Formal specification:

```
void *buf = VT(Did1);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <> 'object reference') then raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise ('InvalidIndex');
        buf = &buf->.val[idx];
}
DID did2 = PS.pop('data identifier');
if (DT(did2).TID != buf->TID) then raise('TypeMismatch');
buf->val = DT(did2).val;
```

Errors:
`InvalidIdentifier`
`InvalidLevel`
`StackUnderflow`
`InvalidIndex`
`TypeMismatch`

13.4 Type conversion rules

This subclause defines the rules that shall apply when a **convert** (`CVT`) instruction is used to convert a parameter stack value (hence a value of a primitive type) from a source type to a destination type.

Values of the `data identifier` and `object reference` types shall not be converted to or from a value of another type.

As regards the other primitive types, not all type conversions are allowed; however, any of them can be converted to any other using sequences of conversions.

Table 4 shows the allowed type conversions together with the number of the subclause in which they are defined:

Table 4: Type conversions

Source/Destination	O	S	L	W	U	F	D	B	C
O	N/A	14.4.2.2	N/A	14.4.2.2	N/A	N/A	N/A	14.4.4	N/A
S	N/A	N/A	14.4.2.3	14.4.1	14.4.3	N/A	N/A	14.4.4	N/A
L	N/A	14.4.5	N/A	N/A	14.4.1	14.4.2.3	N/A	14.4.4	N/A
W	14.4.5	14.4.1	14.4.2.3	N/A	14.4.2.3	N/A	N/A	14.4.4	14.4.1
U	N/A	N/A	14.4.1	14.4.5	N/A	14.4.2.3	N/A	14.4.4	N/A
F	N/A	N/A	14.4.6	N/A	14.4.6	N/A	14.4.2.3	N/A	N/A
D	N/A	N/A	N/A	N/A	N/A	14.4.5	N/A	N/A	N/A
B	14.4.2.1	14.4.2.1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
C	N/A	N/A	N/A	14.4.1	N/A	N/A	N/A	N/A	N/A

13.4.1 Reversible conversions

The following conversions are lossless (i.e. preserve information) when reversed:

- between `unsigned short` and `character` (WC, CW);
- between `short` and `unsigned short` (SW, WS);
- between `long` and `unsigned long` (LU, UL).

For all these conversions, the result of the conversion shall be the value of the target type that has the same complement-to-two notation as the source value.

13.4.2 Lossless extensions

The following conversions extend the source value in a lossless fashion:

- from `boolean` (BO, BS) (see subclause 13.4.2.1);
- from `octet` to a numeric type (OS, OW) (see subclause 13.4.2.2);
- from a signed numeric type to a signed numeric type with a larger range (SL, LF, FD) (see subclause 13.4.2.3);
- from an unsigned numeric type to any numeric type with a larger range (WL, WU, UF) (see subclause 13.4.2.3).

13.4.2.1 Conversions from `boolean`

If the value of the source `boolean` is false, the value in the destination type shall be 0.

If the value of the source `boolean` is true, the value in the destination type shall be the value which corresponds to all bits set at 1 (in complement-to-two notation), i.e.

- 255 for an `octet` destination type;
- -1 for a `short` destination type.

13.4.2.2 Conversions from `octet` to a numeric type

The value in the destination type shall be the octet value.

13.4.2.3 Lossless conversions from a numeric to a larger numeric type

The value in the destination type shall be the same numeric value as the value in the source type.

13.4.3 Lossy extensions

The conversion from `short` to `unsigned long` (SU) shall perform as follows:

- if the source value is positive or null, the destination value shall be the same numeric value as the source value;
- if the source value is strictly negative, the destination value is unspecified.

13.4.4 Truncations to `boolean`

Truncations from an `octet` or numeric type to `boolean` (OB, SB, WB, LB, UB) shall perform as follows:

- if the source value is 0, the destination value shall be 'false';
- if the source value is different from 0, the destination value shall be 'true'.

13.4.5 Truncations between integer or between floating-point types

Truncations from an integer type to an `octet` or integer type (WO, LS, UW) or from a floating-point type to another floating-point type (DF) shall perform as follows:

- if the source value is within the range of the destination type, then the destination value shall be the same numeric value as the source value;
- otherwise, the destination value is unspecified.

13.4.6 Truncations from floating-point to integer

Truncations from a floating-point type to an integer type (FL, FU) shall perform as follows:

- first the decimal part of the source value shall be truncated to an integer value (rounding down);
- then the rules defined in subclause 13.4.5 shall apply to the truncated value.

14 IDL mapping to MHEG-SIR

This Clause specifies how an IDL specification shall be mapped to the declarations of an interchanged script, when this IDL specification is intended for use by the script as an external service provider.

This Clause defines the mapping to MHEG-SIR declarations for

- IDL interfaces and modules;
- IDL types;
- IDL constants;
- references to IDL objects;
- IDL operations;
- IDL attributes;
- IDL exceptions.

14.1 IDL specifications

An IDL specification shall be mapped to an MHEG-SIR `PackageDeclaration` declared as a component of an `external-package-declarations` component of the `InterchangedScript`. The name of the IDL specification shall be mapped to the `name` component of this package declaration.

NOTE: Examples of IDL specifications are MHEG API, MPEG/DSM-CC.

If the number of operations or exceptions of an IDL specification exceed the size of a package, the specification shall be splitted into several packages sharing the same name, but having different MHEG-SIR identifiers.

14.2 IDL interfaces and modules

As the package declaration is a "flat" organisation, there is neither a mapping for an IDL module nor for an IDL interface. However, a reference to the embedding interface (i.e. a parameter of type `Object`) shall be provided as an implicit parameter to each invocation of function describing an IDL operation.

14.3 IDL operations

An IDL operation shall be mapped to an MHEG-SIR `services` component of the package declaration that maps the IDL specification to which the operation belongs.

14.3.1 Operation name

The global name for an IDL operation shall be mapped to the MHEG-SIR `name` component of this service description.

14.3.2 Operation parameters

The parameters of an IDL operation shall be mapped to the `parameters-description` component of the service description. In a `ServiceParameterDescription`, each IDL parameter type shall be mapped to the `type` component which identifies a type declared according to the type mapping rules defined in this Clause. The IDL passing mode for a parameter shall be mapped to the `passing-mode` component of the corresponding MHEG-SIR service parameter description.

If the operation has neither an output parameter nor a return value and is specifically designed to return several exceptions in sequence (e.g. for notification purposes), the value of its `calling-mode` component should be 'asynchronous'. Otherwise, the value of the `calling-mode` component shall be 'synchronous'.

If a semantically synchronous operation is intended to raise several exceptions in sequence, it should be splitted into two MHEG-SIR operations: a **synchronous** one and an **asynchronous** one.

14.3.3 Implicit parameter

When an IDL operation is mapped to an MHEG-SIR service description, the object instance to which the operation applies shall remain an implicit parameter, i.e. shall not be expressed as part of the signature of the service.

NOTE: However, upon invoking the operation, this parameter is provided as the leading actual parameter as if its `type` were 'object reference' and its `passing-mode` were 'in'.

14.3.4 Return value

The return value type of an IDL operation shall be mapped to the `return-value-type` component of the service description.

14.4 IDL attributes

An IDL attribute shall be mapped to two service descriptions within a package declaration: one accessor service, whose function is to get the value of the attribute, and one modifier service, whose function is to set the value of the attribute.

14.4.1 Accessor

As concerns the accessor service, the global IDL attribute name whose final identifier is prefixed with "get_" shall be mapped to the MHEG-SIR `name` component of the service description. An accessor service shall have no explicit parameter. The IDL attribute type shall be mapped to the `return-value-type` component of the service description.

EXAMPLE: In the MHEG-3 API, the `routine_id` attribute of the `RoutineInvocation` object shall be mapped to the IDL global name `MHEG_3::RoutineInvocation::get_RoutineId`

14.4.2 Modifier

As concerns the modifier service, the global IDL attribute name whose final identifier is prefixed with "set_" shall be mapped to the MHEG-SIR `name` component of the service description. A modifier service shall have one parameter with `in` passing mode and such that the IDL attribute type shall be mapped to the `type` component of the parameters description for this service. A modifier service shall have no return value.

14.4.3 Readonly attribute

If an IDL attribute is defined as `readonly`, only the accessor service shall be provided as part of the package declaration.

14.5 IDL inherited operations

Inherited IDL operations shall be mapped as if they were defined in the specific interface.

14.6 IDL exceptions

An IDL exception shall be mapped to an MHEG-SIR `exception-description` component of the package declaration that maps the IDL specification to which the exception belongs.

14.6.1 Exception name

The IDL global name of the exception shall be mapped to the MHEG-SIR `name` component of this exception description.

14.6.2 Exception members

Members of an IDL exception shall be mapped to the `parameters-description` component of this exception description. In this parameters description, each IDL member type shall be mapped to the `type` component which identifies a type declared according to the type mapping rules defined in this Clause.

14.6.3 Implicit member

When an IDL exception is mapped to an MHEG-SIR exception description, the object instance from which the exception originates shall remain an implicit member, i.e. shall not be expressed as part of the signature of the exception.

NOTE: However, upon raising the exception, this member is provided as the leading actual member as if its `type` were 'object reference'.

14.7 IDL types

An IDL type shall be mapped to an MHEG-SIR `TypeDeclaration` declared as a component of the `type-declarations` component of the `InterchangedScript`. A type declaration shall have a global scope in the interchanged script.

IDL basic types and constructors shall be mapped to MHEG-SIR primitive types and constructors as summarised in Table 4:

Table 4: Type mapping

IDL	MHEG-SIR
void	void
octet	octet
short	short
unsigned short	unsigned short
long	long
unsigned long	unsigned long
float	float
double	double
boolean	boolean
char	character
enum	unsigned long
string	string
sequence	sequence
array	array
struct	structure
union	union
(object)	object reference
any	data identifier (see below)

14.7.1 `char` type

Mapping IDL `char` types to MHEG-SIR `character` types shall involve transcoding values from ISO 8859-1 to ISO 10646-1.

14.7.2 `enum` type

The range checking of `enum` values need not be preserved.

14.7.3 Constructed types

An IDL type definition shall be mapped to an MHEG-SIR `TypeDescription`. If the IDL type is a basic type or if it has already been the subject of another type declaration, this type description shall consist of a type identifier. Otherwise, it shall be constructed according to the following mapping rules:

- an IDL `struct` field shall be mapped to its rank in the MHEG-SIR `structure` description; its name shall not be preserved;
- an IDL `union` tag value shall be mapped to its rank in the MHEG-SIR `union` description; its name and value shall not be otherwise preserved;
- a multidimensional IDL `array` shall be mapped to an MHEG-SIR `array` whose element type is `array`.

14.7.4 `any` type

The IDL `any` type shall be mapped to MHEG-SIR `data identifier` provided that the `any` type is used with an associated key to determine the actual type:

```
struct { Key the_key; any value }
```

where `Key` is a `string`, numeric or `enum` type whose value completely determines the type of the `value` field.

The above IDL type shall be mapped to an MHEG-SIR `structure` of two elements:

- an `unsigned short` representing a valid TID within the script, to map the key;
- a `data identifier` representing a variable of the type identified by the first element and which holds the value.

Any other use of the `any` type is not guaranteed to have its semantics preserved when mapped to MHEG-SIR.

14.7.5 Restrictions on types

If two IDL constructed types have the same structure, they shall be mapped to a single MHEG-SIR type.

14.8 IDL constants

IDL constants shall be mapped to an MHEG-SIR `ConstantDeclaration` declared as a component of the `constant-declarations` component of the `InterchangedScript`. A constant declaration shall have a global scope in the interchanged script.

15 The MHEG-3 API

This Clause specifies the syntax and semantics of the MHEG-3 API.

Interchanged scripts shall use the MHEG-3 API according to the IDL interface syntax defined in this Clause and in Annex F.

MHEG-SIR script interpreters shall provide the MHEG-3 API according to the IDL interface syntax defined in this Clause and in Annex F, with the semantics defined in this Clause. The invocation of the operations shall have the effect specified in this Clause.

All MHEG-SIR predefined functions that map MHEG-3 API operations shall be **synchronous**.

The MHEG-3 API definition consists of a unique IDL module called `MHEG_3`. This module defines predefined types, three exceptions and four object interfaces; there is no inheritance relationship among the four objects.

15.1 `ScriptInterpreter` object

The `ScriptInterpreter` object represents the script interpreter. It shall be unique. It is used as a factory for `MhScript` objects.

To invoke operations on the `ScriptInterpreter` object, interchanged scripts shall use 'null' as the value of the implicit object reference parameter.

15.1.1 `kill` operation

Synopsis:

Interface:	`ScriptInterpreter`
Operation:	`kill`
Result:	`void`

Description:

The `kill` operation is used to kill the `ScriptInterpreter` object and terminate the script interpreter process.

When the operation is invoked, the main process shall invoke a `destroy` operation on all available `MhScript` objects then terminate the script interpreter process.

Unlike the other MHEG-3 API operations, this operation is not an MHEG-SIR predefined function. Therefore, it shall not be available for use by MHEG-SIR interchanged scripts.

15.1.2 `prepare` operation

Synopsis:

Interface:	`ScriptInterpreter`	
Operation:	`prepare`	
Result:	`MhScript`	
In:	`ContentReference`	`content_reference`
Exception:	`InvalidParameter`	
Exception:	`InvalidScript`	
Exception:	`OperationFailed`	

Description:

The `prepare` operation is used to create an `MhScript` object from an interchanged script and request the script interpreter to initialise that mh-script.

The `content_reference` parameter specifies the location of the interchanged script. It consists of two strings: a public identifier and a system identifier. If any one of these strings is null, it shall be ignored. At least one of both string field values shall be non-null.

When the operation is invoked, the main process shall perform the mh-script initialisation operations as specified by subclause 9.5.2. As soon as this has been achieved, the status of the mh-script shall become **available**.

The result of the operation shall be an object reference to the created `MhScript`.

The `InvalidParameter` exception shall be raised if the `content_reference` parameter does not allow to access an interchanged script. Then the `rank` member shall be 1.

The `InvalidScript` exception shall be raised if an illegal statement is detected during parsing of the interchanged script. Then the `the_entity` member shall represent the type of the first entity in the declarations part on which an error has been detected, whereas the `identifier` member shall represent the identifier of this entity as follows:

- a TID for a type declaration;
- a DID for a constant declaration or a variable declaration;

- a FID for a service declaration or a routine declaration;
- a MID for an exception declaration or a handler declaration;
- a PID for a package declaration.

The `OperationFailed` exception shall be raised if the mh-script initialisation operations cannot be completed although no error has been detected in the syntax of the interchanged script.

Whenever an exception is raised, the `MhScript` object shall not be created and the status of the mh-script shall remain **not available**.

15.2 `MhScript` object

The `MhScript` object represents an available mh-script. It is used as a factory for `RtScript` objects.

15.2.1 `destroy` operation

Synopsis:

Interface: `MhScript`
Operation: `destroy`
Result: `void`

Description:

The `destroy` operation is used to kill the `MhScript` object and destroy the corresponding mh-script.

When the operation is invoked, the main process shall perform the following steps in the specified order:

- put the target mh-script to **not available** status;
- invoke a `delete` operation on all existing `RtScript` objects that have been created by this mh-script;
- perform the **package unload** procedure for all packages;
- release all the mh-script memory areas attached to the mh-script.

15.2.2 `new` operation

Synopsis:

Interface: `MhScript`
Operation: `new`
Result: `RtScript`
Exception: `OperationFailed`

Description:

The `new` operation is used to create an `RtScript` object from the mh-script and request the script interpreter to initialise that rt-script.

When the operation is invoked, the main process shall perform the rt-script initialisation operations as specified by subclause 9.5.3. After successful initialisation, the status of the rt-script shall become **ready**.

The result of the operation shall be an object reference to the created `RtScript`.

The `OperationFailed` exception shall be raised if the rt-script initialisation operations cannot be completed. Then the `RtScript` object shall not be created and the status of the rt-script shall remain **not ready**.

15.3 `RtScript` object

The `RtScript` object represents an rt-script whose status is **ready**, **running** or **erroneous**. It is used as a factory for `RoutineInvocation` objects.

15.3.1 `delete` operation

Synopsis:

Interface: `RtScript`
Operation: `delete`
Result: `void`

Description:

The `delete` operation is used to kill the `RtScript` object and destroy the corresponding rt-script.

When the operation is invoked, the main process shall perform the following steps in the specified order:

- put the target rt-script to **not ready** status;
- invoke a `close` operation on all `RoutineInvocation` objects that have been created by the rt-script;
- terminate all processing units;
- release all rt-script memory areas attached to the rt-script.

15.3.2 `setPriority` operation

Synopsis:

Interface: `RtScript`
Operation: `setPriority`
Result: `void`
In: `unsigned short priority`

Description:

The `setPriority` operation is used to modify the scheduling priority associated with the rt-script.

The `priority` parameter specifies the new priority value.

When the operation is invoked, the main process may modify its scheduling policy accordingly. The precise effect of this operation is not specified by this part of ISO/IEC 13522. Depending on the implementation, it may have no effect. However, the execution unit of an rt-script with a lower priority value than another rt-script shall not be given more CPU time than the execution unit of the latter.

15.3.3 `getPriority` operation

Synopsis:

Interface: `RtScript`
Operation: `getPriority`
Result: `unsigned short`

Description:

The `getPriority` operation is used to retrieve the current value of the scheduling priority associated with the rt-script. If no priority has been explicitly set to this rt-script, the default value specified by the script interpreter shall be used.

15.3.4 `setData` operation

Synopsis:

Interface:	`RtScript`	
Operation:	`setData`	
Result:	`void`	
In:	`DID`	`variable_id`
In:	`any`	`variable_value`
Exception:	`InvalidParameter`	
Exception:	`OperationFailed`	

Description:

The `setData` operation is used to assign a value to a global or dynamic variable of the rt-script.

The `variable_id` parameter specifies the data identifier of the data to modify.

The `variable_value` parameter specifies the value to assign to the variable. The type of the actual parameter is determined by the type of the variable.

When the operation is invoked, the main process shall request the rt-script execution unit to assign the target variable to the provided value.

The `InvalidParameter` exception shall be raised

- if the `variable_id` parameter references a constant, a local variable or a non-existing constant or variable. Then the `rank` member shall be 1;
- if the `variable_value` parameter is not of an IDL type that matches the MHEG-SIR type of the target variable. Then the `rank` member shall be 2.

The `OperationFailed` exception shall be raised if the status of the rt-script is **running** or **erroneous**.

15.3.5 `getData` operation

Synopsis:

Interface:	`RtScript`	
Operation:	`getData`	
Result:	`any`	
In:	`DID`	`data_id`
Exception:	`InvalidParameter`	
Exception:	`OperationFailed`	

Description:

The `getData` operation is used to retrieve the current value of a constant or variable.

The `data_id` parameter specifies the data identifier of the data to access.

When the operation is invoked, the rt-script execution unit shall return the current value of the constant or variable.

The result of the operation shall be the requested value and shall be of an IDL type that matches the MHEG-SIR type of the target constant or variable.

The `InvalidParameter` exception shall be raised if the `data_id` parameter references a non-existing constant or variable. Then the `rank` member shall be 1.

The `OperationFailed` exception shall be raised if the status of the rt-script is **running** or **erroneous**.

15.3.6 `allocate` operation

Synopsis:

Interface:	`RtScript`	
Operation:	`allocate`	
Result:	`DID`	
In:	`TID`	`variable_type_id`
Exception:	`InvalidParameter`	
Exception:	`OperationFailed`	

Description:

The `allocate` operation is used to create a dynamic variable of a given type within the rt-script.

The `variable_type_id` parameter specifies the MHEG-SIR type identifier of the target variable, as declared within the rt-script.

When the operation is invoked, the rt-script execution unit shall perform as if it would execute an `ALLOC` instruction with `variable_type_id` as operand, i.e. it shall reserve appropriate heap memory, generate a new DID and return it.

The result of the operation shall be the data identifier of the new dynamic variable.

The `InvalidParameter` exception shall be raised if the value of the `variable_type_id` parameter is neither a predefined type nor a type declared within the rt-script. Then the `rank` member shall be 1.

The `OperationFailed` exception shall be raised wherever the `ALLOC` instruction would raise an error. Then the variable shall not be allocated and the error register shall not be modified.

15.3.7 `free` operation

Synopsis:

Interface:	`RtScript`	
Operation:	`free`	
Result:	`void`	
In:	`DID`	`variable_id`
Exception:	`InvalidParameter`	

Description:

The `free` operation is used to destroy a dynamic variable of the rt-script.

The `variable_id` parameter specifies the data identifier of the variable to be released.

When the operation is invoked, the rt-script execution unit shall perform as if it would execute a `FREE` instruction with `variable_id` as parameter, i.e. it shall release the dynamic variable and make its identifier invalid.

The `InvalidParameter` exception shall be raised if the `variable_id` parameter does not refer to an existing dynamic variable previously allocated through the MHEG-3 API. Then the `rank` member shall be 1.

NOTE: A script interpreter may use a data identifier allocation policy that allows to distinguish easily variables allocated through the MHEG-3 API from variables allocated using an instruction, for instance by the range to which their data identifier belongs.

15.3.8 `stop` operation

Synopsis:

Interface: `RtScript`
Operation: `stop`
Result: `void`
Exception: `OperationFailed`

Description:

The `stop` operation is used to put the rt-script back into ready status.

When the operation is invoked, the script interpreter shall request the rt-script execution unit to stop, flush the calling stack, message queue and parameter stack and reset all registers. It shall then put the rt-script to **ready** status. Unlike the `reInit` operation, the global and dynamic variables shall not be changed.

The `OperationFailed` exception shall be raised if the operation could not be performed successfully, for instance if the rt-script memory areas have been corrupted due to an execution error.

15.3.9 `reInit` operation

Synopsis:

Interface: `RtScript`
Operation: `reInit`
Result: `void`
Exception: `OperationFailed`

Description:

The `reInit` operation is used to put the rt-script back into its initial state, i.e. just after initialisation.

When the operation is invoked, the script interpreter shall

- terminate the rt-script execution unit;
- release all dynamic variables;
- set the global variables back to their initial values (as in the mh-script global variable definition table);
- flush the parameter stack, the message queue and the calling stack, releasing local variable tables;
- reset all registers;
- finally, put the rt-script to **ready** status.

The `OperationFailed` exception shall be raised if the operation could not be performed successfully, for instance if the rt-script memory areas have been corrupted due to an execution error.

15.3.10 `getRtScriptStatus` operation

Synopsis:

Interface:	`RtScript`
Operation:	`getRtScriptStatus`
Result:	`RtScriptStatus`

Description:

The `getRtScriptStatus` operation is used to retrieve the current status of the rt-script.

The result of the operation shall be one of the following: `READY`, `RUNNING` or `ERRONEOUS`.

15.3.11 `open` operation

Synopsis:

Interface:	`RtScript`	
Operation:	`open`	
Result:	`RoutineInvocation`	
In:	`FID`	`routine_id`
Exception:	`InvalidParameter`	

Description:

The `open` operation is used to create an `RoutineInvocation` object from the rt-script.

The `routine_id` parameter specifies the function identifier of the routine with which the new `RoutineInvocation` object is associated.

The script interpreter may opt for either of the following policies:

a) to retrieve the signature of the target routine when the `open` operation is invoked, so as to check the validity of the passed parameters "on the fly", i.e. as soon as a `setParameter` operation is invoked;

b) to check the validity of parameters only upon invocation of the `run` operation.

The result of the operation shall be an object reference to the created `RoutineInvocation`.

The `InvalidParameter` exception shall be raised if the `routine_id` parameter does not identify a valid routine of the rt-script. In this case, the `rank` member shall be 1.

15.4 `RoutineInvocation` object

The `RoutineInvocation` object represents an invocation context of a routine. This invocation context is used to pass parameters to and to request execution of a given routine of the rt-script.

15.4.1 `close` operation

Synopsis:

Interface:	`RoutineInvocation`
Operation:	`close`
Result:	`void`

Description:

The `close` operation is used to kill the `RoutineInvocation` object and close the corresponding routine invocation context.

15.4.2 `routine_id` readonly attribute

Synopsis:

Interface:	`RoutineInvocation`	
Attribute:	`FID`	`routine_id`

Description:

The `routine_id` attribute is a readonly attribute that is set at creation of the `RoutineInvocation` object by the `open` operation. Its value shall be the function identifier of the routine that the `RoutineInvocation` object addresses.

Interchanged scripts shall access the value of this attribute using the `get_RoutineId` predefined function.

15.4.3 `setParameter` operation

Synopsis:

Interface:	`RoutineInvocation`	
Operation:	`setParameter`	
Result:	`void`	
In:	`unsigned short`	`rank`
In:	`TID`	`parameter_type_id`
In:	`any`	`parameter_value`
Exception:	`InvalidParameter`	

Description:

The `setParameter` operation is used to pass the value of a parameter of the routine for use by the next `run` operation.

The `rank` parameter specifies the rank of the passed parameter in the routine signature description, where 0 indicates the first parameter. It therefore corresponds to the index of the parameter in the routine's local variable table.

The `parameter_type_id` parameter specifies the MHEG-SIR type identifier of the passed parameter, as declared within the rt-script.

The `parameter_value` parameter specifies the value of the passed parameter. The type of the value is determined by the `parameter_type_id` parameter.

When the operation is invoked, the script interpreter shall buffer the parameter for use by the next `run` operation on this routine. If the script interpreter opts for policy a) defined in subclause 15.3.11, it shall check the validity of the `parameter_type_id` and `parameter_value` parameters with regard to the routine's signature.

If the script interpreter opts for policy a) defined in subclause 15.3.11, the `InvalidParameter` exception shall be raised

- if the operation's `rank` parameter exceeds the number of the last parameter of the routine. Then the exception's `rank` member shall be 1;
- if the `parameter_type_id` parameter does not correspond to the type of parameter in the routine's signature. Then the exception's `rank` member shall be 2;
- if the `parameter_value` parameter is not of an appropriate type, i.e. an IDL type that matches the type described by the `parameter_type_id` parameter, when the passing mode is by **value**, and a DID type when the passing mode is by **reference**. Then the exception's `rank` member shall be 3;
- when the passing mode is by reference, if the `parameter_value` parameter is a DID that does not identify an existing global or dynamic variable whose type matches the parameter type defined by the routine's signature. Then the exception's `rank` member shall be 3.

15.4.4 `getPrototype` operation

Synopsis:

Interface: `RoutineInvocation`
Operation: `getPrototype`
Result: `Prototype`

Description:

The `getPrototype` operation is used to retrieve the signature of the routine.

When the operation is invoked, the script interpreter shall return the signature of the routine.

The result of the operation shall be a description of the routine signature:

a) the `return_value_type` field shall be set to `RT[routine_id].TID`;
b) the nth item of the signature field shall correspond to `RT[routine_id].sig[n]`:
   1) the `passing_mode` field shall be set to `BY_VALUE`, or `BY_REFERENCE` respectively, when `RT[routine_id].sig[n].mod` is 'value', 'reference' respectively;
   2) the `parameter_type_id` field shall be set to `RT[routine_id].sig[n].TID`.

15.4.5 `run` operation

Synopsis:

Interface: `RoutineInvocation`
Operation: `run`
Result: `void`
Exception: `OperationFailed`

Description:

The `run` operation is used to request the execution of the routine with the parameter values previously provided using the `setParameter` operation.

When the operation is invoked, the main process shall

- create a message whose message identifier is the index of the routine (i.e. the value of the `routine_id` attribute) and whose parameters are the parameters set by the preceding `setParameter` operations;
- insert this message into the message queue of the target rt-script;
- if the current status of the rt-script is **ready**, put it to **running**.

If the script interpreter opts for policy b) defined in subclause 15.3.11, it shall check the validity, with regard to the routine's signature, of all the type identifiers and values of the parameters previously provided using the `setParameter` operation.

The `OperationFailed` exception shall be raised if any of the provided parameters does not map the routine's signature.

15.4.6 `reset` operation

Synopsis:

Interface: `RoutineInvocation`
Operation: `reset`
Result: `void`

Description:

The `reset` operation is used to clear the routine invocation context to prepare a new invocation.

When the operation is invoked, the parameters previously buffered as the result of a `setParameter` operation shall be cleared.

NOTE: Using this operation after each `run` avoids any risk of collision. Not using it allows to repeat the same invocation without supplying the parameters again.

15.4.7 `getInvocationStatus` operation

Synopsis:

Interface: `RoutineInvocation`
Operation: `getInvocationStatus`
Result: `InvocationStatus`

Description:

The `getInvocationStatus` operation is used to retrieve the current routine invocation status.

The result of the operation shall be one of the following values:

- `NOT_STARTED`: no `run` operation has been invoked since the creation of the object or since the last `reset` operation;
- `PROCESSING`: a `run` operation has been invoked but the routine execution has not been completed by the rt-script execution unit (either the request is in the message queue or the routine is currently under execution);
- `TERMINATED`: the routine execution triggered by the last invoked `run` operation has been completed by the rt-script execution unit;
- `ABORTED`: the routine execution triggered by the last invoked `run` operation has resulted in an instruction execution error.

Annex A
(normative)
ASN.1 specification of interchanged scripts

This Annex specifies the ASN.1 notation, according to ISO/IEC 8824-1 [1], for the syntax of the "script data" component of the MHEG "script" class.

Interchanged scripts shall have the syntax defined by the ASN.1 `ISOMHEG-sir` module.

```
-- Module: MHEG-SIR (sir)--
--
-- Copyright statement:
-- --------------------
--      (c) International Organization for Standardization, 1996.
--      Permission to copy in any form is granted for use with conforming
--      MHEG-3 engines and applications as defined by ISO/IEC 13522-3
--      provided this notice is included in all copies.

ISOMHEG-sir {joint-iso-itu-t (2) mheg (19) version (1) script-interchange-representation (11)}
DEFINITIONS IMPLICIT TAGS ::= BEGIN

EXPORTS         InterchangedScript;

InterchangedScript      ::=     SEQUENCE
{
        type-declarations               SEQUENCE (SIZE (1.. max-nb-declared-types)) OF
                                            TypeDeclaration OPTIONAL,
        constant-declarations           [0] SEQUENCE (SIZE (1 .. max-nb-constants)) OF
                                        ConstantDeclaration OPTIONAL,
        global-variable-declarations    [1] SEQUENCE (SIZE (1 .. max-nb-global-variables)) OF
                                        VariableDeclaration OPTIONAL,
        external-package-declarations   [2] SEQUENCE (SIZE (1 .. max-nb-packages)) OF
                                        PackageDeclaration OPTIONAL,
        handler-declarations            [3] SEQUENCE (SIZE (1 .. max-nb-messages)) OF
                                        HandlerDeclaration OPTIONAL,
        routine-declarations            [4] SEQUENCE (SIZE (1 .. max-nb-routines)) OF
                                        RoutineDeclaration OPTIONAL
}

TypeDeclaration         ::=     SEQUENCE
{
        identifier                      [0] TypeIdentifier OPTIONAL,
        description                     TypeDescription
}

TypeDescription         ::=     CHOICE
{
        string-description              [1] INTEGER (0..max-size-string) OPTIONAL,
        sequence-description            [2] SequenceDescription,
        array-description               [3] ArrayDescription,
        structure-description           [4] StructureDescription,
        union-description               [5] UnionDescription
}

SequenceDescription     ::=     SEQUENCE
{
        bound                           INTEGER (0 .. max-size-sequence),
        element-type                    TypeIdentifier
}

ArrayDescription        ::=     SEQUENCE
{
        size                            INTEGER (1 .. max-size-array),
        element-type                    TypeIdentifier
}

UnionDescription        ::=     SEQUENCE (SIZE (1 .. max-size-union)) OF TypeIdentifier

StructureDescription    ::=     SEQUENCE (SIZE (1 .. max-size-structure)) OF TypeIdentifier

ConstantDeclaration     ::=     SEQUENCE
{
```

```
        identifier                      [0] DataIdentifier OPTIONAL,
        type                            TypeIdentifier ALL EXCEPT 0,
        value                           ConstantValue
}
ConstantValue           ::=     CHOICE
{
        octet                           [1] OctetValue,
        short                           [2] ShortValue,
        long                            [3] LongValue,
        unsigned-short                  [4] UnsignedShortValue,
        unsigned-long                   [5] UnsignedLongValue,
        float                           [6] FloatValue,
        double                          [7] DoubleValue,
        boolean                         [8] BooleanValue,
        character                       [9] CharacterValue,
        data-identifier                 [10] DataIdentifier (0..<max-nb-constants),
        string                          [11] StringValue,
        sequence                        [12] SequenceValue,
        array                           [13] ArrayValue,
        structure                       [14] StructureValue,
        union                           [15] UnionValue
}

SequenceValue           ::=     SEQUENCE (SIZE (0 .. max-size-sequence)) OF ConstantValue

ArrayValue              ::=     SEQUENCE (SIZE (1 .. max-size-array)) OF ConstantValue

UnionValue              ::=     SEQUENCE
{
        tag                             INTEGER (0 .. < max-size-union),
        value                           ConstantValue
}

StructureValue          ::=     SEQUENCE (SIZE (1 .. max-size-structure)) OF ConstantValue

VariableDeclaration     ::=     SEQUENCE
{
        identifier                      [0] DataIdentifier OPTIONAL,
        type                            TypeIdentifier,
        initial-value                   ConstantReference OPTIONAL
}

PackageDeclaration      ::=     SEQUENCE
{
        identifier                      [0] PackageIdentifier OPTIONAL,
        name                            VisibleString OPTIONAL,
        services                        SEQUENCE (SIZE (0 .. max-nb-services)) OF
                                        ServiceDescription,
        exceptions                      SEQUENCE (SIZE (0 .. max-nb-exceptions)) OF
                                        ExceptionDescription
}

ServiceDescription      ::=     SEQUENCE
{
        identifier                      [0] FunctionIdentifier OPTIONAL,
        name                            VisibleString OPTIONAL,
        calling-mode                    ENUMERATED {synchronous (0), asynchronous (1)}
                                        DEFAULT synchronous,
        return-value-type               TypeIdentifier DEFAULT 0,
        parameters-description          SEQUENCE OF ServiceParameterDescription OPTIONAL
}

ServiceParameterDescription     ::=     SEQUENCE
{
        passing-mode                    ENUMERATED {in (1), out (2), inout (3)} DEFAULT in,
        type                            TypeIdentifier ALL EXCEPT 0
}

ExceptionDescription    ::=     SEQUENCE
{
        identifier                      [0] MessageIdentifier OPTIONAL,
        name                            VisibleString OPTIONAL,
        parameters-description          SEQUENCE OF TypeIdentifier OPTIONAL
}

HandlerDeclaration      ::=     SEQUENCE
{
        message-identifier              MessageIdentifier,
```

```
        function-identifier             FunctionIdentifier
}

RoutineDeclaration      ::=     SEQUENCE
{
        routine-description             RoutineDescription,
        program-code                    OCTET STRING
}

RoutineDescription      ::=     SEQUENCE
{
        identifier                      [0] FunctionIdentifier OPTIONAL,
        return-value-type               TypeIdentifier DEFAULT 0,
        parameters-description          [1] SEQUENCE OF RoutineParameterDescription OPTIONAL,
        local-variable-table            [2] SEQUENCE (SIZE (0 .. max-nb-local-variables)) OF
                                        VariableDeclaration OPTIONAL
}

RoutineParameterDescription ::= SEQUENCE
{
        passing-mode                    ENUMERATED {value (1), reference (3)} DEFAULT value,
        type                            TypeIdentifier ALL EXCEPT 0
}

ConstantReference       ::=     CHOICE
{
        identifier                      [16] DataIdentifier,
        value                           ConstantValue
}

max-size-sequence               INTEGER         ::=     65535
max-size-string                 INTEGER         ::=     65535
max-size-array                  INTEGER         ::=     65536
max-size-union                  INTEGER         ::=     256
max-size-structure              INTEGER         ::=     256
max-nb-global-variables         INTEGER         ::=     28672
max-nb-constants                INTEGER         ::=     4096
max-nb-local-variables          INTEGER         ::=     256
max-nb-dynamic-variables        INTEGER         ::=     32512
max-nb-data                     INTEGER         ::=     65536
                -- max-nb-constants+max-nb-global-variables+max-nb-local-variables+max-nb-dynamic-
        -- variables
max-nb-packages                 INTEGER         ::=     192
max-nb-services                 INTEGER         ::=     256
max-nb-routines                 INTEGER         ::=     4096
max-nb-predef-functions         INTEGER         ::=     12288
max-nb-functions                INTEGER         ::=     65536
                -- max-nb-packagesxmax-nb-services+max-nb-predef-functions+max-nb-routines
max-nb-exceptions               INTEGER         ::=     256
max-nb-predef-messages          INTEGER         ::=     16384
max-nb-messages                 INTEGER         ::=     65536
                -- max-nb-packagesxmax-nb-exceptions+max-nb-predef-messages
max-nb-declared-types           INTEGER         ::=     16384
max-nb-predef-types             INTEGER         ::=     16384
max-nb-types                    INTEGER         ::=     32768
                -- max-nb-predef-types + max-nb-declared-types

OctetValue                      ::=     OCTET STRING (SIZE (1))
ShortValue                      ::=     INTEGER (-32768 .. 32767)
LongValue                       ::=     INTEGER (-2147483648 .. 2147483647)
UnsignedShortValue              ::=     INTEGER (0 .. 65535)
UnsignedLongValue               ::=     INTEGER (0 .. 4294967295)
FloatValue                      ::=     REAL
DoubleValue                     ::=     REAL
BooleanValue                    ::=     BOOLEAN
CharacterValue                  ::=     BMPString (SIZE (1))
StringValue                     ::=     BMPString (SIZE (0.. max-size-string))

TypeIdentifier                  ::=     INTEGER (0 .. < max-nb-types)
DataIdentifier                  ::=     INTEGER (0 .. < max-nb-data)
FunctionIdentifier              ::=     INTEGER (0 .. < max-nb-functions)
MessageIdentifier               ::=     INTEGER (0 .. < max-nb-messages)
PackageIdentifier               ::=     INTEGER (0 .. < max-nb-packages)

END
```

Annex B
(normative)
Coded representation of interchanged scripts

B.1 Coding for interchanged scripts

Interchanged scripts shall be encoded according to the ASN.1 Distinguished Encoding Rules (DER) as specified by ISO/IEC 8825-1 [2].

NOTE: This is intended to make the MHEG-3 engine's decoding task as efficient as possible by removing all ASN.1 encoding options that might delay or complicate it.

B.2 Coding for the program code

The value of the `program-code` component of the `RoutineDeclaration` type defined by `ISOMHEG-sir` (see Annex A) shall be encoded according to the rules defined in this Clause.

The sequence of instructions that make up the program code of a routine shall be encoded as a sequence of octets. The order of encoding will be the same as the order in which the instructions are intended to be executed.

Each instruction shall be encoded using one octet for the op-code, followed by zero to three octets for the operands, depending on the op-code.

B.2.1 Instruction op-codes

The op-codes shall be encoded using the bitstring defined by Table B.1.

B.2.2 Instruction operands

According to the op-code of the instruction, the operands shall have the length and encoding defined by Table B.1. All multiple-byte operands shall be encoded in big-endian order, i.e. most significant byte first.

B.2.2.1 `Data identifier` operands

DID operands shall be encoded using two octets as follows:

- if bit 16 is '1' and bits 15 to 9 are '0', the DID shall reference a local variable, where bits 8 to 1 represent the local variable index (from 0 to 255)
- if bit 16 is '1' otherwise, the DID shall reference a dynamic variable, where bits 15 to 1 represent the dynamic variable index (from 0 to 32511) incremented by 256;
- if bits 16 to 13 are '0000', the DID shall reference a constant, where bits 12 to 1 represent the constant index (from 0 to 4095);
- otherwise, the DID shall reference a global variable, where bits 15 to 1 represent the global variable index (from 0 to 28671) incremented by 4096.

B.2.2.2 Function identifier operands

FID operands shall be encoded on two octets as follows:

- if bits 16 to 13 are '0000', the FID shall reference a routine, where bits 12 to 1 represent the routine index (from 0 to 4095);
- if bits 16 and 15 are '00' otherwise, the FID shall reference a predefined function, where bits 14 to 1 represent the predefined function index (from 0 to 12287) incremented by 4096;

- otherwise, the FID shall reference a service, where bits 16 to 9 represent the package identifier (from 0 to 191) incremented by 64, and where bits 8 to 1 represent the service index (from 0 to 255) within this package.

B.2.2.3 Miscellaneous numeric operands

1-octet "offset" operands shall be encoded in complement-to-one notation on 1 octet: bit 8 represents the direction of movement, bits 7 to 1 represent the number of units to shift in that direction.

2-octet "offset" operands shall be encoded in complement-to-one notation on 2 octets: bit 16 represents the direction of movement, bits 15 to 1 represent the number of units to shift in that direction.

"Value" operands shall be encoded in complement-to-two notation on two octets, for interpretation as signed integer values.

"Index" operands shall be encoded on one octet, for interpretation as unsigned integer values.

Table B.1: Encoding of MHEG-SIR instructions

Instruction mnemonics	Op-code (binary)	Opcode (hexa)	Op1 length	Op1 encoding	Op2 length	Op2 encoding
NOP	0000 0000	00	0			
YIELD	0000 0010	02	0			
RET	0000 0011	03	0			
FREE	0000 1000	08	0			
NOT_B	0001 0000	10	0			
NOT_O	0001 0001	11	0			
NOT_W	0001 0010	12	0			
NOT_U	0001 0011	13	0			
OR_B	0001 0100	14	0			
OR_O	0001 0101	15	0			
OR_W	0001 0110	16	0			
OR_U	0001 0111	17	0			
XOR_B	0001 1000	18	0			
XOR_O	0001 1001	19	0			
XOR_W	0001 1010	1A	0			
XOR_U	0001 1011	1B	0			
AND_B	0001 1100	1C	0			
AND_O	0001 1101	1D	0			
AND_W	0001 1110	1E	0			
AND_U	0001 1111	1F	0			
EQR	0010 0000	20	0			
EQ_O	0010 0001	21	0			
EQ_S	0010 0010	22	0			
EQ_L	0010 0011	23	0			
EQ_W	0010 0100	24	0			
EQ_U	0010 0101	25	0			
EQ_F	0010 0110	26	0			
EQ_D	0010 0111	27	0			
EQ_B	0010 1000	28	0			
EQ_C	0010 1001	29	0			
EQ_I	0010 1010	2A	0			
EQ_R	0010 1011	2B	0			
LT_C	0011 0000	30	0			
LT_O	0011 0001	31	0			
LT_S	0011 0010	32	0			
LT_L	0011 0011	33	0			
LT_W	0011 0100	34	0			
LT_U	0011 0101	35	0			
LT_F	0011 0110	36	0			
LT_D	0011 0111	37	0			
GT_C	0011 1000	38	0			
GT_O	0011 1001	39	0			
GT_S	0011 1010	3A	0			
GT_L	0011 1011	3B	0			
GT_W	0011 1100	3C	0			

GT_U	0011 1101	3D	0			
GT_F	0011 1110	3E	0			
GT_D	0011 1111	3F	0			
ADD_O	0100 0001	41	0			
ADD_S	0100 0010	42	0			
ADD_L	0100 0011	43	0			
ADD_W	0100 0100	44	0			
ADD_U	0100 0101	45	0			
ADD_F	0100 0110	46	0			
ADD_D	0100 0111	47	0			
SUB_O	0100 1001	49	0			
SUB_S	0100 1010	4A	0			
SUB_L	0100 1011	4B	0			
SUB_W	0100 1100	4C	0			
SUB_U	0100 1101	4D	0			
SUB_F	0100 1110	4E	0			
SUB_D	0100 1111	4F	0			
MUL_O	0101 0001	51	0			
MUL_S	0101 0010	52	0			
MUL_L	0101 0011	53	0			
MUL_W	0101 0100	54	0			
MUL_U	0101 0101	55	0			
MUL_F	0101 0110	56	0			
MUL_D	0101 0111	57	0			
DIV_O	0101 1001	59	0			
DIV_S	0101 1010	5A	0			
DIV_L	0101 1011	5B	0			
DIV_W	0101 1100	5C	0			
DIV_U	0101 1101	5D	0			
DIV_F	0101 1110	5E	0			
DIV_D	0101 1111	5F	0			
NEG_S	0110 0010	62	0			
NEG_L	0110 0011	63	0			
NEG_F	0110 0110	66	0			
NEG_D	0110 0111	67	0			
REM_O	0111 1001	79	0			
REM_S	0111 1010	7A	0			
REM_L	0111 1011	7B	0			
REM_W	0111 1100	7C	0			
REM_U	0111 1101	7D	0			
DUP_O	1000 0001	81	0			
DUP_S	1000 0010	82	0			
DUP_L	1000 0011	83	0			
DUP_W	1000 0100	84	0			
DUP_U	1000 0101	85	0			
DUP_F	1000 0110	86	0			
DUP_D	1000 0111	87	0			
DUP_B	1000 1000	88	0			

DUP_C	1000 1001	89	0			
DUP_I	1000 1010	8A	0			
DUP_R	1000 1011	8B	0			
CVT_SW	1001 0100	94	0			
CVT_WS	1001 0101	95	0			
CVT_LU	1001 0110	96	0			
CVT_UL	1001 0111	97	0			
CVT_CW	1001 1010	9A	0			
CVT_WC	1001 1011	9B	0			
CVT_BS	1010 0000	A0	0			
CVT_OS	1010 0001	A1	0			
CVT_SL	1010 0010	A2	0			
CVT_LF	1010 0011	A3	0			
CVT_WL	1010 0100	A4	0			
CVT_UF	1010 0101	A5	0			
CVT_FD	1010 0110	A6	0			
CVT_BO	1010 1000	A8	0			
CVT_OW	1010 1001	A9	0			
CVT_SU	1010 1010	AA	0			
CVT_WU	1010 1100	AC	0			
CVT_OB	1011 0001	B1	0			
CVT_SB	1011 0010	B2	0			
CVT_LB	1011 0011	B3	0			
CVT_WB	1011 0100	B4	0			
CVT_UB	1011 0101	B5	0			
CVT_WO	1011 1001	B9	0			
CVT_LS	1011 1010	BA	0			
CVT_FL	1011 1011	BB	0			
CVT_UW	1011 1100	BC	0			
CVT_FU	1011 1101	BD	0			
CVT_DF	1011 1110	BE	0			
JT	1100 0000	C0	1	(signed) offset		
JF	1100 0001	C1	1	(signed) offset		
JMP	1100 0010	C2	1	(signed) offset		
SHIFT_O	1100 0101	C5	1	(signed) offset		
SHIFT_W	1100 0110	C6	1	(signed) offset		
SHIFT_U	1100 0111	C7	1	(signed) offset		
GETOR	1100 1001	C9	1	package identifier		
LJT	1101 0000	D0	2	(signed) offset		
LJF	1101 0001	D1	2	(signed) offset		
LJMP	1101 0010	D2	2	(signed) offset		
CALL	1101 0100	D4	2	function identifier		
XCALL	1101 0110	D6	2	function identifier		
PUSH	1110 0000	E0	2	data identifier		
PUSHR	1110 0001	E1	2	data identifier		
PUSHI	1110 0011	E3	2	(signed) value		
POP	1110 0100	E4	2	data identifier		
POPR	1110 0101	E5	2	data identifier		

POPC	1110 0110	E6	2	data identifier		
ALLOC	1110 1000	E8	2	type identifier		
INC	1110 1100	EA	2	data identifier		
DEC	1110 1101	EB	2	data identifier		
GET	1111 0000	F0	2	data identifier	1	(unsigned) index
GETC	1111 0010	F2	2	data identifier	1	(unsigned) index
SET	1111 0100	F4	2	data identifier	1	(unsigned) index
SETC	1111 0110	F6	2	data identifier	1	(unsigned) index

Annex C
(normative)
MHEG-SIR predefined elements

This Annex lists the predefined types, functions and messages of MHEG-SIR, together with their corresponding indices.

Predefined types, functions and messages may be referenced by their identifier and used within interchanged scripts in the same way types, functions and messages declared within the global declarations part of interchanged scripts would.

C.1 Predefined types

MHEG-SIR predefined types comprise

- primitive types
- MHEG API types.

C.1.1 Primitive types

The primitive types defined by this part of ISO/IEC 13522 shall be encoded using predefined type identifiers as listed in Table C.1:

Table C.1: Predefined type identifiers for primitive types

Type name	Type identifier
`void`	0
`octet`	1
`short`	2
`long`	3
`unsigned short`	4
`unsigned long`	5
`float`	6
`double`	7
`boolean`	8
`character`	9
`data identifier`	10
`object reference`	11

All types that may be expressed in MHEG-SIR (including predefined MHEG types) can be built using the MHEG-SIR primitive types and the following constructors:

- `string`;
- `sequence`;
- `array`;
- `structure`;
- `union`.

By convention, the unbounded `string` type (the only constructed type without an element or a parameter) shall be predefined and shall have 12 as its type identifier.

C.1.2 MHEG API types

The MHEG API types defined by the MHEG API shall be encoded using predefined type identifiers.

NOTE: MHEG API types are intended for use by interchanged scripts to express information which is exchanged between the script interpreter and MHEG entities.

The IDL definition of these types, as provided by an MHEG API, shall be mapped to MHEG-SIR type descriptions using the general IDL mapping rules defined in Clause 14 and the specific MHEG API mapping rules defined in Clause E.2.

C.2 Predefined functions

MHEG-SIR predefined functions comprise

- MHEG API operations;
- MHEG-3 API operations.

C.2.1 MHEG API operations

The MHEG API operations defined by the MHEG API shall be encoded using predefined function identifiers.

Predefined message identifiers for the MHEG API operations shall start at 1100h.

The IDL definition of these operations, as provided by the MHEG-3 API, shall be mapped to MHEG-SIR function descriptions using the general IDL mapping rules defined in Clause 14 and the specific MHEG API mapping rules defined in Clause E.2.

C.2.2 MHEG-3 API operations

The MHEG-3 API operations defined by the MHEG-3 API, as defined in Clause 15, shall be encoded using predefined function identifiers according to Table C.2:

Table C.2: Predefined function identifiers for MHEG-3 API operations

Operation name	Predefined function index	Function identifier
prepare	0	1000h
destroy	1	1001h
new	2	1002h
delete	3	1003h
setPriority	4	1004h
getPriority	5	1005h
setData	6	1006h
getData	7	1007h
allocate	8	1008h
free	9	1009h
stop	10	100Ah
reInit	11	100Bh
getRtScriptStatus	12	100Ch
open	13	100Dh
close	14	100Eh
getRoutineId	15	100Fh
setParameter	16	1010h
getPrototype	17	1011h
run	18	1012h
reset	19	1013h
getInvocationStatus	20	1014h

The IDL definition of these operations, as defined in Annex F, shall be mapped to MHEG-SIR function descriptions using the IDL mapping rules defined in Clause 14.

C.3 Predefined messages

MHEG-SIR predefined messages targeted at an rt-script result from

- invocation of the MHEG-3 API `run` operation;
- the `InstructionExecutionError` exception;
- MHEG-3 API exceptions;
- MHEG API exceptions.

C.3.1 MHEG-3 API operations

The identifier of the message resulting from the invocation of a `run` operation, as defined in subclause 15.4.5, shall be equal to the function identifier of the target routine.

Messages resulting from MHEG-3 API operations shall therefore have a message identifier value between 0 and 0FFFh.

C.3.2 The InstructionExecutionError exception

The `InstructionExecutionError` exception, as defined in subclause 9.5.2, shall have 1000h as its message identifier.

The `InstructionExecutionError` exception shall have one member of type `unsigned long`, whose value shall be set to the value of the ER.

The major error code shall determine the least significant byte of the member (and the ER) as defined by by Table C.3:

Table C.3: Instruction execution error codes

Error name	Error code
InvalidOperand	1
InvalidParameter	2
InvalidType	3
InvalidIdentifier	4
InvalidLevel	5
InvalidIndex	6
StackUnderflow	7
ArithmeticOverflow	8
DivisionByZero	9
HandlerNotFound	10
InvalidReturnValue	11
BadPackageStatus	12
InvalidObjectReference	13
TypeMismatch	14
JumpOutOfRange	15
AllocationFailed	16

C.3.3 MHEG-3 API exceptions

The MHEG-3 API exceptions, as defined in Clause 15, shall have the message identifiers defined by Table C.4:

Table C.4: Predefined message identifiers for the MHEG-3 API exceptions

Exception name	Predefined message index	Message identifier
InvalidScript	1	1001h
InvalidParameter	2	1002h
OperationFailed	3	1003h

The IDL definition of these exceptions, as defined in Annex F, shall be mapped to MHEG-SIR message descriptions using the IDL mapping rules defined in Clause 14.

C.3.4 MHEG API exceptions

The MHEG API exceptions defined by the MHEG API shall be encoded using predefined message identifiers.

Predefined message identifiers for the MHEG API exceptions shall start at 1100h.

The IDL definition of these exceptions, as provided by the MHEG API, shall be mapped to MHEG-SIR message descriptions using the general IDL mapping rules defined in Clause 14 and the specific MHEG API mapping rules defined in Clause E.2.

Annex D
(normative)
IDL Platform mapping specification form

MHEG-3 engines shall allow access to the services provided by the run-time environment of a given platform, provided this run-time environment complies with the registered "platform mapping specification" for this platform.

The registered "platform mapping specifications" shall be provided according to the template specified in this Annex, with all fields being completed.

This MHEG-SIR platform-mapping specification defines the mechanisms that need be used by MHEG-3 engines to access the services provided by the run-time environment on the platform.

Platform description

The platform to which this specification applies is <platform_description>.

Package availability procedure

To know whether an IDL specification is available within the run-time environment and to locate it, an MHEG-3 engine shall proceed as follows. <package_availability_procedure>

Package load procedure

To make the operations of an available IDL specification accessible, an MHEG-3 engine shall proceed as follows. <package_load_procedure>

Package unload procedure

To stop the operations of an available IDL specification from being accessible, an MHEG-3 engine shall proceed as follows. <package_unload_procedure>

Operation invocation procedure

To invoke an operation of an accessible IDL specification, an MHEG-3 engine shall proceed as follows. <operation_invocation_procedure>

Parameter passing procedure

When invoking an IDL operation, an MHEG-3 engine shall pass **in** parameters as follows. <in_parameter_passing _procedure>

When invoking an IDL operation, an MHEG-3 engine shall pass **out** parameters as follows. <out_parameter_passing _procedure>

When invoking an IDL operation, an MHEG-3 engine shall pass **inout** parameters as follows. <inout_parameter_passing _procedure>

Output parameter retrieval procedure

To retrieve the values of **out** or **inout** parameters after invoking an IDL operation, an MHEG-3 engine shall proceed as follows. <output_parameter_retrieval_procedure>

Return value retrieval procedure

To retrieve the return value of a previously invoked IDL operation, an MHEG-3 engine shall proceed as follows. <return_value _retrieval_procedure>

Data encoding rules

The values of data that are interchanged between the MHEG-3 engine and the run-time environment shall be encoded as follows. <data_encoding_rules>

Exception retrieval procedure

To retrieve exceptions that are raised by the run-time environment, an MHEG-3 engine shall proceed as follows. <exception_retrieval_procedure>

System exceptions

The system exceptions that may be raised by the run-time environment and retrieved by an MHEG-3 engine are defined as follows. <system_exception_definitions>

Resource limitations

When using the run-time environment on the platform, the following resource limitations apply. <resource_limitations_statement>

Annex E
(normative)
MHEG API definition process

As exposed in subclause 8.3.3, this generic part of ISO/IEC 13522 does not define a specific MHEG API. It defines instead a generic set of rules and procedures applicable to the definition of the MHEG API to be provided by any part of ISO/IEC 13522 that describes presentation objects. This comprises

- the rules that shall be used to produce the MHEG API definition (see Clause E.1);
- the procedure that shall be used to define the MHEG-SIR mapping of this MHEG API (see Clause E.2).

E.1 Generic API definition framework

Producing an MHEG API specification from another part of ISO/IEC 13522 that describes presentation objects (hereafter called an MHEG specification) is a process that consists in producing IDL elements from MHEG elements.

The MHEG elements on which this process applies are described in subclause E.1.1. The IDL elements to be produced from these MHEG elements are described in subclause E.1.2. The rules used to produce the IDL elements from the MHEG elements are described in subclause E.1.3 sq.

E.1.1 MHEG elements input to MHEG API definition process

The different parts of ISO/IEC 13522 share a number of key features. The following MHEG elements must be present in the source MHEG specification:

- MHEG data types, described using ASN.1 or Extended Backus-Naur Form (EBNF);
- MHEG entities (i.e. objects targeted by MHEG elementary actions), related to each other by inheritance relationships;
- static and dynamic attributes of MHEG entities;
- MHEG elementary actions applying to MHEG entities;
- MHEG exceptions raised as the MHEG effect of elementary actions.

E.1.2 IDL elements output by MHEG API definition process

The API definition process should consist in mapping these elements to a set of IDL elements:

- IDL non-object types shall map MHEG data types;
- IDL object interfaces, related to each other by inheritance relationships, shall map MHEG entities;
- IDL attributes, provided by IDL object interfaces, shall map static and dynamic attributes of MHEG entities;
- IDL operations, provided by IDL object interfaces, shall map MHEG elementary actions;
- IDL exceptions shall map MHEG exceptions raised as the effect of elementary actions.

E.1.3 Requirements on the MHEG API definition process

According to ISO/IEC JTC1 guidelines for API standardisation, the MHEG API shall be defined as an abstract API specification, i.e. a language-independent description of the semantics of a set of functionality in an abstract syntax using abstract data types.

As an enforcement of the recommendations of ETR 225 "API and script representation for MHEG - Requirements and framework", an MHEG API definition shall meet the following requirements:

- portability (see subclause E.1.3.1);

- genericity (see subclause E.1.3.2);
- conformance testability (see subclause E.1.3.3);
- implementability (see subclause E.1.3.4).

E.1.3.1 Portability

The **portability** requirement states that MHEG applications need use the MHEG object manipulation and interchange service provided by MHEG engines (i.e. an MHEG API) in a way independent of

- the programming language used for the MHEG application;
- the underlying operating system.

To meet the portability requirement, an MHEG API shall be defined as an abstract API specification.

E.1.3.2 Genericity

The **genericity** requirement states that all the common requirements of MHEG applications need be supported by an MHEG API.

To meet the genericity requirement, an MHEG API shall be defined at the most primitive level, i.e. in terms of primitives that match MHEG elementary actions and data types that match MHEG data types. This guarantees to maximise the range of MHEG object manipulations made available to applications.

E.1.3.3 Conformance testability

The **conformance testability** requirement states that it should be as easy as possible to test

- the conformance of an MHEG engine to an MHEG API specification, i.e. the correct provision of this API by an MHEG engine under test;
- the conformance of an MHEG application to an MHEG API specification, i.e. the correct use of this API by an MHEG application under test.

To meet the conformance testability requirement, an MHEG API shall express formally its requirements on conforming implementations and conforming applications and it shall use a formal description technique for the definition of the MHEG API.

E.1.3.4 Implementability

The **implementability** requirement states that implementation of MHEG engines that conform to the MHEG API specification need be as easy as possible. For this purpose, the MHEG API definition should take into account simplicity and clarity both in the definition and the formulation.

To meet the implementability requirement, an MHEG API shall provide or refer to guidelines to produce language mapping specifications and message encoding rules from the abstract API specification.

E.1.3.5 Fulfilment of technical requirements

The use of IDL contributes to the fulfilment of the portability and implementability technical requirements:

- IDL is independent from a programming language. Moreover, there are public specifications of IDL mappings to common programming languages such as C and C++;

- IDL provides a complete formal description language which allows a very concise, readable, efficient specification of an MHEG API. Moreover, IDL is also appropriate for automatic compilation, so that MHEG API implementations may be automatically generated for a given language and operating system using appropriate IDL compilers.