

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)

AVA **Documents**
Document Sharing Hub

Contents

1 Scope 1

1.1 Context of the scope 1

1.2 Scope of this part of ISO/IEC 13522 1

2 Normative references 1

3 Definitions 2

4 Abbreviations 6

5 Conformance 7

5.1 Information object conformance 7

5.1.1 Profiles 7

5.1.2 Encoding 7

5.1.3 Syntax 8

5.1.4 Semantics 8

5.2 Implementation conformance 8

5.2.1 Conformance requirements 8

5.2.2 Conformance documentation 8

5.3 Application conformance 9

5.4 Test Methods 9

6 Overview 9

6.1 Description methodology 9

6.2 Data processing operations 10

6.3 Access to external data and functions 10

7 MHEG/MHEG-3 relationship 11

7.1 MHEG entities 11

7.2 Functional entities 11

7.3 MHEG-SIR script interpreter 12

8 Elements of MHEG-SIR 12

8.1 Data types 12

8.1.1 Predefined types 13

8.1.1.1 Primitive types 13

8.1.1.1.1 void type 14

8.1.1.1.2 octet type 14

8.1.1.1.3 short type 14

8.1.1.1.4 long type 14

8.1.1.1.5 unsigned short type 14

© ISO/IEC 1997

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

		8.1.1.1.6	unsigned long type	14	
		8.1.1.1.7	float type	14	
		8.1.1.1.8	double type	14	
		8.1.1.1.9	boolean type	14	
		8.1.1.1.10	character type	14	
		8.1.1.1.11	data identifier type	14	
		8.1.1.1.12	object reference type	15	
	8.1.1.2		Predefined constructed types	15	
8.1.2			Declared constructed types	15	
	8.1.2.1		sequence types	15	
	8.1.2.2		string types	16	
	8.1.2.3		array types	16	
	8.1.2.4		structure types	17	
	8.1.2.5		union types	17	
8.2	Data			17	
	8.2.1		Immediate values	18	
	8.2.2		Constants	18	
	8.2.3		Variables	18	
		8.2.3.1	Global variables	19	
		8.2.3.2	Local variables	19	
		8.2.3.3	Dynamic variables	19	
8.3	Functions			19	
	8.3.1		Routines	20	
	8.3.2		Services	20	
	8.3.3		Predefined functions	21	
8.4	Messages			21	
	8.4.1		Package exceptions	21	
	8.4.2		Predefined messages	22	
8.5	Instructions			22	
8.6	Identifiers			22	
	8.6.1		Type identifiers	22	
	8.6.2		Data identifiers	23	
	8.6.3		Function identifiers	23	
	8.6.4		Message identifiers	23	
9	The MHEG-SIR virtual machine			23	
9.1	Structure of the MHEG-SIR virtual machine			24	
9.2	Structures and notations			24	
	9.2.1		Table	24	
	9.2.2		Stack	24	
	9.2.3		Parameter stack	25	
	9.2.4		Queue	25	
	9.2.5		Data representation	25	
9.3	Memory areas			26	
	9.3.1		Mh-script memory areas	26	
		9.3.1.1	Data areas	27	
			9.3.1.1.1	Type definition table	27
			9.3.1.1.2	Constant table	27
			9.3.1.1.3	Global variable definition table	27
		9.3.1.2	Code areas	27	
			9.3.1.2.1	Routine definition table	27
			9.3.1.2.2	Package definition table	28
			9.3.1.2.3	Service definition table	28
			9.3.1.2.4	Exception definition table	28
			9.3.1.2.5	Handler definition table	29
			9.3.1.2.6	Program code area	29

9.3.2	Rt-script memory areas.....	29
9.3.2.1	Dynamic memory areas.....	29
9.3.2.1.1	Variable table.....	29
9.3.2.1.2	Call stack.....	30
9.3.2.1.3	Parameter stack.....	30
9.3.2.1.4	Message queue.....	31
9.3.2.1.5	Heap.....	31
9.3.2.2	Registers.....	31
9.3.2.2.1	Instruction pointer register.....	32
9.3.2.2.2	Instruction register.....	32
9.3.2.2.3	Error register.....	32
9.3.2.2.4	Stack pointer register.....	32
9.3.2.2.5	Frame pointer register.....	32
9.3.2.2.6	Queue pointer register.....	32
9.3.2.2.7	Function register.....	32
9.4	Script statuses.....	33
9.4.1	Mh-script statuses.....	33
9.4.1.1	Not available.....	33
9.4.1.2	Available.....	33
9.4.2	Rt-script statuses.....	33
9.4.2.1	Not ready.....	33
9.4.2.2	Ready.....	33
9.4.2.3	Running.....	34
9.4.2.4	Erroneous.....	34
9.5	Processing units.....	34
9.5.1	Message reception.....	34
9.5.1.1	MHEG-3 API operations.....	34
9.5.1.2	External exception.....	34
9.5.1.3	InstructionExecutionError exception.....	35
9.5.1.4	MHEG-3 API exception.....	35
9.5.2	Mh-script initialisation.....	35
9.5.3	Rt-script initialisation.....	35
9.5.4	Rt-script execution unit.....	36
9.5.5	MHEG-SIR instruction execution unit.....	36
10	Provisions for run-time environment access.....	36
10.1	General model.....	36
10.2	Declaration of IDL interfaces.....	37
10.3	Invocation of external operations in an MHEG-SIR program.....	38
10.4	Handling of external exceptions in an MHEG-SIR program.....	38
10.5	Invocation of external operations by an MHEG-3 engine.....	38
10.6	Handling of external exceptions by an MHEG-3 engine.....	38
10.7	Platform mapping specifications.....	39
11	Provisions for MHEG object manipulation.....	39
11.1	Invoking MHEG actions.....	39
11.1.1	Sending messages to other scripts.....	39
11.1.2	Exchange of information with MHEG objects.....	40
11.2	Receiving MHEG messages.....	40
11.2.1	MHEG-3 API run operations.....	40
11.2.2	MHEG API exceptions.....	40
12	MHEG-SIR declarations.....	40
12.1	Type declaration.....	41
12.1.1	Type identifier.....	41
12.1.2	Type description.....	41

	12.1.2.1	String description.....	42
	12.1.2.2	Sequence description.....	42
	12.1.2.3	Array description.....	42
	12.1.2.4	Structure description.....	42
	12.1.2.5	Union description.....	42
12.2	Constant declaration.....		43
	12.2.1	Data identifier.....	43
	12.2.2	Type identifier.....	43
	12.2.3	Constant value.....	43
12.3	Global variable declaration.....		44
	12.3.1	Data identifier.....	44
	12.3.2	Type identifier.....	44
	12.3.3	Constant reference.....	44
12.4	Package declaration.....		45
	12.4.1	Package identifier.....	45
	12.4.2	Name.....	45
	12.4.3	Service description.....	45
	12.4.3.1	Function identifier.....	45
	12.4.3.2	Name.....	46
	12.4.3.3	Calling mode.....	46
	12.4.3.4	Type identifier.....	46
	12.4.3.5	Parameter description.....	46
		12.4.3.5.1 Passing mode.....	46
		12.4.3.5.2 Type identifier.....	47
	12.4.4	Exception description.....	47
	12.4.4.1	Message identifier.....	47
	12.4.4.2	Parameter description.....	47
12.5	Handler declaration.....		47
	12.5.1	Message identifier.....	48
	12.5.2	Function identifier.....	48
12.6	Routine declaration.....		48
	12.6.1	Function identifier.....	48
	12.6.2	Type identifier.....	48
	12.6.3	Parameter description.....	48
	12.6.3.1	Passing mode.....	49
	12.6.3.2	Type identifier.....	49
	12.6.4	Local variable declaration.....	49
	12.6.4.1	Data identifier.....	49
	12.6.4.2	Type identifier.....	49
	12.6.4.3	Constant reference.....	49
	12.6.5	Program code.....	50
13	MHEG-SIR instructions.....		50
13.1	Presentation methodology.....		50
	13.1.1	Error conditions.....	50
	13.1.2	Formal specification.....	51
	13.1.3	Data table notation.....	51
	13.1.4	Template instruction notation.....	51
	13.1.5	Primitives.....	52
13.2	Classification of MHEG-SIR instructions.....		52
13.3	Description of instructions.....		54
	13.3.1	No operation.....	54
	13.3.2	Yield.....	54
	13.3.3	Return.....	54
	13.3.4	Free.....	55
	13.3.5	Not.....	55

13.3.6	Or	56
13.3.7	Exclusive or	56
13.3.8	And	57
13.3.9	Equal reference	57
13.3.10	Equal	58
13.3.11	Less than	58
13.3.12	Greater than	59
13.3.13	Add	59
13.3.14	Subtract	59
13.3.15	Multiply	60
13.3.16	Divide	60
13.3.17	Negate	61
13.3.18	Remainder	61
13.3.19	Duplicate	62
13.3.20	Convert	62
13.3.21	Jump on true	62
13.3.22	Jump on false	63
13.3.23	Jump	63
13.3.24	Shift	64
13.3.25	Get object reference	64
13.3.26	Long jump on true	65
13.3.27	Long jump on false	65
13.3.28	Long jump	65
13.3.29	Call	66
13.3.30	External call	67
13.3.31	Push	68
13.3.32	Push reference	69
13.3.33	Push immediate	69
13.3.34	Pop	70
13.3.35	Pop reference	70
13.3.36	Pop contents	70
13.3.37	Allocate	71
13.3.38	Increment	71
13.3.39	Decrement	72
13.3.40	Get	72
13.3.41	Get contents	73
13.3.42	Set	74
13.3.43	Set contents	75
13.4	Type conversion rules	75
13.4.1	Reversible conversions	76
13.4.2	Lossless extensions	76
13.4.2.1	Conversions from boolean	76
13.4.2.2	Conversions from octet to a numeric type	76
13.4.2.3	Lossless conversions from a numeric to a larger numeric type	76
13.4.3	Lossy extensions	77
13.4.4	Truncations to boolean	77
13.4.5	Truncations between integer or between floating-point types	77
13.4.6	Truncations from floating-point to integer	77
14	IDL mapping to MHEG-SIR	77
14.1	IDL specifications	77
14.2	IDL interfaces and modules	78
14.3	IDL operations	78
14.3.1	Operation name	78
14.3.2	Operation parameters	78

14.3.3	Implicit parameter.....	78
14.3.4	Return value.....	78
14.4	IDL attributes.....	78
14.4.1	Accessor.....	79
14.4.2	Modifier.....	79
14.4.3	Readonly attribute.....	79
14.5	IDL inherited operations.....	79
14.6	IDL exceptions.....	79
14.6.1	Exception name.....	79
14.6.2	Exception members.....	79
14.6.3	Implicit member.....	79
14.7	IDL types.....	80
14.7.1	char type.....	80
14.7.2	enum type.....	80
14.7.3	Constructed types.....	80
14.7.4	any type.....	81
14.7.5	Restrictions on types.....	81
14.8	IDL constants.....	81
15	The MHEG-3 API.....	81
15.1	ScriptInterpreter object.....	81
15.1.1	kill operation.....	82
15.1.2	prepare operation.....	82
15.2	MhScript object.....	83
15.2.1	destroy operation.....	83
15.2.2	new operation.....	83
15.3	RtScript object.....	84
15.3.1	delete operation.....	84
15.3.2	setPriority operation.....	84
15.3.3	getPriority operation.....	84
15.3.4	setData operation.....	85
15.3.5	getData operation.....	85
15.3.6	allocate operation.....	86
15.3.7	free operation.....	86
15.3.8	stop operation.....	87
15.3.9	relnit operation.....	87
15.3.10	getRtScriptStatus operation.....	88
15.3.11	open operation.....	88
15.4	RoutineInvocation object.....	88
15.4.1	close operation.....	88
15.4.2	routine_id readonly attribute.....	89
15.4.3	setParameter operation.....	89
15.4.4	getPrototype operation.....	90
15.4.5	run operation.....	90
15.4.6	reset operation.....	91
15.4.7	getInvocationStatus operation.....	91
Annex A (normative) ASN.1 specification of interchanged scripts.....		92
Annex B (normative) Coded representation of interchanged scripts.....		95
B.1	Coding for interchanged scripts.....	95
B.2	Coding for the program code.....	95
B.2.1	Instruction op-codes.....	95
B.2.2	Instruction operands.....	95

B.2.2.1	Data identifier operands	95
B.2.2.2	Function identifier operands	95
B.2.2.3	Miscellaneous numeric operands	96
Annex C	(normative) MHEG-SIR predefined elements	101
C.1	Predefined types	101
C.1.1	Primitive types	101
C.1.2	MHEG API types	102
C.2	Predefined functions	102
C.2.1	MHEG API operations	102
C.2.2	MHEG-3 API operations	102
C.3	Predefined messages	103
C.3.1	MHEG-3 API operations	103
C.3.2	The InstructionExecutionError exception	103
C.3.3	MHEG-3 API exceptions	104
C.3.4	MHEG API exceptions	104
Annex D	(normative) IDL Platform mapping specification form	105
	Platform description	105
	Package availability procedure	105
	Package load procedure	105
	Package unload procedure	105
	Operation invocation procedure	105
	Parameter passing procedure	105
	Output parameter retrieval procedure	105
	Return value retrieval procedure	106
	Data encoding rules	106
	Exception retrieval procedure	106
	System exceptions	106
	Resource limitations	106
Annex E	(normative) MHEG API definition process	107
E.1	Generic API definition framework	107
E.1.1	MHEG elements input to MHEG API definition process	107
E.1.2	IDL elements output by MHEG API definition process	107
E.1.3	Requirements on the MHEG API definition process	107
E.1.3.1	Portability	108
E.1.3.2	Genericity	108
E.1.3.3	Conformance testability	108
E.1.3.4	Implementability	108
E.1.3.5	Fulfilment of technical requirements	108
E.1.4	General structure of the MHEG API	109
E.1.5	IDL non-object datatype definition	109
E.1.5.1	Name mapping	109
E.1.5.1.1	Data types	109
E.1.5.1.2	Components	109
E.1.5.1.3	Values	110
E.1.5.2	Type mapping	110
E.1.5.2.1	INTEGER	110
E.1.5.2.2	BOOLEAN	110
E.1.5.2.3	OCTET STRING	111
E.1.5.2.4	ENUMERATED	111

	E.1.5.2.5	SEQUENCE OF	111
	E.1.5.2.6	CHOICE	111
	E.1.5.2.7	SEQUENCE	112
	E.1.5.3	Order of declarations	112
E.1.6		IDL interface definition	114
E.1.7		IDL attribute definition	114
	E.1.7.1	MHEG interchanged attributes	114
	E.1.7.2	MHEG internal attributes	115
E.1.8		IDL operation definition	115
	E.1.8.1	Operations mapping MHEG elementary actions	115
	E.1.8.2	Operations enabling the deletion of an interface instance	116
	E.1.8.3	Operations to attach and detach an interface instance to a MHEG entity	117
E.1.9		IDL exception definition	117
E.2		MHEG API mapping to MHEG-SIR	118
Annex F (normative) IDL specification of the MHEG-3 API			119
Annex G (normative) Relationships with other parts of ISO/IEC 13522			121
G.1		Relationships with ISO/IEC 13522-1	121
G.2		Relationships with ISO/IEC 13522-5	122
Annex H (informative) MHEG-SIR syntax (EBNF notation)			123
Annex J (informative) Textual notation for MHEG-SIR scripts			125
Annex K (informative) MHEG entities			128
K.1		MHEG objects	128
K.2		Mh-objects	128
K.3		Rt-objects	128
K.4		Interchanged MHEG objects	129
Annex L (informative) Main features of MHEG-SIR			130
L.1		Features of using applications	130
	L.1.1	Manipulation of MHEG entities	130
	L.1.2	Computations, variable handling and control structures	130
	L.1.3	External device control	130
	L.1.4	Data acquisition	130
	L.1.5	Access to external data	131
	L.1.6	Access to arbitrary external run-time services	131
L.2		Functional features	131
	L.2.1	Data processing operations	131
	L.2.2	Access to external data and functions	131
L.3		Technical features	132
	L.3.1	Hardware independence	132
	L.3.2	Final form representation	133
	L.3.3	Compactness	133

L.3.4 Ease of implementation 133

L.3.5 Interpretation efficiency..... 133

L.3.6 Openness and extensibility 133

L.3.7 Non-revisability 134

L.3.8 Provisions for real-time interchange 134

L.3.9 Semantic validation for quality of service purposes..... 134

L.3.10 Syntax checkability (with regard to contamination hazards)..... 134

L.3.11 Non-proprietary representation..... 134

L.3.12 Secure script processing 134

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.

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 $((Y+64) \ll 8) + X$ 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 $((Y+64) \ll 8) + X$ 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 *n*th 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:

- `RT[FID].TID`: type of return value (expressed as a type identifier);
- `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 *i*th parameter;
 - 2) `RT[FID].sig[i].mod` is the passing mode (**value** or **reference**) of the *i*th 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 *i*th parameter;
 - 2) `ST[FID].sig[i].mod` is the passing mode (**in**, **inout** or **out**) of the *i*th 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 *i*th 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 an 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