

1 **IEEE 1484.11.2/D2**  
2 **Draft Standard for Learning Technology—ECMAScript**  
3 **Application Programming Interface for Content to**  
4 **Runtime Services Communication**

5 Sponsored by the Learning Technology Standards Committee  
6 of the IEEE Computer Society

7 Copyright © 2002 by the Institute of Electrical and Electronics Engineers, Inc.  
8 Three Park Avenue  
9 New York, NY 10016-5997, USA

10 All rights reserved. This document is an unapproved draft of a proposed IEEE Standard. As such,  
11 this document is subject to change. USE AT YOUR OWN RISK! Because this is an unapproved  
12 draft, this document must not be utilized for any conformance/compliance purposes. Permission is  
13 hereby granted for IEEE Standards Committee participants to reproduce this document for pur-  
14 poses of IEEE standardization activities only. Prior to submitting this document to another stan-  
15 dards development organization for standardization activities, permission must first be obtained  
16 from the Manager, Standards Licensing and Contracts, IEEE Standards Activities Department.  
17 Other entities seeking permission to reproduce this document, in whole or in part, must obtain  
18 permission from the Manager, Standards Licensing and Contracts, IEEE Standards Activities De-  
19 partment.

20 IEEE Standards Activities Department  
21 Standards Licensing and Contracts  
22 445 Hoes Lane, P.O. Box 1331  
23 Piscataway, NJ 08855-1331, USA

24 **Abstract:**

25 This Standard describes an ECMAScript application programming interface (API) for content-to-  
26 runtime-services communication. It is based on a current industry practice called "CMI—  
27 Computer Managed Instruction." This API enables the communication of information between  
28 content and a runtime service (RTS) typically provided by a learning management system (LMS)  
29 via common API services using the ECMAScript language. The purpose of this Standard is to  
30 build consensus around, resolve ambiguities, and correct defects in existing specifica-  
31 tions for an ECMAScript API for exchanging data between learning-related content and an LMS.

32 **Keywords:**

33 application programming interface, API, computer managed instruction, content object, ECMAS-  
34 cript, ECMAScript API, learning content, learning management system, LMS, runtime service.

## 35 **Introduction**

36 (This introduction is not a part of P1484.11.2, Draft Standard for ECMAScript API for Content to  
37 Runtime Services Communication.)

38 This document describes a learning content ECMAScript API to support the data transfer needs of  
39 content with a runtime service in a Web-browser-based content delivery environment. This docu-  
40 ment provides a starting point for specifying a data communication channel and methods to  
41 support the data transfer needs of training and human performance support content. The capabili-  
42 ties of this API model can be extended as other communication needs or methods arise.

## 43 **Participants**

44 At the time this Standard was completed, the working group had the following membership:

45 Tyde Richards, *Chair*

46 Jack Hyde, *Chair (December, 1988 – March, 2001)*

47 Scott Lewis, *Technical Editor*

48

49 T.B.D

50 The following persons were on the balloting committee: (To be provided by IEEE editor at time  
51 of publication.)

## 52 **Acknowledgements**

53 The IEEE LTSC P1484.11.2 CMI working group wishes to thank Tom King and Boyd Nielsen  
54 for their contributions of initial documents used for the preparation of this Standard.

55	Contents	
56	Introduction.....	2
57	Participants.....	2
58	Acknowledgements.....	2
59	1. Overview.....	5
60	1.1 Scope.....	5
61	1.2 Purpose.....	5
62	2. Normative references.....	6
63	3. Definitions.....	7
64	4. Conformance.....	9
65	4.1 Behavior.....	9
66	4.2 API implementation.....	9
67	4.3 Content object use of an API implementation.....	9
68	4.4 Outside of scope.....	10
69	5. Conceptual model.....	11
70	5.1 Simplified learning content ECMAScript API communication model.....	11
71	5.2 Basic Scenario.....	12
72	5.2.1 Environment and preparations.....	12
73	5.2.2 Sequence of operations.....	12
74	5.3 Implementation for Web-browser-based content.....	13
75	5.4 ECMAScript API extensibility.....	15
76	6. API instantiation and binding.....	16
77	6.1 Instantiation of an instance of an API implementation.....	16
78	6.2 Multiple instances.....	16
79	6.3 Binding a content object to an API instance.....	16
80	7. Content communication state model.....	18
81	7.1 Communication states.....	18
82	7.2 Events.....	18
83	7.2.1 Initialize communication session event.....	19
84	7.2.2 Terminate communication session event.....	19
85	7.2.3 Retrieve data event.....	20
86	7.2.4 Store data event.....	20
87	7.2.5 Commit data event.....	21
88	7.2.6 Get error code event.....	22
89	7.2.7 Get error string event.....	22
90	7.2.8 Get diagnostics event.....	22
91	8. ECMAScript API methods and syntax.....	24
92	8.1 Session methods.....	24
93	8.1.1 Initialize communication session method.....	24
94	8.1.2 Terminate communication session method.....	25
95	8.2 Data-transfer methods.....	26
96	8.2.1 Retrieve data method.....	26
97	8.2.2 Store data method.....	27
98	8.2.3 Commit data method.....	28
99	8.3 Support methods.....	29
100	8.3.1 Get error code.....	29
101	8.3.2 Get error string.....	30
102	8.3.3 Get API-implementation-specific diagnostics.....	31

103	8.4 API implementation error codes.....	32
104	Annex A .....	35
105	Annex B .....	36

106 **Draft Standard for Learning Technology—ECMAScript**  
107 **Application Programming Interface for Content to**  
108 **Runtime Services Communication**

109 **1. Overview**

110 The scope and purpose of this Standard are discussed in subclauses 1.1 and 1.2.

111 **1.1 Scope**

112 This Standard describes an ECMAScript application programming interface (API) for content-to-  
113 runtime-services communication. This Standard is based on an API defined in the "CMI Guide-  
114 lines for Interoperability," version 3.4 [A1]<sup>1</sup>, defined by the Aviation Industry CBT Committee  
115 (AICC). It defines common API services in the ECMAScript language that enable the communi-  
116 cation of information between learning-related content and a runtime service (RTS) used to sup-  
117 port learning management. This Standard does not address the data structures that may be trans-  
118 mitted, data security, or communication between an RTS and a related management system.

119

120 **1.2 Purpose**

121 There is widespread acknowledgement that the ECMAScript API for content-to-runtime-services  
122 communication defined in the AICC "CMI Guidelines for Interoperability", version 3.4, has broad  
123 applicability to systems used for learning management. The purpose of this Standard is to build  
124 consensus around, resolve ambiguities, and correct defects in this ECMAScript API for exchang-  
125 ing data between learning-related content and a runtime service used to support learning man-  
126 agement.

127

128

129

130

131

---

<sup>1</sup>The numbers in brackets correspond to those of the bibliography in Annex A.

132 **2. Normative references**

133 This Standard shall be used in conjunction with the following publication.

134 ISO/IEC 16262:1998, Information technology—ECMAScript language specification.

### 135 3. Definitions

136 For purposes of this Standard, the following terms and definitions apply. IEEE 100, *The*  
137 *Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition [A2], should be ref-  
138 erenced for terms not defined in this Clause.

139 **application programming interface (API):** A set of standard software interrupts, calls,  
140 functions, and data formats that can be used by an application program to access network  
141 services, devices, or operating systems.

142 **application programming interface implementation:** An implementation of this Stan-  
143 dard that *supplies* the services of the application programming interface, in contrast to an  
144 implementation of this Standard that *uses* the application programming.

145 **application programming interface instance:** An individual execution context and  
146 state of an application programming interface implementation. Note: The notion of "exe-  
147 cution context" in this Standard is same as in ECMAScript.

148 **communication session:** An active connection between a content object and an applica-  
149 tion program interface implementation.

150 **content object:** A collection of digital content that is intended for presentation to a  
151 learner by a learning system. A content object may include learning material and process-  
152 ing code. Example: A content object might be an HTML page with an embedded video  
153 clip and an ECMAScript.

154 **Document Object Model (DOM):** A platform- and language-neutral interface that will  
155 allow programs and scripts to dynamically access and update the content, structure and  
156 style of documents [A3]. The document can be further processed, and the results of that  
157 processing can be incorporated back into the presented page.

158 **ECMAScript:** An object-oriented programming language for performing computations  
159 and manipulating computational objects within a host environment as defined by  
160 ISO/IEC 16262:1998.

161 **implementation behavior:** Observable actions or appearance of an implementation. *See*  
162 *also: implementation-defined behavior; undefined behavior; unspecified behavior.*

163 **implementation-defined behavior:** Unspecified behavior for which each implementa-  
164 tion documents how the choice among the available alternatives is made. Example: Per-  
165 mitting a maximum size, as measured in octets, of a coding. *See also: implementation*  
166 *behavior; undefined behavior; unspecified behavior.*

167 **launch (v.):** To cause a content object to be delivered to a learner.

168 **learner:** An individual engaged in acquiring knowledge or skills with a learning tech-  
169 nology system.

- 170 **learning content:** Digital content intended for use in a learning experience.
- 171 **learning management system (LMS):** A computer system that may include the capa-  
172 bilities to register learners, schedule learning resources, control and guide the learning  
173 process, analyze and report learner performance, and schedule and track learners.
- 174 NOTE—Some implementations of LMSs also have the ability to launch and deliver content. For  
175 this Standard, these capabilities are known as a runtime service (RTS).
- 176 **runtime service (RTS):** Software that controls the execution and delivery of learning  
177 content and that may provide services such as resource allocation, scheduling, input-  
178 output control, and data management.
- 179 **undefined behavior:** Implementation behavior for which a standard imposes no re-  
180 quirements. Possible undefined behaviors include, but are not limited to: ignoring the  
181 situation completely, unpredictable results, behaving in a documented manner character-  
182 istic of the environment, and terminating processing. *See also:* **implementation behav-**  
183 **ior; implementation-defined behavior; unspecified behavior.**
- 184 **unspecified behavior:** Implementation behavior for which a standard provides two or  
185 more alternatives and imposes no further requirements on what is chosen in any instance.  
186 *See also:* **implementation behavior; implementation-defined behavior; undefined**  
187 **behavior.**
- 188

## 189 **4. Conformance**

190 Conformance to this Standard is discussed in subclauses 4.1 through 4.4.

### 191 **4.1 Behavior**

192 In this Standard, "shall" is to be interpreted as a requirement on an implementation; "shall not" is  
193 to be interpreted as a prohibition. This standard defines undefined behavior in the following ways:

- 194 – if a "shall" requirement or "shall not" prohibition is violated;
- 195 – by the words "undefined behavior"; and
- 196 – by the omission of any explicit definition of behavior.

197 There is no difference in emphasis among these ways; they all describe "behavior that is unde-  
198 fined."

199 NOTE 1—"Unspecified behavior" and "implementation-defined behavior" both describe conforming be-  
200 havior. With "unspecified behavior," several choices may be possible. This Standard does not specify  
201 which one is chosen, and the implementation *is not required* to "define" (i.e., tell in advance) which set of  
202 behaviors it has chosen. Similarly, with "implementation-defined behavior," several choices may be possi-  
203 ble and this Standard does not specify which one is chosen. However, the implementation *is required* to  
204 "define" (i.e., tell in advance) which set of behaviors it has chosen.

205 NOTE 2—"Implementation behavior" describes observable action or appearance, which may be conform-  
206 ing or nonconforming. "Undefined behavior" describes nonconforming implementations of this Standard.

### 207 **4.2 API implementation**

208 A conforming ECMAScript API implementation shall

- 209 – be instantiated as an object within the Document Object Model (DOM) [A4] environment  
210 of the content object as specified in Clause 6, API instantiation and binding;
- 211 – conform to the state model in Clause 7, Content communication state model; and
- 212 – implement all methods and error returns, as specified in Clause 8, ECMAScript API meth-  
213 ods and syntax.

### 214 **4.3 Content object use of an API implementation**

215 A conforming content object shall

- 216 – find an API implementation as an ECMAScript-compatible object within the DOM envi-  
217 ronment of the content object as described in Clause 6, API instantiation and binding;
- 218 – call `Initialize()` after the content object is launched and has located the API imple-  
219 mentation before calling any other method (see subclause 8.1.1);
- 220 – call `Terminate()` when the content object determines it no longer needs to communicate  
221 with the API implementation or is unloaded, whichever happens first (see subclause  
222 8.1.2);
- 223 – not call any data-transfer method or `Initialize()` after `Terminate()` has been called;  
224 and

- 225       – use conforming syntax when calling any optional data-transfer methods (see subclause  
226       8.2) or when calling any optional support methods (see subclause 8.3).

227 A content object strictly conforms to this Standard if it does not depend on implementation-  
228 defined behavior or unspecified behavior.

229 NOTE—It is anticipated, but not required, that a conforming content object will call one or more data-  
230 transfer methods. Without such calls, an RTS can only make inferences about the communication session,  
231 such as elapsed time, and specific data is neither explicitly received nor explicitly transmitted during the  
232 communication session.

## 233 **4.4 Outside of scope**

234 The following are outside of the scope of conformance to this Standard:

- 235       – the contents of the `GetDiagnostic("parameter")` implementation (see subclause  
236       8.3.3); (They are particular to the stakeholders interested in using this method and not to  
237       all implementers generally.)
- 238       – the semantics and behavior associated with specific data model elements, as associated  
239       with the retrieve data and store data methods (see subclauses 8.1.1 and 8.1.2);
- 240       – the choice of language of implementation for the API implementation and backend com-  
241       munication.
- 242       – whether a data-transfer cache is implemented and where it is implemented;
- 243       – behavior of the API implementation after abnormal termination of communication;
- 244       – the implementation of the API;
- 245       – how an API implementation processes a call to the commit method (see subclause 8.2.3);  
246       and
- 247       – how to initialize communication between an API implementation and a content object.

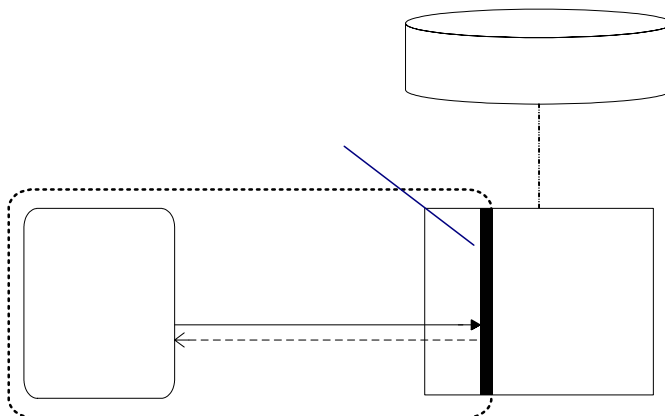
## 248 5. Conceptual model

249 This Clause presents a conceptual model for communication between a content object and an RTS  
 250 using an implementation of the ECMAScript API.

251 (This Clause is informative and not normative.)

### 252 5.1 Simplified learning content ECMAScript API communication model

253 Figure 1 shows how the learning content ECMAScript API implementation works at its most ba-  
 254 sic level. An RTS refers to that part of an environment, typically an LMS, that delivers content to  
 255 a learner.



256

257 **Figure 1—Learning content ECMAScript API model**

258 This conceptual model applies to both connected and disconnected delivery of content objects.  
 259 Possible implementations include

- 260 – the RTS resides on a server;
- 261 – the RTS resides on a client; and
- 262 – the RTS is distributed between a client and a server.

263 Regardless of how they are used, the concept is the same—a common set of communication  
 264 methods is implemented in the API implementation.

265 The common implementation of this model is in an environment (Web-browser-based delivery)  
 266 where a content object initiates all communication. This communication model makes no provi-  
 267 sion for communication initiated by an RTS to the content object.

268 Interoperability is achieved by sending requests through the shared communication methods im-  
 269 plemented in the API. Communication behavior and method interfaces are the same for every in-  
 270 stance of an API implementation. The contents of the communications may be whatever is mean-  
 271 ingful and agreed upon by different communities of practice. Nevertheless, the behavior of each  
 272 agent (e.g., content object, RTS) implementing an API may be specific to its role.

273 NOTE 1—Because of potential communication latency between an API implementation and persistent  
274 data storage, a common optimization may be a cache associated with the API implementation itself. This  
275 Standard provides a method that allows a content object to set the value of any number of data items and  
276 then signal the API implementation that the data values held in the cache, if it exists, can be committed to  
277 more permanent storage. Whether a cache is implemented and where it is implemented are outside of the  
278 scope of this Standard. Caching behavior is transparent to the content object. To the content object, it ap-  
279 pears that all gets and sets of data values are to persistent storage. How an API implementation processes a  
280 call to the commit method call is also outside of the scope of this Standard. Other than the commit method,  
281 this Standard does not specify any other caching methods or behaviors or any transaction methods or be-  
282 haviors. For example, there are no transaction start or rollback methods.

283 NOTE 2—A particular binding may specify that one or more data elements are session specific, read only,  
284 transient, or disposable. In such cases, the specified elements may not be available for future use, including  
285 being unavailable for use by a later communication session of the same content object by the same user.  
286 Writers of data binding specifications are encouraged to identify such special-case elements as well as all  
287 expected transformations of such elements through both single- and multiple-session transitions.

## 288 **5.2 Basic Scenario**

289 The basic scenario for communication is discussed subclauses 5.2.1 and 5.2.2.

### 290 **5.2.1 Environment and preparations**

291 As depicted in Figure 1, an RTS implements an API *in the content object's environment*. The im-  
292 plementer incorporates in the content object the capability to discover and communicate with an  
293 API implementation. An LMS or the front-end to a content repository (local or remote) provides  
294 an RTS for the learner. The RTS either delivers a content object to the learner and starts it, or  
295 launches a URI to initiate the content object.

296 NOTE—A content object has integrated procedures to locate an API implementation as described in sub-  
297 clause 5.2.2.

### 298 **5.2.2 Sequence of operations**

299 The RTS initiates the launch of a content object. As the content object starts up, it searches for the  
300 API implementation. After verifying that the API implementation is accessible in the content ob-  
301 ject's environment, the content object initializes a communication session through the instance  
302 that has been located.

303 All subsequent communication is part of this communication session until it is ended. The content  
304 object may request data through the API implementation. Through the API implementation, the  
305 RTS returns the requested data or a message identifying an error condition. While running, con-  
306 tent may send or set data-model data elements for storage across communication sessions. The  
307 RTS may use data elements or other data in reports on a learner's status with that content. The  
308 content object may elicit a more detailed error message.

309 The content object may continue communicating in this fashion, requesting and sending data until  
310 a learner finishes a content object, a learner terminates the communication session before finish-  
311 ing, or the communication session is abnormally terminated (e.g., loss of power, system crash). In  
312 the first two cases, the content object tells the API implementation that it is closing the communi-  
313 cation session. In the last case, the RTS will not receive a signal through the API implementation

314 that the communication session is closed. The RTS will need to deal with the abnormal event ac-  
315 cording to API implementation specifications. (Such specifications are outside of the scope of this  
316 Standard.)

317 Summary of the expected sequence of operations

- 318 a) The RTS instantiates the API implementation in the content DOM and initiates launch of a  
319 content object.
- 320 b) The content object locates the API instance. (Note—This is a required action of the con-  
321 tent.)
- 322 c) The content object invokes the initialize communication session method of the API im-  
323 plementation prior to calling any other method. (Note—This use of this session method is  
324 a required action of the content.)
- 325 d) If the content invokes one or more data-retrieval requests through the API implementation,  
326 the RTS returns the data or an appropriate error message through the API implementation.  
327 Although calls to retrieve data (data-transfer methods) are optional actions of the content  
328 object, the API implementation returns a value or message if a call is made.
- 329 e) If the content object invokes one or more data-storage requests through the API imple-  
330 mentation, the RTS returns an acknowledgement, either "true" or "false" and sets an  
331 appropriate error status, either "0" for no error or an error number. Although calls to store  
332 data (data-transfer methods) are optional actions of the content object, the API implemen-  
333 tation attempts to store valid data and returns an acknowledgement, as well as makes  
334 available an error-status value to be returned on request by the content object.
- 335 f) If the content object invokes one or more of the predefined support methods through the  
336 API implementation, the RTS responds appropriately with data or messages through the  
337 API implementation. Although calls for support methods are optional actions of the con-  
338 tent object, the API implementation returns a value or message if a request is made.
- 339 g) The content object invokes the termination method of the API implementation. (Note—  
340 This use of this session method is a required action of the content object.)
- 341 h) The API implementation rejects any attempt by this instance of the content object to reini-  
342 tialize communication.

### 343 5.3 Implementation for Web-browser-based content

344 As shown in Figure 1, the API implementation uses ECMAScript-compatible methods that can be  
345 called by the content object. From the point of view of the content object, the key attributes of the  
346 ECMAScript API are

- 347 – methods are invoked using ECMAScript-compatible calls;  
348 – all parameters and values are passed as character strings;  
349 – method names are case sensitive;  
350 – communication is invoked by content objects only; and  
351 – integers, reals, and times are be encoded as they would be by the ECMAScript-to-string  
352 cast conversion.

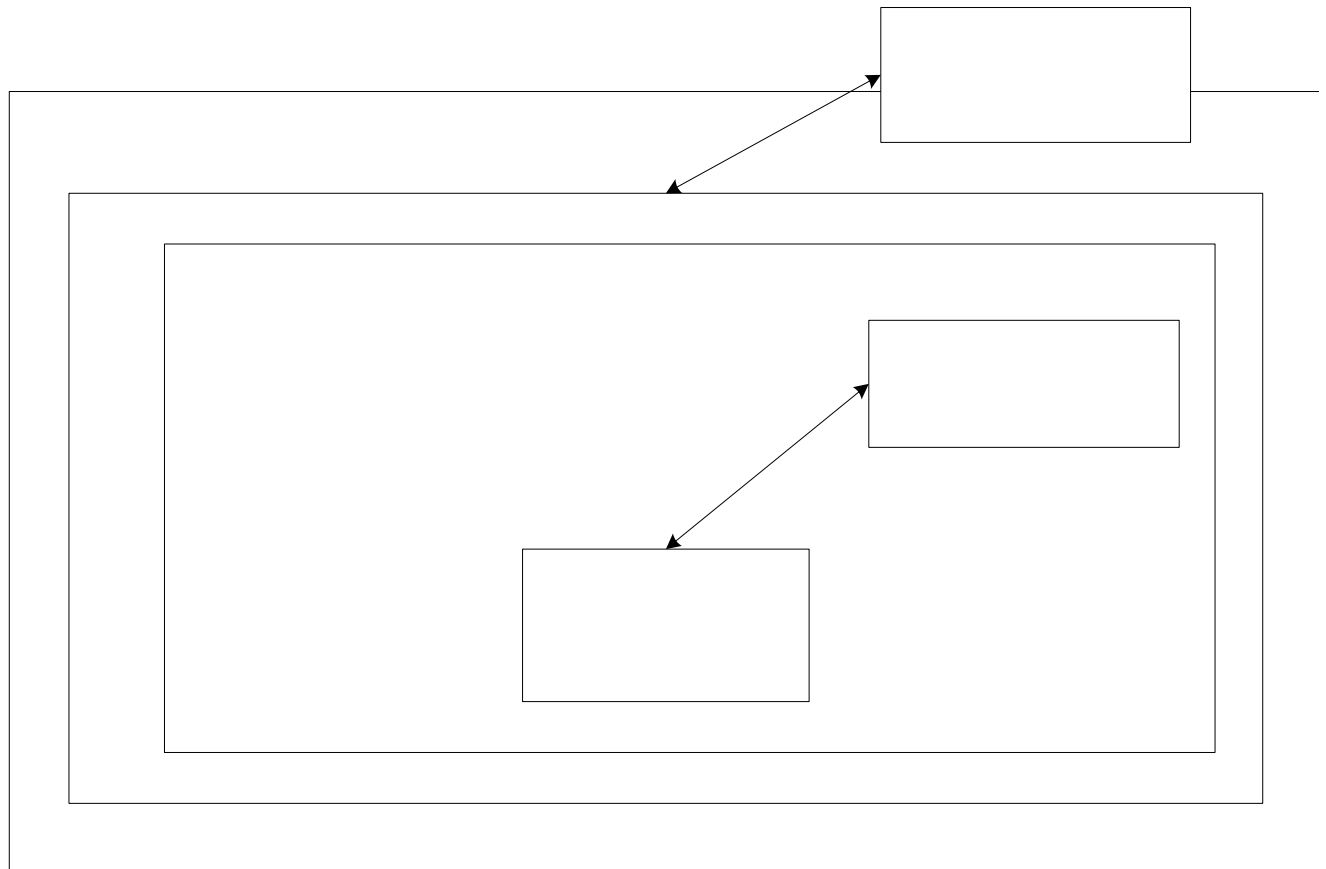
353 The RTS instantiates an API implementation within the content object DOM. The content object  
354 finds the API implementation and establishes communication by calling the initialize communica-  
355 tion session method defined within this Standard. The content object uses the first API implemen-  
356 tation found according to the precedence rules specified in Clause 6.

357 An API implementation need not be implemented in ECMAScript, but the API implementation  
 358 has to support the ECMAScript API.

359 NOTE 1—Implementation of an API is outside of the scope of this Standard. For example, an API imple-  
 360 mentation can be written in any language, as long as the ECMAScript calls are supported. An API imple-  
 361 mentation can shield a content object from particular server-side communication protocols, technology,  
 362 and requirements for communicating with content or backend services. The particular implementation  
 363 needs of an implementer are hidden behind a common, shared set of communication methods. This ap-  
 364 proach shields content object and RTS providers from needing to implement an API that is unique to each  
 365 content or RTS implementer. The ECMAScript calling convention gives wide latitude to implementers.

366 NOTE 2—The API instance is typically placed within the content object DOM by the RTS. The API in-  
 367 stance is either embedded in the content page, or in a frameset or window on a learner's Web  
 368 browser. Finding and establishing communication with the API instance typically occurs immediately after  
 369 the content object is launched.

370 NOTE 3—More than one API instance may exist in a DOM instance as shown in Figure 2. For example,  
 371 this may occur if a content object acts as an RTS for another content object. The precedence rules in  
 372 Clause 6 for locating the API instance ensure that the proper API implementation is found by each content  
 373 object.



374  
 375

376 **Figure 2—Multiple API implementations in the same DOM instance**

377 NOTE 4—Different user communities have specific rules about the kinds of software that is permitted and  
 378 prohibited in Web-browser-based environments. For example, some prohibit the use of plug-ins of any

379 kind for security reasons. Likewise, Java applets, ECMAScript code or other means used to implement or  
380 support the API methods may need to be signed by a trusted authority to properly function within certain  
381 security configurations of a Web browser when API communication spans domains. These restrictions may  
382 limit the applicability of this Standard in these communities and environments.

#### 383 **5.4 ECMAScript API extensibility**

384 Adopters of this Standard are encouraged to fit their communication needs into the general mes-  
385 saging structure identified in Clause 5. If needed, this conceptual communication model can be  
386 extended by introducing new methods, error codes, and error descriptions.

## 387 **6. API instantiation and binding**

388 The instantiation of an API implementation and the detection of the instance by a content object  
389 are discussed in subclauses 6.1 through 6.3.

390 NOTE—In this Clause, "window" is a DOM window object.

### 391 **6.1 Instantiation of an instance of an API implementation**

392 An API implementation shall be instantiated in the DOM environment of the content object be-  
393 fore the content object is launched. This instantiation shall be an object named API\_1484\_11.

394 The API implementation may be instantiated in any of the following DOM elements, in order of  
395 preference:

- 396 a) the chain of parents of the window within which the content object is launched, up to and  
397 including the top window of the Web browser;
- 398 b) the opener window of the window within which the content object is launched, if any; and
- 399 c) the chain of parents of the opener window, if any exist, up to and including the top win-  
400 dow of the Web browser.

### 401 **6.2 Multiple instances**

402 An environment that supports multiple concurrent content objects shall provide a separate in-  
403 stance of an API implementation for each content object. Only a single instance of an API imple-  
404 mentation may be instantiated in any particular DOM element.

### 405 **6.3 Binding a content object to an API instance**

406 To use the API, a content object shall locate an instance of the API. The content object shall look  
407 for an instance of the API implementation in the following locations, in order of precedence, and  
408 stop as soon as an instance is found:

- 409 a) the chain of parents of the current window, if any exist, until the top window of the parent  
410 chain is reached;
- 411 b) the opener window, if any; and
- 412 c) the chain of parents of the opener window, if any exist, until the top window of the parent  
413 chain is reached.

414 NOTE—This order of precedence in searching for an instance of an API implementation ensures that a  
415 content object may act as an RTS for another content object. If the order of precedence were to look in the  
416 top window first, for example, a nested content object would find the wrong instance.

417 (The remainder of this subclause is informative.)

418 A content object may follow a simple algorithm to find an instance of an API implementation.

- 419 – Step 1: follow the algorithm until an instance is found;
- 420 – Step 2: when found, return the instance and exit the "find API instance" routine; or

421       – Step 3: if not found, return a null and exit the routine.

422 A sample ECMAScript implementation of this algorithm tested with several Web browsers is  
423 provided below. This sample is informative and may not be an optimal implementation.

```
424     var nFindAPITries = 0;
425     var API = null;
426     var maxTries = 500;
427
428     function ScanForAPI(win) {
429         while ((win.API_1484_11 == null) && (win.parent != null) &&
430             (win.parent != win)) {
431             nFindAPITries ++;
432             if (nFindAPITries > maxTries) {
433                 alert("Error in finding API instance -- too deeply
434                     nested.");
435                 return null;
436             }
437             win = win.parent;
438         }
439         return win.API_1484_11;
440     }
441
442     function GetAPI() {
443         if ((win.parent != null) && (win.parent != win)) {
444             API = ScanForAPI(window.parent);
445         }
446         if ((API == null) && (window.opener != null)) {
447             API = ScanForAPI(window.opener);
448         }
449     }
```

## 450 **7. Content communication state model**

451 This Clause defines how an API implementation shall behave in response to calls from a content  
452 object. It does not define the behavior of the API-to-RTS communication. The behavior is de-  
453 scribed in terms of the state transitions for each of the events (versus describing them as the tran-  
454 sitions for each event in each state).

455 It is not a requirement that a specific implementation of an API uses a formal state model or that  
456 an API implementation encodes or otherwise represents these specific states or this specific state  
457 model. It is required that an API implementation shall provide the described behavior.

### 458 **7.1 Communication states**

459 Three states exist for each instance of communication between the content object that is launched  
460 and the RTS or its agent, the API implementation. The state model defines the behavior for each  
461 instance.

#### 462 **Communication state**

463 The following states are mutually exclusive:

- 464 – not initialized,
- 465 – running, and
- 466 – terminated.

467 The initial API implementation state (before the content object is launched) shall be "not initial-  
468 ized".

469 NOTE—While it is possible to define the state model only in terms of "not initialized" and "running," add-  
470 ing the "terminated" state provides for future expansion of the model.

#### 471 **Error state**

472 The following states are mutually exclusive:

- 473 – no error (numeric error code value is 0), and
- 474 – error (numeric error code value is > 0).

475 The initial error state (before the content object is launched) shall be "No error" (numeric error  
476 code value "0").

### 477 **7.2 Events**

478 Events are the calls and actions that a content object may explicitly invoke on an API implemen-  
479 tation and the action or actions resulting from a call. These are also the state transitions. There is  
480 one event for launch and one for each method of the API implementation.

### 481 **7.2.1 Initialize communication session event**

482 The initialize communication session event occurs when the content object calls the initialize  
483 communication session method to enable communication with the API implementation.

#### 484 **Initialize communication session event behavior**

- 485 a) Content calls the API instance with the initialize communication session method after  
486 finding the API instance.
- 487 b) When the communication state is "not initialized" and initialization of communication  
488 succeeds, the API instance
- 489 1) sets the communication state to "running";
  - 490 2) sets the error state to "0" (No error); and
  - 491 3) returns "true" to the calling content object.
- 492 c) When the communication state is "not initialized" and initialization of communication  
493 fails, the API instance
- 494 1) makes no change to the communication state's value;
  - 495 2) sets the error state to "102" (General initialization failure); and
  - 496 3) returns "false" to the calling content object.
- 497 d) When the communication state is "running", the API instance
- 498 1) makes no change to the communication state's value;
  - 499 2) sets the error state to "103" (Already initialized); and
  - 500 3) returns "false" to the calling content object.
- 501 e) When the communication state is "terminated", the API instance
- 502 1) makes no change to the communication state's value;
  - 503 2) sets the error state to "104" (Content instance terminated); and
  - 504 3) returns "false" to the calling content object.

### 505 **7.2.2 Terminate communication session event**

506 The terminate communication session event occurs when the content object calls the terminate  
507 communication session method to terminate communication between the content object and the  
508 API implementation (except for error handling).

#### 509 **Terminate communication session event behavior**

- 510 a) Content calls the API instance with the terminate communication session method when  
511 content is ready to cease communications.
- 512 b) When the communication state is "running" and terminating communication succeeds, the  
513 API instance
- 514 1) sets the communication state to "terminated";
  - 515 2) sets the error state to "0" (No error); and
  - 516 3) returns "true" to the calling content object.
- 517 c) When the communication state is "running" and termination of communication fails, the  
518 API instance
- 519 1) makes no change to the communication state's value;
  - 520 2) sets the error state to "111" (General terminate failure); and
  - 521 3) returns "false" to the calling content object.
- 522 d) When the communication state is "not initialized", the API instance
- 523 1) makes no change to the communication state's value;

- 524           2) sets the error state to "112" (Attempt to terminate before initialize); and  
 525           3) returns "false" to the calling content object.  
 526        e) When the communication state is "terminated", the API instance  
 527           1) makes no change to the communication state's value;  
 528           2) sets the error state to "113" (Attempt to terminate after terminated); and  
 529           3) returns "false" to the calling content object.

### 530   **7.2.3 Retrieve data event**

531   The retrieve data event occurs when the content object calls the retrieve data method to obtain a  
 532   value for a data element of a data model via the API implementation.

#### 533   **Retrieve data event behavior**

- 534       a) Content calls the API instance with the retrieve data method when content wants to re-  
 535       trieve data stored in the RTS.  
 536       b) When the communication state is "running" and the API instance succeeds in retrieving  
 537       the requested data from the RTS, the API instance  
 538           1) makes no change to the communication state's value;  
 539           2) sets the error state to "0" (No error); and  
 540           3) returns the requested data to the calling content object.  
 541       c) When the communication state is "running" and the API instance fails to retrieve the re-  
 542       quested data from the RTS, the API instance  
 543           1) makes no change to the communication state's value;  
 544           2) sets the error state to the most appropriate of the following:  
 545            i) "401" (Undefined data model element);  
 546            ii) "402" (Unimplemented data model element);  
 547            iii) "403" (Data model element value not initialized);  
 548            iv) "405" (Data model element is write only); or  
 549            v) "301" (General get failure); and  
 550           3) returns "" (an empty string) to the calling content object.  
 551       d) When the communication state is "not initialized", the API instance  
 552           1) makes no change to the communication state's value;  
 553           2) sets the error state to "122" (Attempt to get before initialize); and  
 554           3) returns "" (an empty string) to the calling content object.  
 555       e) When the communication state is "terminated", the API instance  
 556           1) makes no change to the communication state's value;  
 557           2) sets the error state to "123" (Attempt to get after terminate); and  
 558           3) returns "" (an empty string) to the calling content object.

### 559   **7.2.4 Store data event**

560   The store data event occurs when the content object calls the store data method to set a value for a  
 561   data element in the data model via the API implementation.

562   NOTE—Data may be cached or passed on directly to the RTS by the API instance as the result of a call to  
 563   the store data method.

564 **Store data event behavior**

- 565 a) Content calls the API instance with the store data method when content wants to store data  
566 in the RTS.
- 567 b) When the communication state is "running" and the API instance succeeds in setting the  
568 data, the API instance
- 569 1) makes no change to the communication state's value;
  - 570 2) sets the error state to "0" (No error); and
  - 571 3) returns "true" to the calling content object.
- 572 c) When the communication state is "running" and the API instance fails to set the specified  
573 data to the RTS, the API instance
- 574 1) makes no change to the communication state's value;
  - 575 2) sets the error state to the most appropriate of the following:
    - 576 i) "401" (Undefined data model element);
    - 577 ii) "402" (Unimplemented data model element);
    - 578 iii) "403" (Data model element vale not initialized);
    - 579 iv) "404" (Data model element is read-only);
    - 580 v) "406" (Data model element type mismatch); or
    - 581 vi) "351" (General set failure); and
  - 582 3) returns "false" to the calling content object.
- 583 d) When the communication state is "not initialized", the API instance
- 584 1) makes no change to the communication state's value;
  - 585 2) sets the error state to "132" (Attempt to set before initialize); and
  - 586 3) returns "false" to the calling content object.
- 587 e) When the communication state is "terminated", the API instance
- 588 1) makes no change to the communication state's value;
  - 589 2) sets the error state to "133" (Attempt to set after terminate); and
  - 590 3) returns "false" to the calling content object.

591 **7.2.5 Commit data event**

592 The commit data event occurs when the content object calls the commit data method to request  
593 that the API implementation flush any cached data to the persistent data store of the RTS. If the  
594 API instance does not cache data, the behavior shall be consistent with that of an API instance  
595 that succeeds in transmitting the cached data to the RTS.

596 **Commit data event behavior**

- 597 a) Content calls the API instance with the commit data method when content wants the API  
598 instance to store all cached data in the RTS.
- 599 b) When the communication state is "running" and the API instance succeeds in transmitting  
600 the cached data to the RTS, the API instance
- 601 1) makes no change to the communication state's value;
  - 602 2) sets the error state to "0" (No error); and
  - 603 3) returns "true" to the calling content object.
- 604 c) When the communication state is "running" and the API instance fails to commit the  
605 specified data to the RTS, the API instance
- 606 1) makes no change to the communication state's value;
  - 607 2) sets the error state to "391" (general commit failure); and
  - 608 3) returns "false" to the calling content object.

- 609 d) When the communication state is "not initialized", the API instance  
610 1) makes no change to the communication state's value;  
611 2) sets the error state to "142" (Attempt to commit before initialize); and  
612 3) returns "false" to the calling content object.  
613 e) When the communication state is "terminated", the API instance  
614 1) makes no change to the communication state's value;  
615 2) sets the error state to "143" (Attempt to commit after terminate); and  
616 3) returns "false" to the calling content object.

#### 617 **7.2.6 Get error code event**

618 The get error code event occurs when the content object calls the get error code method to get the  
619 error code from the API implementation.

#### 620 **Get error code event behavior**

- 621 a) Content calls the API instance with the get error code method when content wants to re-  
622 trieve the latest error state from the API instance.  
623 b) Regardless of the communication state ("not initialized", "running", "terminated"), the  
624 API instance  
625 1) makes no change to the communication state's value; and  
626 2) returns the error state to the calling content object (e.g., "301").

#### 627 **7.2.7 Get error string event**

628 The get error string event occurs when the content object calls the get error string method to re-  
629 turn the error text for the error code associated with the method parameter.

#### 630 **Get error string event behavior**

- 631 a) Content calls the API instance with the get error string method when content wants to re-  
632 trieve the textual message associated with an error code from the API instance.  
633 b) Regardless of the communication state ("not initialized", "running", "terminated"), the  
634 API instance  
635 1) makes no change to the communication state's value; and  
636 2) returns the text for an error code when the requested error code is known; or  
637 3) returns "" (an empty string) when the requested error code is unknown.

#### 638 **7.2.8 Get diagnostics event**

639 The get diagnostics event occurs when the content object calls the get diagnostics method to re-  
640 turn extended diagnostic information.

#### 641 **Get diagnostics even behavior**

- 642 a) Content calls the API instance with the get diagnostics method when content wants to re-  
643 trieve the implementation-specific diagnostic information associated with an error state  
644 from the API instance.  
645 b) Regardless of the communication state ("not initialized", "running", "terminated"), the  
646 API instance  
647 1) makes no change to the communication state's value; and

- 648           2) returns the diagnostic information, (if implemented) for an error state when the re-  
649           requested error state code is known; or  
650           3) returns "" (an empty string) when the requested error state code is unknown or the get  
651           diagnostics method is not implemented.

## 652 8. ECMAScript API methods and syntax

653 The ECMAScript API includes three kinds of methods

- 654 – **session methods**—used to mark the beginning and end of communication between a con-
- 655 tent object and an RTS through the API implementation;
- 656 – **data-transfer methods**—used to transfer data model values between a content object and
- 657 an RTS through the API implementation; and
- 658 – **support methods**—used for auxiliary communications (e.g., error handling) between a
- 659 content object and the API implementation or RTS.

660 The API implementation shall conform to ECMAScript calling conventions as specified by  
661 ISO/IEC 16262:1998. The parameters and return values shall be of type character string. The syn-  
662 tax and conventions used for methods are shown below.

```
663     method_label ( [parameter [ , parameter ] ] )
```

664

- 665 – a method's label shall be case sensitive; and
- 666 – square brackets are used to indicate additional parameters, as described in each method's
- 667 specifications.

668 Examples:

- 669 – `method_a ()` // requires no parameters
- 670 – `method_b ( param1 )` // requires exactly one parameter
- 671 – `method_c ( param1, param2 )` // requires exactly two parameters

672 NOTE 1—White space is used in this Clause to improve readability and is not required by ECMAScript.

673 NOTE 2—This Standard does not specify the data bindings of parameters.

### 674 8.1 Session methods

675 Sessions methods are used to initiate and terminate data communication between an API imple-  
676 mentation and a single instance of a content object during a single communication session.

#### 677 8.1.1 Initialize communication session method

##### 678 Interface syntax

```
679     return_status = Initialize(parameter)
```

##### 680 Description

681 This method is used to initiate communication between a content object and an instance of the  
682 API implementation. Until the communication state has changed to "running" and a call to this  
683 method has returned "true", any call to another API method results in behavior as defined in  
684 subclause 7.2.

685 NOTE 1—Conformance for content objects requires that `Initialize()` be the first method called.

686 NOTE 2—Subsequent calls to `Initialize()` after the initial call has returned "true" do not reinitialize  
687 the communication session (see subclause 7.2.1).

### 688 **Control inputs**

689 Called as an ECMAScript API function. Enables communication.

### 690 **Control outputs**

691 Always returns from function call. The API implementation shall take whatever actions needed to  
692 initialize communication with this instance of the content object. If communication for this in-  
693 stance of the content object was already enabled, no attempt to re-enable communication shall be  
694 made and an error shall be returned (see subclause 7.2.1).

### 695 **Data inputs**

696 - `parameter`: An empty character string ("").

697 The API implementation shall accept a call to this method as enabling communication.

698 NOTE—The default `parameter` is an empty character string (""). Future editions of this Standard may  
699 specify the meaning of non-empty character strings.

### 700 **Data outputs**

701 The character string value "true" shall be returned if initialization was successful. Otherwise,  
702 the character string value "false" shall be returned. If this method returns "false", the API  
703 implementation shall set the error code to a value specific to that error, and the content object may  
704 call `GetLastError()` to determine the nature of the error (see subclause 8.3.1).

### 705 **Example**

```
706 //Initialize communication  
707 status = Initialize("")
```

## 708 **8.1.2 Terminate communication session method**

### 709 **Interface syntax**

```
710 return_status = Terminate(parameter)
```

### 711 **Description**

712 This method is used to terminate communication between a content object and an instance of the  
713 API implementation. If this method returns "false" the API behavior shall be as defined in sub-  
714 clause 7.2.

715 NOTE—If `Terminate()` returns "false", the content object can call `GetLastError()` to determine  
716 whether the failure was because the session had already been terminated (see subclause 8.3.1).

### 717 **Control inputs**

718 Called as an ECMAScript API function. Ends communication.

**719 Control outputs**

720 Always returns from function call. The API implementation shall ensure all valid data has been or  
721 is written to the appropriate persistent store. This implies an implicit `Commit("")` (see subclause  
722 8.2.3).

**723 Data inputs**

724 - `parameter`: An empty character string (`""`).

725 The API implementation shall accept a call to this method as ending communication.

726 NOTE—The default value is an empty character string (`""`). Future editions of this Standard may specify  
727 the meaning of non-empty character strings.

**728 Data outputs**

729 The character string value `"true"` shall be returned if termination was successful. Otherwise,  
730 `"false"` shall be returned. When the method returns `"false"`, the API implementation shall set  
731 the error code to a value specific to that error, and the content object may call `GetLastError()`  
732 to determine the nature of the error (see subclause 8.3.1).

**733 Examples**

```
734 //Terminate communication  
735 status = Terminate("")
```

**736 8.2 Data-transfer methods**

737 Data-transfer methods are used to direct the storage and retrieval of data that is to be available  
738 within the current communication session.

739 NOTE—Typically, data which is available in the current communication session will be stored for future  
740 use.

**741 8.2.1 Retrieve data method****742 Interface syntax**

```
743 return_value = GetValue(parameter)
```

**744 Description**

745 This method requests the information associated with `parameter`.

**746 Control inputs**

747 Called as an ECMAScript API function.

**748 Control outputs**

749 Always returns from function call.

**750 Data inputs**

751 - `parameter`: The complete identification of an element within a data model, including the  
752 identification of the data model (e.g., `data_model.element`). If an error occurs, the content  
753 object should not rely on the empty string as being a reliable value.

**754 Data outputs**

755 A character string containing the value associated with `parameter` shall be returned. The  
756 maximum length shall be determined by the data element as specified in the binding that corre-  
757 sponds to the parameter. If an error occurs, then the API implementation shall set the error code to  
758 a value specific to that error and return an empty string. The content object may call `GetLastEr-`  
759 `ror()` to determine the nature of the error (see subclause 8.3.1).

**760 Example**

```
761 //Get the student ID  
762 student_id = GetValue("cmi.learner_id");
```

**763 8.2.2 Store data method****764 Interface syntax**

```
765 return_status = SetValue(parameter_1, parameter_2)
```

**766 Description**

767 This method is used to transfer to the RTS the value of `parameter_2` for the data element speci-  
768 fied by `parameter_1`.

**769 Control inputs**

770 Called as an ECMAScript API function.

**771 Control outputs**

772 Always returns from function call.

**773 Data inputs**

774 - `parameter_1`: The complete identification of a data element within a data model, includ-  
775 ing the identification of the data model (e.g., `data_model.element`).  
776 - `parameter_2`: The value to which the content of `parameter_1` is to be set.

**777 Data outputs**

778 The character string value "true" shall be returned if the RTS accepts the content of  
779 `parameter_2` to set the value of `parameter_1`. Otherwise, "false" shall be returned. When

780 the method returns "false", the API implementation shall set the error code to a value specific  
781 to that error, and the content object may call `GetLastError()` to determine the nature of the  
782 error (see subclause 8.3.1).

### 783 **Example**

```
784     //Set the lesson status  
785     status = SetValue("cmi.lesson_status", "passed");
```

### 786 **8.2.3 Commit data method**

#### 787 **Interface syntax**

```
788     return_status = Commit(parameter)
```

#### 789 **Description**

790 This method requests forwarding to the persistent, long-term, data store any data from the content  
791 object that may have been cached by the API implementation since the last call to  
792 `Initialize()` or `Commit()`, whichever occurred most recently.

793 If the API implementation does not cache values, `Commit()` shall return "true", set the error  
794 code to "0" ("No error"), and do no other processing.

795 Cached data shall not be modified as a result of a call to the commit method. For example, if the  
796 content object sets the value of a data element, then calls the commit method, and then subse-  
797 quently gets the value of the same data element, the value returned shall be the value set in the  
798 call prior to invoking the commit method.

799 NOTE 1—`Commit()` would typically be used defensively to reduce the likelihood that persistent data is  
800 lost because a communication session is interrupted, ends abnormally, or otherwise terminates prematurely  
801 without a call to `Terminate()` (see subclause 8.1.2).

802 NOTE 2—At this time there is no explicit `parameter` for this method.

803 NOTE 3—At this time there is no companion method to "rollback" or discard any data that may be kept in  
804 internal, buffered, temporary, or other transient storage.

#### 805 **Control inputs**

806 Called as an ECMAScript API function. Always returns from function call.

#### 807 **Control outputs**

808 If the API implementation is caching data, data shall be forwarded from the temporary store to the  
809 persistent store.

#### 810 **Data inputs**

811 – `parameter`: An empty character string ("").

812 The API implementation shall accept a call to this method as a directive to store data.

813 NOTE—The default value is an empty character string (""). Future editions of this standard may specify  
814 the meaning of non-empty character strings.

### 815 **Data outputs**

816 The character string value "true" shall be returned if the data was successfully committed. Other-  
817 wise, "false" shall be returned. If the method returns "false", then the API implementation  
818 shall set the error code to a value specific to that error, and the content object may call  
819 `GetLastError()` to determine the nature of the error (see subclause 8.3.1). If the API imple-  
820 mentation is not caching values, the character string value "true" shall be returned.

### 821 **Examples**

```
822     //Flush the cache  
823     status = Commit("");
```

## 824 **8.3 Support methods**

825 Support methods are used for error handling and diagnostics.

### 826 **8.3.1 Get error code**

#### 827 **Interface syntax**

```
828     error_code = GetLastError()
```

#### 829 **Description**

830 This method requests the error code for the current error state of the API implementation. Calling  
831 this method shall not result in a change to the current error state.

832 `GetLastError` may be called after `Terminate()`.

833 NOTE—For any method called other than `GetLastError()`, `GetErrorString()`, and  
834 `GetDiagnostic()`, the API implementation sets the error state as defined in subclause 7.2.

#### 835 **Control inputs**

836 Called as an ECMAScript API function.

#### 837 **Control outputs**

838 Always returns from function call.

#### 839 **Data inputs**

840 None.

**841 Data outputs**

842 A character string (convertible to an integer in the range from 0 to 65535 inclusive) representing  
843 the number of the last error shall be returned. If an error occurs during the processing of session  
844 or data-transfer methods, then the API implementation shall set the error code to a value specific  
845 to that error, and the content object may call `GetLastError()` to determine the nature of the  
846 error.

**847 Examples**

```
848 //Get the last error code  
849 code = GetLastError();
```

850 NOTE— Repeated calls to `GetLastError()` with no other interceding calls to any other methods will  
851 return the same error code.

**852 8.3.2 Get error string****853 Interface syntax**

```
854 error_text = GetErrorString(parameter)
```

**855 Description**

856 This method requests the textual message for the error code specified by the value of `parameter`.  
857 A conforming API implementation shall support the error codes identified in Table 2 in subclause  
858 8.4. Calling this method shall not result in a change to the current error state.

859 NOTE—For any method called other than `GetLastError()`, `GetErrorString()`, and  
860 `GetDiagnostic()`, the API implementation sets the error state as defined in subclause 7.2.

**861 Control inputs**

862 Called as an ECMAScript API function.

**863 Control outputs**

864 Always returns from function call.

**865 Data inputs**

866 – `parameter`: The character string representation of the integer value corresponding to an  
867 error message.

**868 Data outputs**

869 A character string containing the textual description corresponding to the error code specified by  
870 the value of `parameter` shall be returned. The maximum length shall be 256 bytes (including  
871 null terminator). If the requested error code is unknown, an empty string ("") is returned.

**872 Examples**

```
873 //Get text for error code  
874 error_text = GetErrorString("201")
```

**875 8.3.3 Get API-implementation-specific diagnostics****876 Interface syntax**

```
877 diagnostic_text = GetDiagnostic(parameter)
```

**878 Description**

879 This method allows an RTS implementer to provide detailed diagnostics through the API imple-  
880 mentation. Calling this method shall not result in a change to the current error state.

881 NOTE 1—This method exists for implementation-specific use. It is useful to those who supply or know the  
882 parameter values that may return extended diagnostic information for an instance of an API implementa-  
883 tion.

884 NOTE 2—A legitimate use for `GetDiagnostic()` is to get additional machine-interpretable information  
885 about an error code. For example, `GetDiagnostic("101")` might be defined in an application profile as  
886 returning detailed error information that includes state information.

887 NOTE 3—For any method called other than `GetLastError()`, `GetErrorString()`, and  
888 `GetDiagnostic()`, the API implementation sets the error state as defined in subclause 7.2.

**889 Control inputs**

890 Called as an ECMAScript API function.

**891 Control outputs**

892 Always returns from function call.

**893 Data inputs**

894 - `parameter`: An implementer-specific value for diagnostics. The maximum length of the  
895 `parameter` value shall be 256 bytes (including null terminator).

896 NOTE—The value of the `parameter` need not be an error code.

**897 Data outputs**

898 A character string shall be returned. The maximum length shall be 256 bytes (including null ter-  
899 minator).

**900 Examples**

```
901 //Get implementer diagnostics  
902 return = GetDiagnostic("some_implementer_parameter")
```

## 903 8.4 API implementation error codes

904 Error codes shall be integers converted to character strings. The integer values of error codes shall  
 905 be in the range 0 to 65535 inclusive. Unspecified error codes in the range 0 to 999 inclusive are  
 906 reserved for use in future editions of this Standard. Implementation-defined error codes shall be  
 907 in the range 1000 to 65535.

908 If multiple errors are detected in processing a call to an API implementation, one error code shall  
 909 be returned by a call to GetLastError() (see subclause 8.3.1). The precedence among error codes  
 910 is unspecified behavior.

911 NOTE—Calls to support methods leave the error state unchanged. Successful calls to methods other than  
 912 support methods reset the error state to 0.

913 Table 1 lists the ranges for classes of error codes.

914

**Table 1—Classes of error codes**

<b>Numeric range</b>	<b>Error type</b>
100 to 199	General errors
200 to 299	Syntax errors
300 to 399	RTS errors
400 to 499	Data model errors

915

916 Table 2 lists specific error codes and messages. The description column in Table 2 is informative.  
 917 The actual character string for the textual message returned for the error code is left to the imple-  
 918 mentation. Additional codes may be specified with data bindings.

919

**Table 2—Specific error codes and messages**

<b>Code</b>	<b>Name</b>	<b>Description</b>
0	No error	No errors encountered. Successful API call.
101	General exception	A general exception for which a more specific error code is not available.
102	General initialization failure	An attempt to initialize failed. No other error information is available.
103	Already initialized	An attempt was made to reinitialize communication that had already been initialized.

<b>Code</b>	<b>Name</b>	<b>Description</b>
104	Content instance terminated	An attempt was made to reinitialize communication by an instance of a content object with a state of "terminated".
111	General termination failure	An attempt to terminate the communication session failed. No other error information is available.
112	Attempt to terminate before initialize	An attempt was made to terminate communication prior to initialization.
113	Attempt to terminate after terminated	An attempt was made to terminate communication that had already been terminated.
122	Attempt to get before initialize	An attempt was made to get a value for a data element before communication had been initialized.
123	Attempt to get after terminate	An attempt was made to get a value for a data element after communication had been terminated.
132	Attempt to set before initialize	An attempt was made to set a value for a data element before communication had been initialized.
133	Attempt to set after terminate	An attempt was made to set a value for a data element after communication had been terminated.
142	Attempt to commit before initialize	An attempt was made to commit data to storage before communication had been initialized.
143	Attempt to commit after terminate	An attempt was made to commit data to storage after communication had been terminated.
201	General argument error	An attempt was made to pass an invalid argument. (An error in the range 401 through 499 should be used if applicable.)
301	General get failure	An attempt to get the value for a data element failed. No other error information is available.
351	General set failure	An attempt to set the value for a data element failed. No other error information is available.

Code	Name	Description
391	General commit failure	An attempt to commit data to storage failed. No other information error is available.
401	Undefined data model element	<p>The data model element passed as <code>parameter</code> in <code>GetValue()</code> or <code>parameter_1</code> in <code>SetValue()</code> is undefined.</p> <p>Note: An attempt was made to use a data model element that is not recognized by the implementation.</p>
402	Unimplemented data model element	<p>The data model element passed as <code>parameter</code> in <code>GetValue()</code> or <code>parameter_1</code> in <code>SetValue()</code> is not supported by the implementation.</p> <p>Note: An attempt was made to use a data model element that is recognized by the implementation but is not supported.</p>
403	Data model element value not initialized	The data model element passed as <code>parameter</code> in <code>GetValue()</code> has not been initialized with a value by an implementation. The value returned by the <code>GetValue()</code> is unreliable.
404	Data model element is read only	<p>The data model element passed as <code>parameter_1</code> in <code>SetValue()</code> is implemented as a read-only data model element.</p> <p>Note: This Standard does not specify accessibility rights of data model elements.</p>
405	Data model element is write only	<p>The data model element passed as <code>parameter</code> in <code>GetValue()</code> is implemented as a write-only data model element.</p> <p>Note: This Standard does not specify the accessibility rights of data model elements.</p>
406	Data model element type mismatch	<p>The value passed as <code>parameter_2</code> in <code>SetValue()</code> does not evaluate to a valid type for the data model element indicated in <code>parameter_1</code> of a <code>SetValue()</code>.</p> <p>Note: This Standard does not specify types specific to a data model.</p>

921 **Annex A**

922 (informative)

923 **Bibliography**

924 [A1] AICC CMI001, AICC/CMI Guidelines For Interoperability, Version 3.4, October 2000, Ap-  
925 pendix B – API-Based CMI Communication. (See: [http://www.aicc.org/pages/down-docs-  
index.htm](http://www.aicc.org/pages/down-docs-<br/>926 index.htm))

927 [A2] IEEE 100, The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition

928 [A3] W3C, Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C  
929 Working Draft 14 January 2002. (See: <http://www.w3.org/DOM/>)

## 930 **Annex B**

931 (informative)

### 932 **When to call the terminate communication session method**

933 To ensure that the terminate communication session method will be called even if a content object  
934 is unloaded, a properly initialized content object should be coded to call `Terminate()` at the fol-  
935 lowing times and under the following circumstances:

- 936 – When the content object no longer needs to communicate with the RTS. Example: When a  
937 learner clicks a "submit" button at the end of a test and the learner will not be allowed to  
938 change responses, the content object may upload the results of the test with  
939 `SetValue`, then call `Terminate()`.
- 940 – If `Terminate()` has not been called successfully yet, when an `onbeforeunload` browser  
941 event is processed. (This will only happen if the Web browser is Internet Explorer.)
- 942 – If `Terminate()` has not been called successfully yet, when an `onunload` browser event is  
943 processed.

944 NOTE—A call to `Terminate` is successful if the call returns "true". The call may return "false" if the  
945 RTS determines that an error or time out has happened that prevents the RTS from terminating the com-  
946 munication session normally. How to handle this situation is not defined, but it may be good practice to set  
947 a timer to call `Terminate()` again after a few seconds.

948