DRAFT 0.8

Cooking up a SCORM

A SCORM 1.2 Content Cookbook for Developers

Version 1.2 – Draft 0.8

Claude Ostyn
Learning Standards Strategist
Click2learn, Inc.

click2learn™

Copyright and Terms of Use

Click2learn Copyrights and Permissions
110 - 110th Ave NE, Suite 700
Bellevue WA 98004
USA

Point of contact for questions and comments:
Claude Ostyn
Claude.Ostyn@Click2learn.com

Contents

Introduction

Overview

The Shareable Content Object Reference Model (SCORM), published by the Advanced Distributed Learning (ADL) project, is a de facto standard for e-learning content. This document describes how to create Web-based content that complies with the SCORM 1.2 specification. Like the SCORM specification, this is a technical document, not an instructional design document. Like SCORM, it does not deal with the instructional quality of the content, but with the features that are necessary to make the content portable and conforming to the SCORM. The document uses functioning examples to illustrate various aspects of the SCORM specification.

This document is divided into several chapters. Following a brief overview of SCORM, each chapter builds on the basic SCO model to build progressively more capable and complex SCORM-compliant learning objects. Finally, a chapter reviews the basics of packaging--how to assemble the learning objects into a SCORM-conformant package that can be migrated and deployed on any compliant delivery system.

Audience and purpose

If you are already using a fully SCORM 1.2 compliant authoring tool or development environment like ToolBook or Aspen 2.1 LCMS, you probably do not need this document although its introduction may be useful to understand the underlying technology and rules.

Depending on your purpose and interest, you may read this document in different ways:

- As a companion to the SCORM specification, to get a better idea of what is and is not covered by that specification and see examples of how it can be applied in content.
- To understand various technical aspects of the SCORM specification through examples.
- As a technical guide with a set of examples that may help you save time in the actual implementation of content.

This document is intended for content developers with different levels of technical expertise. It may also be useful to instructional designers, quality assurance staff and program managers who want to understand better what SCORM 1.2 enables and makes difficult. However, this document is aimed primarily at a technical audience with at least some familiarity with Web concepts and hands-on basic HTLM and JavaScript experience.

What about SCORM 1.3?

Content created according to the guidelines in this document will not automatically be rendered obsolescent by SCORM 1.3 or other future specifications because:

- SCORM 1.2 is much simpler than SCORM 1.3 and conformance is easier to verify. Since there are entire classes of content that do not require the new features introduced in SCORM 1.3, it is expected that support for SCORM 1.2 for general content delivery, migration and archival will continue for a long time
- Robust implementation of SCORM 1.3 conformance in commercial products may take anywhere from 3 to 24 months, if the history of SCORM 1.2 is any guide
- Systems that support SCORM 1.3 content are expected to also support SCORM 1.2 content, either directly or through the use of a generic adapter
- This document describes a layered approach that allows wholesale upgrades of SCORM content object from 1.2 to 1.3 compliance just by upgrading one or two generic script files

An update of this document is planned for release after the final SCORM 1.3 specification is released in late 2003. The plan includes a section that will explain how to update content created according to this document to be SCORM 1.3 compliant.

About authoring tools

The authoring tool used to develop the examples in this document is a simple text editor. However, these examples are not populated with the rich content and interactions that a real authoring tool can facilitate. A tool such as Click2learn's Aspen 2.1 Web-based enterprise LCMS, or desktop authoring tools such as Click2learn's ToolBook 8.6, does much of what is described in this document automatically: You just specify that you want to export the content as a SCORM package and it will do it for you. Those tools are designed specifically to isolate you from the technical minutiae described in the SCORM specification and in this document, so that you can focus on what your content shows, tells and teaches.

For example, Aspen Learning Content Management Server (LCMS) is a global, scalable, 100% Web-based application that enables rapid creation, delivery and management of high quality learning content for your entire enterprise. Aspen LCMS is designed to facilitate the separation of presentation, logic and behavior in learning objects, providing maximum flexibility and reusability. The team-based authoring environment allows concurrent development, allowing instructional designers, subject matter experts, project managers, and reviewers to work together to create high quality courses quickly. In this big picture, SCORM conformance plays a small part, although it is a critical one.

Note that if when you use ToolBook, Aspen LCMS or another SCORM-conformant authoring tool or content generator to create the content, the actual code and Web document structure those tools generate will be quite different from what is described in this document. There are many ways to achieve SCORM conformance, which is good because it leaves a lot of room for creativity.

About the examples

The examples in this document illustrate how content that conforms to the SCORM specification *can* be implemented. You may use any of the code included in the samples described in this document but, if you do, proper credit should be given to Click2learn, Inc., including a reference to this document. If you choose to reuse any of the sample code, it is entirely at your own risk. The sample listings and the code in the companion files in working examples are not intended to be actual production code and do not represent that Click2learn is actually implementing such code in its products. Some code and features that could be useful in an actual product were deliberately not included because they are considered proprietary, or trade secrets of Click2learn, Inc. In particular, error handling is minimal in order to keep the sample listings brief. The exhaustive testing that would normally be done for actual production code, using all the browsers in common use, has not been performed on most of these samples. The examples have been tested with Microsoft Internet Explorer 6.0 and with the SCORM 1.2.3 test suite, and should work with other browsers and versions, but "your mileage may vary." It is highly recommended that you keep the SCORM 1.2 specification handy for reference as you work your way through these examples.

Your own implementation, coding style and policies may be completely different from what is shown here. However, that should not prevent them from conforming to SCORM and interoperating predictably with other SCORM conformant systems. That is the beauty of a standard. Interoperability is what matters, not how you implemented it.

The actual example files, as well as the most recent version of this document, may be downloaded from http://home.Click2learn.com/standardswork.

Chapter 1 - Overview of SCORM

What is SCORM?

SCORM is a set of specifications that describes:

- How to create Web-based learning content that can be delivered and tracked by different SCORM-compliant learning management systems
- What a SCORM-compliant learning management system must do in order to properly deliver and track SCORM-compliant learning content

The SCORM specifications are based on various other industry standards and specifications. The current official version is 1.2.

The SCORM specification does not cover all aspects of a learning enterprise; for example, it does not specify how tracking information is stored and what reports are generated, what pedagogical or training models should be used, or how learner information is compiled.

SCORM 1.2 also does not specify how content is sequenced by a runtime service. The most common assumption is that the user is free to choose any part of the content. Future SCORM specifications will define how to specify sequencing behavior for content. In the meantime, SCORM 1.2 provides a robust specification for content that can be packaged and migrated between systems, installed on an LMS or archived in a plug-and-play manner.

There is a version 1.3 in the works, which will extend on SCORM 1.2 by specifying how to add prescriptions for sequencing. That new version will probably not become final until sometime in 2003 and content developed to conform to SCORM 1.2 should still be able to function in an implementation of SCORM 1.3.

What is SCORM-compliant learning content?

In the SCORM 1.2 specification, SCORM-compliant learning content is either—in SCORM terminology—a *Content Aggregation Package* or a *Resource Package*. A Resource Package is a collection of learning assets that is not intended for delivery as such, for example to archive or migrate a collection of asset. This document does not describe how to build or use Resource Packages; it focuses on deliverable content. A Content Aggregation Package is:

- Intended for delivery to a learner through a Web browser
- Described by metadata
- Organized as a structured collection of one or more learning objects called *Shareable Content Objects* (typically abbreviated as "*SCO*"[1])
- Packaged in such a way that it can be imported by a compliant learning management system or into a repository used by such a system

SCORM content is made of SCOs aggregated into a content package. The SCOs are a specialized type of learning object. Each SCO is a unit of content that can be delivered to a learner by a SCORM-compliant learning management system in order to create a useful learning experience.

The SCOs used in a SCORM-compliant package may be fully included in the package, or used by reference. For example, under certain conditions a learning sequence may include learning objects that reside on another server. Note that the security restrictions implemented in Web browsers to prevent malicious cross-server exploits make the use of learning objects that reside on another server more difficult. How to solve this problem is an issue for learning content management system and content repository vendors to resolve. There is nothing you can do in the content itself to work around this security barrier.

[1] "SCO" is commonly pronounced "scō", rather than sounding out the letters as in "ĕs'sē'ō"

Kinds of SCORM learning objects

A Web-based learning object that can be included in a package for delivery by a SCORM compliant learning management system as an individual activity is called a "*Shareable Content Object*" (SCO). In practice, SCOs come in two main flavors:

- A minimal SCO. This is HTML content or a service that can be launched in the browser window, and uses the SCORM Application Program Interface (API) for minimal communication with the learning management system. An LMS can track the time between the launch and the normal termination of such an object. Most generic content that can be launched in a browser window, but does not contain links to another learning object, can usually be turned into a SCO by "wrapping" it into a SCO. This could be an HTML page, an Adobe Acrobat file, or a text file. A collection of HTML pages is also acceptable, as long as they only link to each other and not to other learning objects.
- A data-enabled SCO. This is like a basic SCO, but also uses the SCORM API to get or send data to the LMS. The data may include tracking data, learner information, etc. as defined in the SCORM 1.2 specification.

The technical requirements for SCOs will be described in more detail in the technical sections of this document and illustrated with examples.

Online or offline learning objects

Regardless of how many SCORM features they use, SCORM 1.2 compatible learning objects do not communicate across the Web to a remote server. They only communicate with other objects within the same browser environment on the client side. The implementer of the delivery environment provides that object. There is a well-defined way for the content to find that object and communicate with it using simple JavaScript code or its equivalent. This has major advantages:

- SCOs can be very easy to implement, because they do not need to include the complex communication protocol required to send and receive data across the Web.
- SCOs can be run in an offline environment without requiring a local Web server or proxy server, because they communicate with other local browser objects rather than with a server-based object.

The SCORM API implementation is instantiated on the client side by a runtime service before the SCO is launched. This implementation may vary from vendor to vendor. For example, the API may be implemented in an HTML frameset that contains a "stage" frame within which the learning objects are launched.

Organizing and sequencing learning objects

The person or entity that creates a package of learning objects decides how the learning objects are organized. However, since SCORM 1.2 does not define any sequencing information, the learner will be able to choose which learning object to use and in which order.

Future versions of SCORM, starting with SCORM 1.3 will add a more advanced sequencing model to SCORM 1.2, to allow the implementation of richer pedagogical or instructional models.

SCORM 1.2 uses the IMS Packaging specification as a foundation for the packaging and organization of learning objects. A package may contain more than one organization of the same learning objects. For example, you could define two or more tracks covering the same subject at different levels of depth, or for different audiences. An LMS can take advantage of this to allow a choice of the more appropriate organization.

SCORM 1.2 specifies how to build a package, but it does not specify how an LMS uses some optional features of the package such as multiple content organizations. Once the learner, manager or learning managements system has chosen an organization within the package, however, SCORM 1.2 does specify that the learner must be able to launch each of the learning objects (SCOs) defined in the organization.

Organizing learning objects and SCOs

An organization of items

The organization of the SCORM learning objects in a package is described in a hierarchical tree structure, such as a course structure or hierarchy of content. SCORM does not specify a particular depth of the tree. Also, SCORM does not specify any particular nomenclature to name the levels of the tree, such as "course, lesson, topic" or "unit, module, lesson." You are free to use whatever nomenclature you like, or none. The length of the branches of the trees may vary.

Each item in the tree can point to a learning object, or it can contain other items. Each item must have a title, which a run time environment will display to the learner.

An item in the tree can have children and also point to a learning object. For example, if your content hierarchy represents sections, chapters and pages, the chapter headings may have their own "cover page." However, in SCORM 1.3, only learning objects associated with leaf nodes in the tree will be launched and tracked as SCOs.

You can mix and match SCOs at all levels of technical compatibility within the same organization. For example, you can aggregate simple, one-page SCOs created with Notepad with complex SCOs created with an advanced authoring tool such as ToolBook.

Similarity with other organization models

Older organization models, like the AICC block/AU structure, can be mapped directly into this organization model.

Content organization models that use directed graphs cannot be represented directly in the SCORM hierarchical tree structure. However, many directed graphs can be represented by making multiple items point to another organization in the package, instead of referencing a SCO. See the description of the packaging manifest (below) and the IMS Content Packaging specifications for more information on how to achieve this with sub-manifests.

Sequencing

SCORM 1.2 does not define how to sequence SCOs. User-choice sequencing is assumed.

User-choice sequencing

In user-choice sequencing, the runtime service allows the user to choose any item in the entire learning objects organization. Depending on the implementation, this could be accomplished through a visual tree, a menu or a set of nested menus. SCORM 1.2 does not specify what the user interface to choose an item will look like.

For example, a typical implementation with a table of content may work like this: When the learner selects an item in the table of content, the item is highlighted and the corresponding learning object is launched in the stage window. If the item has children, but no learning object of its own, the first child that has a learning object is highlighted and that learning object is launched.

Special rules for some items

Mastery score

You can assign a mastery score to an activity that uses a SCO. If a score is reported for the SCO, the runtime service compares the score with the mastery score to set the status of the activity to "passed" or "failed." This overrides any status that may have been reported by the SCO.

Timeout and timeout action

See details in the SCORM specification. The runtime service can unload a SCO when the time allowed has expired; however this behavior is not well defined and may not be available on all

implementations because it requires the run time environment to be more complex on the client side.

Prerequisites

SCORM 1.2 defines a very basic form of prerequisites. A prerequisite references another element in an organization tree that must be completed or mastered. However, because the SCORM specification does not clearly define the behavior associated with prerequisites, only a way to specify them, use prerequisites at your own risk. SCORM 1.3 will include more robust sequencing rules that will not rely on hard-wired references between items, but that will instead allow adaptive sequencing decisions based on the status of learning objectives.

Implementing SCORM-compliant learning objects and SCOs

The runtime service

In order to implement SCORM-compliant learning objects and SCOs it is useful to understand a little bit of the runtime environment in which they will be used. To distinguish this runtime service from the rest of the LMS and its login, authentication, storage and reporting services, we will refer to the runtime environment and the processes that manage it as the "runtime service."

The distinction between LMS and runtime service can be useful because an LMS and the user experience may have, for example, different latency requirements.

The runtime service can be implemented in part on the client side (i.e., on the learner's computer) and in part on the server side (i.e. somewhere on a server). In an offline situation, the server side of the environment is emulated by an application that runs on the learner's computer.

The client side of the runtime service:

- Is provided by the LMS;
- Is implemented as a Web page or frameset in a browser window;
- Presents some necessary runtime user interface components to the learner, such as a table of content and/or navigation buttons;
- Launches learning objects in a "stage window" that is either part of the frameset, or a separate window that it creates as needed;
- Includes an API object instance named "API" that can be found and called by JavaScript or ECMAScript;
- Is generated or regenerated as needed by the server side.

The server side of the runtime service and the rest of the learning management system are completely invisible to the scripts in SCORM learning objects. The user interface components of the runtime service, such as a table of content or navigation buttons, are also completely invisible to SCORM learning objects. How the client side of the runtime service is implemented, and how it communicates with the server are totally invisible to the content. The only parts of the runtime service with which a SCORM learning object can detect and communicate are:

- The stage window
- The API adapter
- The functions provided by the API adapter

The learning objects are specifically prohibited from navigating to other learning objects within the stage window. Only the runtime service can load another learning object in the stage window. The runtime service uses the content organization defined in the content package as its guide to manage navigation between learning objects.

Chapter 2 - Basic SCO behavior

Simplest SCO

Basically, a SCO is some Web-based content that communicates with the runtime service after it has been launched and again when it finishes.

In its simplest form, such a SCO would be a Web page with a short script. It can also consist of multiple HTML pages. The samples in the Appendix section of this document show how simple a single-page or multi-page SCO can be, especially when using "canned" generic scripts through inclusion.

When the SCO is loaded, the script finds the API adapter, and calls the function `LMSInitialize` of the adapter.

When the SCO is unloaded, or at anytime before that if the SCO determines that it is finished, the script calls the function `LMSFinish` of the API adapter.

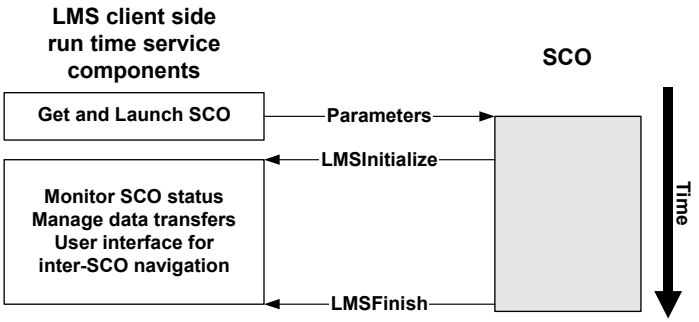


Figure 2.1 – A simple SCO

Behaviors and responsibilities of a simple SCO

- When launched, find the API adapter by searching in a specific way.
- Having found the API adapter, call `LMSInitialize` to begin a communication session.
- When finished, call `LMSFinish` to terminate the communication session.

Data enabled SCO

A data-enabled SCO calls `LMSInitialize` and `LMSFinish` like a simple SCO, but it also exchanges data with the runtime service in between those two calls. SCORM 1.2 uses a data model called the CMI data model, based on an earlier specification developed by the Aviation Industry Computer-Based Training Committee (AICC). A data model is a well-defined catalog of data elements. Both the runtime service and the SCO understand the CMI data model. For example, if the SCO is a quiz, the score obtained by the learner is a value that can be stored in a data model element. The SCO can then send this data element to the runtime service, and the runtime service will know where and how to store the score as tracking information.

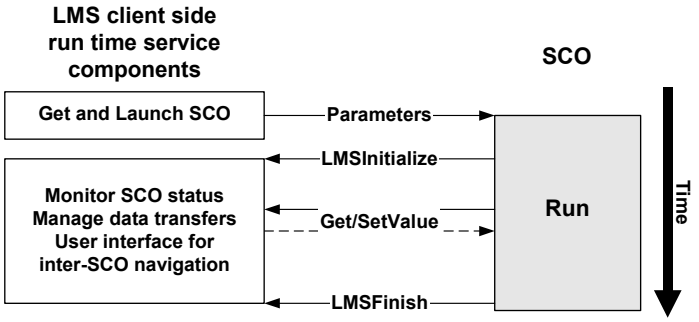


Figure 2.2 – The SCO can call the API adapter to get and set data

Behaviors and responsibilities of a data-enabled SCO

- When launched, find the API adapter by searching in a specific way.
- Having found the API adapter, call `LMSInitialize` to begin a communication session.
- During execution: Call `LMSGetValue`, `LMSSetValue`, `LMSCommit`, `LMSGetLastError` and `LMSGetErrorString` as needed throughout execution of the SCO.
- When finished, call `LMSFinish` to terminate the communication session.
- If unloaded unexpectedly, attempt to call `LMSSetValue` to communicate any unsaved data to the LMS, then call `LMSFinish`. This cleanup should be included in a handler for the `onUnload` event triggered by the unloading of the SCO. Because the behavior of browsers during execution of an `onUnload` can be unreliable, there is no guarantee that the data will effectively be received and stored by the LMS. Note that if `LMSFinish` had been called previously, all of this will be ignored by the runtime service.

Finding the API adapter

A SCO must find an API adapter in order to communicate with the LMS. Because a SCO may be launched in a frame, in a frameset, or in a pop-up window, there is a specific search order. The search ends as soon as an API adapter has been found: First, if there's a SCO parent window, look there, and then in the chain of parent windows until the top window has been searched. If no adapter was found this way, search in the opener window, if there is one, and then in the chain of parents of that window until the top window has been searched.

This strict search order is designed to allow future or advanced implementations in which more than one SCO might be launched at the same time, to guarantee that each SCO will find only the API adapter instance that was intended to communicate with it.

SCORM Communication session

Mandatory SCO-triggered events

The only two events specified in SCORM to manage the communication session between the SCO and the run time environment are `LMSInitialize` and `LMSFinish`. `LMSFinish`, is a "terminal event" in the sense that any communication from the SCO will be ignored by the runtime service after `LMSFinish` has been called successfully.

Note that calling `LMSFinish` only means that the SCO no longer needs to communicate with the runtime service. The runtime service should not interpret a call to `LMSFinish` as a signal to continue to the next SCO. Future versions of SCORM may provide an additional content flag to specify such automatic advance behavior, but there is no such flag in SCORM 1.2.

Once `LMSFinish` has been called, a SCO may not attempt to resume communication by calling `LMSInitialize` again. `LMSInitialized` can be called again only if the SCO is unloaded and then relaunched.

Unexpected unloading of a SCO

A SCO can get unloaded at any time when the user chooses another SCO. Also, by design, browsers do not prevent a user to close the stage window in which a SCO is playing at any time. To minimize the harmful consequences of such unexpected unloading, you should include an "`onunload`" event handler in the top level of the SCO that calls `LMSFinish`.

Browsers can be quirky when executing an unload event. There seem to be cases where scripts triggered by unload do not finish executing, especially if they involve high latency calls to a server, because the browser loading thread does not wait for the script thread to finish.

Because unload may be unreliable, a handler for `onbeforeunload` called before the unloading process begins is preferable. Unfortunately, `onbeforeunload` is only implemented in Microsoft Internet Explorer and is not included in the HTML standard. Also, the use of `href` to call a script, which is a common trick found on many Web pages, causes a call to `onbeforeunload`.

The safest method is to call `LMSSetValue` as soon as a significant event occurs in the SCO to save the information you want to persist, and then immediately call `LMSCommit` so that if the SCO is unloaded unexpectedly, little or no critical information will get lost.

It is OK to call `LMSFinish` more than once, but after the first successful `LMSFinish` the runtime service must ignore any other communication from the SCO, except calls to the functions to get error information.

When to call LMSFinish

To ensure that `LMSFinish` will be called even if the SCO is unloaded expectedly, a properly initialized SCO should be coded to call `LMSFinish` at the following times, and under the following circumstances:

- When it no longer needs to communicate with the runtime service. For example, when the user clicks a "submit" button at the end of a test and the user will not be allowed to change responses, a SCO may upload the results of the test with `LMSSetValue`, then call `LMSFinish`.
- If `LMSFinish` has not been called successfully yet, when an `onbeforeunload` browser event is processed (typically, this will only happen if the browser is Internet Explorer).
- If `LMSFinish` has not been called successfully yet, when an `onunload` browser event is processed.

Note: A call to `LMSFinish` is successful if the call returns "`true`." The call may return false if the runtime service determines that a back end error has happened that prevents it from terminating the communication session normally. How to handle this situation is not defined by the SCORM specification, but one way to handle it might be to try to set a timer to call `LMSFinish` again after a few seconds.

Chapter 3 - Building simple SCOs

Building the simplest SCO

This SCO implements just the required behaviors. It calls `LMSInitialize` when it is loaded, and `LMSFinish` when it is unloaded.

Sample listing 3.1 – The simplest SCO

```
<html>
<head>
<script type="text/javascript" language="JavaScript">
  var nFindAPITries = 0;
  var API = null;
  function FindAPI(win) {
    while ((win.API == null) && (win.parent != null) && (win.parent != win)) {
      nFindAPITries ++;
      if (nFindAPITries > 500) {
        alert("Error in finding API -- too deeply nested.");
        return null
      }
      win = win.parent
    }
    return win.API
  }
  function init() {
    if ((window.parent) && (window.parent != window)){
      API = FindAPI(window.parent)
    }
    if ((API == null) && (window.opener != null)){
      API = FindAPI(window.opener)
    }
    if (API == null) {
      alert("No API adapter found")
    } else {
      API.LMSInitialize("")
    }
  }
  function finish() {
    if (API != null) {
      API.LMSFinish("")
    }
  }
</script>

<title></title>
</head>
<body onload="init()" onunload="finish()">
<h1>Hello world</h1>
</body>
</html>
```

Listing comments

- Most of this script is devoted to finding the API adapter provided by the LMS in the DOM environment before the SCO can communicate with it. The search is in a specific order.
- There is an arbitrary limit on the depth of nested windows that will be searched.
- The `(window.parent != window)` test is necessary to work around a Microsoft Internet Explorer oddity.

A generic script for simple SCOs

If you analyze the script of the simplest SCO above, you will probably notice that a large part of that script would be the same regardless of the actual content of the SCO. This could be made into a reusable script. Such a reusable script may look like this:

Sample listing 3.2 – Generic reusable script for simple SCOs

```
// Saved as file "SCORMGeneric.js"
// Generic Simple SCO Script -
// Concocted by CO 2002-10-14
var nFindAPITries = 0;
var objAPI = null;
var bFinishDone = false;
function FindAPI(win) {
  while ((win.API == null) && (win.parent != null) && (win.parent != win)) {
    nFindAPITries ++;
    if (nFindAPITries > 500) {
      alert("Error in finding LMS API -- too deeply nested.");
      return null
    }
    win = win.parent
  }
  return win.API
}
function APIOK() {
  return ((typeof(objAPI)!= "undefined") && (objAPI != null))
}
function SCOInitialize() {
  if ((window.parent) && (window.parent != window)){
    objAPI = FindAPI(window.parent)
  }
  if ((objAPI == null) && (window.opener != null)){
    objAPI = FindAPI(window.opener)
  }
  if (!APIOK()) {
    alert("Learning Management System interface not found.");
    return "false"
  } else {
    return objAPI.LMSInitialize("")
  }
}
function SCOFinish() {
  if ((APIOK()) && (bFinishDone == false)) {
    bFinishDone = (objAPI.LMSFinish("") == "true")
  }
  return (bFinishDone.toString())
}
```

Listing comments

This script ensures that the API's `LMSFinish` method will not be called again if it has already been called successfully. A single page SCO that uses that simple script may look like Sample listing 3.3.

Sample listing 3.3 – Simplest SCO reusing a generic script

```
<html>
<head>
<script src="SCORMGeneric.js" type="text/javascript" language="JavaScript">
</script>
<title>A very simple SCO</title>
</head>
<body onload="SCOInitialize()"
      onunload="SCOFinish()"
      onbeforeunload="SCOFinish()">
<h1>Hello world</h1>
</body>
</html>
```

Listing comments

Notice that the `<body>` element of this HTML page includes event handlers that call functions in the generic script after the page is loaded or when it is unloaded. An event handler for `onbeforeunload`, is also called just before the page is unloaded. Only Microsoft Internet Explorer supports this event, and thus `onunload` must still be specified as a finish event trigger for other browsers. Where available, triggering on the `onbeforeunload` event is preferred because the processing of this event does not collide with the loading of another page which may be already be taking place while `onunload` is being processed. The generic script ensures that, whether `onbeforeunload or onunload` triggers the finish processing, LMSFinish will be called only once.

Chapter 4 - Your SCO as a Web page

As a content developer, you have no control over how your SCO will be launched, and in particular on the size of the stage window in which the SCO will be launched by a runtime service. Your SCO will always be launched as a Web page. That is the only thing you can rely on.

Stage window

A SCO has no control over the type or initial size or adornments of the stage window, and you should not assume that there is a fully visible table of contents or that a particular color scheme or layout will be used in the areas of the screen that surround the stage window.

For example, Click2learn's Aspen LMS launches SCOs in a frame. The dimensions of this stage window are what is left of the actual Aspen window size after display of the table of contents on the left (200 pixels) and Aspen management bar on the top (50 pixels). If your package contains only one SCO, no table of contents is displayed and the stage window for the SCO is as wide as the Aspen window. For compatibility with Aspen, Click2learn recommends that your content be tested to be usable in a frame as small as 600x550. Other LMS implementations have similar stage window size constraints.

Future versions of SCORM may include the specification of technical metadata that will allow a content publisher to specify more about the required and preferred stage window size and type. However, at this time, work is still in progress within and among standards groups to define such metadata.

Window management

A SCO may not assume that it will run in a top-level window, or attempt to force itself to run into a top-level window. It must be designed so as to be compatible with being launched in a frame that could be nested at any level of depth.

Except for finding and communicating with the API adapter object, a SCO is not allowed to communicate or manipulate other windows that exist in the environment provided by the LMS.

A SCO may open a dependent popup window but this not considered a best practice. In any case, the SCO is responsible for closing such a window before it is unloaded. a SCO is not allowed to leave any trace of itself in the user's environment after it has been unloaded. Experience has shown that the reliable use of popup windows requires very careful design and considerable testing with unexpected situations and behaviors, such as the user reactivating the wrong window, unexpected closing of the opener window, system configurations with multiple monitors, pop-up window blocking utilities, and so on. On the other hand, a dependent popup window is the only way to deliver a SCO as a full-screen user experience since the SCO has no control over the stage window in which it is delivered.

Extensions

A SCO may have extended behaviors that are not specified in SCORM, but that are agreed to between trading partners. a SCO is conformant only if it can be delivered in a strictly conformant LMS that does not implement extended behaviors. In other words, if your SCO depends on specific features of a particular LMS, or if it attempts to distort the normal behavior of a generic LMS, it is not conformant.

Chapter 5 - Turning any browser compatible resource into a SCO

Browsers can display various kinds of resources, such as an Adobe® Acrobat® document, a Macromedia® Flash movie, a text file or a picture file. Those resources cannot communicate as a SCO. Two techniques can be used to "wrap" almost any browser-compatible resource into a SCO. One technique is to create an HTML page with an object element that uses the resource. Another technique, much simpler, is to use a generic frameset that can be used to display almost any browser compatible resource. The frameset itself is the SCO.

SCO implemented as page wrapping a Web resource

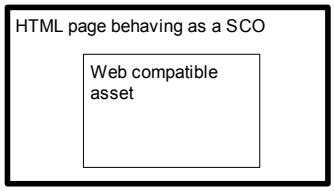


Figure 5.1 – Using an HTML page to turn a Web-compatible asset into a SCO

This wrapper uses a simple HTML page. The page reuses the simple generic script in Sample listing 3.2 to turn a picture into a SCO.

Sample listing 5.1 – Simplest wrapper to turn any asset into a SCO

```
<html>
<head>
<script src="SCORMGeneric.js" type="text/javascript" language="JavaScript">
</script>
<title>Some picture</title>
</head>
<body bgcolor="#FFFFFF" onload="SCOInitialize()" onunload="SCOFinish()"
onbeforeunload="SCOFinish()">
<img src="myimage.jpg" alt="Caption for my image" width="600" height="400" />
<body>
</html>
```

Note: This may seem to be of limited usefulness, but instead of a picture you might embed an object such as a movie, using the proper <object> tag and parameters, of course.

SCO implemented as frameset wrapping a Web-compatible asset

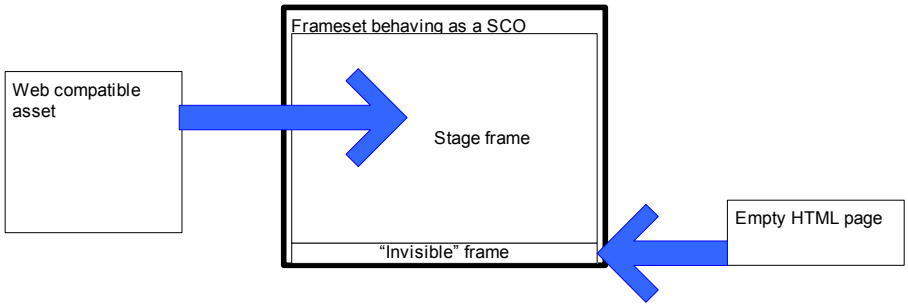


Figure 5.2 – Using a frameset to turn any Web-compatible asset into a SCO

This wrapper uses a frameset with a dummy page that will, for all practical purposes, remain invisible. The frameset uses the simple generic script in Sample listing 3.2 to turn an Adobe Acrobat document into a SCO.

Sample listing 5.2 – Simple frameset wrapper to turn any asset into a SCO

```
<html>
<head>
<script src="SCORMGeneric.js" type="text/javascript" language="JavaScript">
</script>
<title>Some PDF document</title>
</head>
<frameset border="0" rows="100%,*" onload="SCOInitialize()" onunload="SCOFinish()"
onbeforeunload="SCOFinish()">
<frame src="somedoc.pdf" scrolling="AUTO" />
<frame src="dummypage.htm" />
</frameset>
</html>
```

This technique may also work best if you want to display an external Web site. Just be careful that such an external resource does not attempt to assert the top frame, because that would effectively disrupt the delivery of the SCO in a controlled environment.

Note: The dummy page can be any valid HTML document. Older versions of Netscape may not display the frameset unless you specify a value of `"100%,1"` for the `rows` attribute, rather than `"100%,*"`. This will result in a one-pixel frame edge that will essentially blend with the bottom of the frameset window.

Using a SCO wrapper to track use of a resource

You can add a simple script that allows the SCO to report a completed status once it has been launched, this allows tracking of whether the resource has been launched.

Sample listing 5.3 –Simple wrapper to turn any asset into a SCO and report usage

```
<html>
<head>
<script src="SCORMGeneric.js" type="text/javascript" language="JavaScript">
</script>
<script type="text/javascript" language="JavaScript">
function init() {
  if (SCOInitialize() == "true") {
    objAPI.LMSSetValue("cmi.core.lesson_status","completed")
  }
}
</script>
<title>Some picture</title>
</head>
<body bgcolor="#FFFFFF" onload="init()" onunload="SCOFinish()"
onbeforeunload="SCOFinish()">
<img src="myimage.jpg" alt="Caption for my image" width="600" height="400" />
<body>
</html>
```

Listing comments

This page calls its own function called init after it is loaded. The init function then calls the `SCOInitialize` function in the generic script. If that function succeeds, i.e. if `SCOInitialize` was able to find an API adapter object and call `LMSInitialize` of that object successfully, init calls the `LMSSetValue` function of the API object to set the status to "`completed`."

Advanced uses for SCO wrappers

Translation layer for incompatible learning objects

This wrapping technique can also be used to "wrap" other scriptable Web resources that cannot communicate directly as a SCO. For example, a Macromedia® Flash movie with built-in scripted behaviors can be embedded into an HTML page that contains scripts. The Flash movie can use FSCommands to communicate with scripts on the page. Conversely, scripts on the page may get data through the SCORM interface and use them to control the Flash movie. A typical example would be bookmarking, where the SCO wrapper makes the Flash movie go to the same frame that was displayed when leaving a previous session.

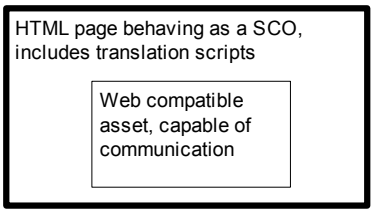


Figure 5.3 – SCO wrapper for an incompatible learning object

Translation layer for non-conformant SCOs

The frameset wrapping technique illustrated above can also be used to make SCOs run within SCOs. While this kind of application may seem perverted, it can be used, for example, to make incompletely or incorrectly coded legacy SCOs run in any SCORM compliant environment without having to rebuild them. In this case, the wrapper SCO would instantiate its own API object that the client SCO would find first. As the client SCO communicates with the wrapper's API object, the wrapper can massage data as needed before forwarding the calls to the other API adapter it found in its delivery environment.

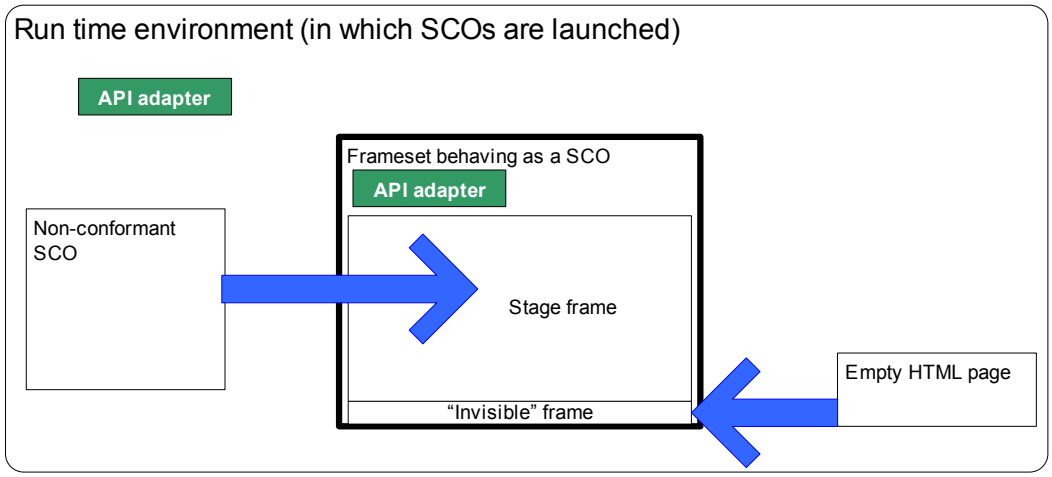


Figure 5.4 - a SCO acting as translation layer for another SCO

Chapter 6 - Building a Multiple-page SCO

Nothing in the SCORM specification prohibits you from using more than one HTML file in a SCO, and from navigating from page to page within the SCO. Indeed, in many cases a single page just cannot adequately deliver the desired learning experience.

However, there is a basic problem with using multiple HTML pages to make up a SCO. As you go from page to page, no state information is retained. For example, if page1.html called `LMSInitialize`, when you went to page2.html nothing persisted to remember that `LMSInitialize` has already been called. Since the SCO must remember at least the state of the communication session with the API, a method must be devised to remember this information. If such a method exists, it also enables the SCO to keep track of other important information, like the status of learning objectives, or what the sequence of pages should be.

We will not consider here the case where you would change assets displayed within a standing HMTL page. There is no problem there since the page itself remains loaded.

Three methods to persist state in multiple-page SCOs

Since no help can be expected from the runtime service in maintaining state information, the SCO must use a persistence method that can be implemented in a client-side browser. Three methods have been suggested:

- Using a frameset, in which the separate pages can be displayed without unloading the frameset.
- Using a separate dependent window that can remain open even as the pages change in the "stage" window. Typically that window would be "hidden" behind the main window or moved out of sight beyond the visible boundaries of the screen.
- Using session cookies.

Using a frameset for a multi-page SCO

By far the most robust method appears to be the use of a frameset. Only one frame is really used as a stage for the pages of the SCO. The other frame, which is required by some browsers, only displays a dummy page and is shrunk to the smallest possible size so that it is virtually invisible.

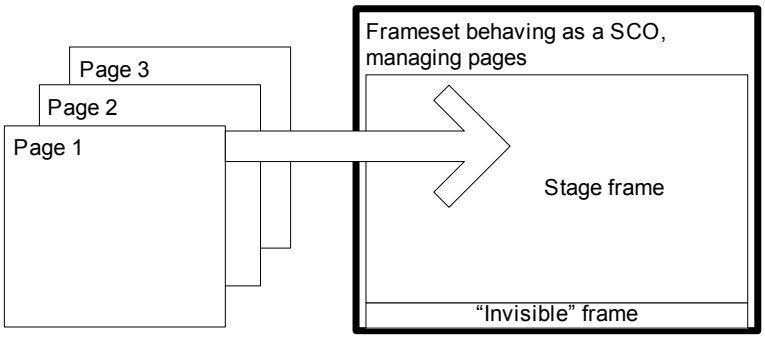


Figure 6.1 - A multiple page SCO implemented as a frameset

Sample listing 6.1 – HTML Fragment: Structure of a simple multi-page SCO frameset

```
<frameset rows="100%,*">
<frame name="myStage" title="Learning Object display window" src="page1.htm" />
<frame src="dummypage.htm" title="Empty Placeholder"/>
</frameset>
</html>
```

A complete example

This example of a three-page SCO reuses the generic script for the simplest SCOs illustrated in Sample listing 3.2. It implements the following features:

- You can navigate back and forth among the SCO pages without leaving the SCO.
- Pages do not contain complex scripts.
- The SCO can be forcibly unloaded while on any page.
- The example illustrates how a page can set the SCO status to "completed" when it is reached.

Sample listing 6.2 shows the main HTML file for the frameset.

Sample listing 6.3 shows the dummy page, which a user would normally never see, and the next three listings show pages 1 to 3 of the SCO. Except for their content and navigation objects, pages 1, 2 and 3 are identical. The scripts include a paranoid resynchronization feature, in case a navigation event, such as clicking the Back button in a browser toolbar, occurs outside the control of the frameset script: Each page confirms its number after loading so that the navigation script in the frameset cannot accidentally overshoot the bounds of the page array.

Sample listing 6.2 – HTML Source: Simple multi-page SCO frameset

```
<html>
<head>
<script src="SCORMGeneric.js" type="text/javascript" language="JavaScript">
</script>

<script type="text/javascript" language="JavaScript">
  // Manage page navigation
  var znPages=3;
  var znThisPage=1;
  function NextPage() {
    if (znThisPage < znPages){
      znThisPage++;
      myStage.location.href = "page" + znThisPage + ".htm"
    }
  }
  function PreviousPage() {
    if (znThisPage > 1){
      znThisPage--;
        myStage.location.href = "page" + znThisPage + ".htm"
    }
  }
  function SetThisPage(n) {
    znThisPage=n
  }
</script>

<title>Multiple page SCO sample</title>
</head>
<frameset rows="100%,*" onload="SCOInitialize()" onunload="SCOFinish()"
onbeforeunload="SCOFinish()">
<frame name="myStage" title="Learning Object display window" src="page1.htm" />
<frame src="dummypage.htm" />
</frameset>
</html>
<html>
<head></head>
<body> </body>
</html>
```

Sample listing 6.3 – HTML Source: Dummy page in a multi-page SCO frameset

```
<html>
<head><title>empty</title></head>
<body> </body>
</html>
```

Sample listing 6.4 – HTML Source: Page 1 of a simple multi-page SCO frameset

```
<html>
<head><title>Page 1</title></head>
<body onload="window.parent.SetThisPage(1)">
<p>This is page 1</p>
<p align="right">
<a href="JavaScript:window.parent.NextPage()">Next</a>
</p>
</body>
</html>
```

Sample listing 6.5 – HTML Source: Page 2 of a simple multi-page SCO frameset

```
<html>
<head><title>Page 2</title></head>
<body onload="window.parent.SetThisPage(2)">
<p>This is page 2</p>
<p align="right">
<a href="JavaScript:window.parent.PreviousPage()">Previous</a>
<a href="JavaScript:window.parent.NextPage()">Next</a>
</p>
</body>
</html>
```

Sample listing 6.6 – HTML Source: Last page of a simple multi-page SCO frameset

```
<html>
<head><title>Page 3</title></head>
<body onload="window.parent.SetThisPage(3)">
<p>This is page 3 (the last page)</p>
<p align="right">
<a href="JavaScript:window.parent.PreviousPage()">Previous</a>
</p>
</body>
</html>
```

Functional issues with framesets

Accessibility is an important consideration. Although the stage frame and the frame of the window in which the SCO is played will be visually indistinguishable, it is important to give each frame a title so that users who rely on assistive technologies can identify the main stage for your SCO. The header of any HTML page you display in these frames must also specify a title.

DHTML vs. framesets

Of course, DHTML can also be used instead of framesets for multi-page SCOs, but such implementations are rather more complex because different browsers and browser versions tend to require different, idiosyncratic DHTML content implementations. The frameset examples shown here have been tested successfully with a range of reasonably recent browser. Because DHTML is so difficult to code and debug, advanced authoring tools like ToolBook that generate the DHTML code while allowing you to work in a much more convivial interface are often the most appropriate way to implement DHTML based SCOs.

Chapter 7 - Managing SCO state and communications

Now that we have seen how to implement and manage basic communication with the SCORM API, it is time to do some real work. a SCO can exchange data with the runtime service through the API adapter. Typically, this may be tracking data on learner progress and success.

In the example that follows, we will create a simple SCO that communicates status and score information to the LMS.

A careful reading of the SCORM specification reveals some specific, sometimes complex rules and behaviors that govern how and when some data should be communicated. These rules are based on best practices gathered over years of managing computer-based instruction.

The sample SCO illustrated here contains a simple test item and pushes off most of the logic that manages the rules for orderly data communication into a reusable script.

A more complex reusable SCO script

This SCO will have to call other API adapter methods besides `LMSFinish` and `LMSInitialize`. It must also not get or set data before `LMSInitialize` has been successful, or after `LMSFinish` has been called. That kind of logic is common to a lot of SCOs; it makes sense to embed it in a reusable script.

In addition, SCORM 1.2 specifies certain relationships and behaviors between data elements. For example, the SCO queries the LMS for the mastery score, and that mastery score influences the pass/fail value of the status for the SCO. Or, the SCO should provide minimum and maximum score data when it reports a score, to ensure that the LMS will properly interpret the score.

The reusable script we will use in this example implements the following features:

- API communication session management: Finding the API adapter, calling `LMSInitialize` and calling `LMSFinish` are all done automatically if you put the appropriate calls in the `<body>` or `<frameset>` element of your HTML document.
- Page or frameset scripts do not need to know where or what the API adapter object is. They can call the API through transfer functions: `SCOGetValue`, `SCOSetValue`, `SCOCommit`, `SCOGetLastError`, `SCOGetErrorString`.
- Initialization: As soon as `LMSInitialize` succeeds,
  - The script queries the mode through the API and sets a flag if it's "browse."
  - If the mode is not "browse," the script queries the status through the API. If it is "not attempted," it is changed to "incomplete."
  - The script checks whether the page or frameset has a `SCOInitData` function. If it finds one, it calls it.
  - The script records the current time so it will be able to report the time elapsed in this session later.
- Cleanup: Before calling `LMSFinish`, the script reports the time elapsed in this session. Then, it checks whether the page or frameset has a `SCOSaveData` function. If it finds one, it calls it.
- Reporting session time. The script reports the time elapsed in the session, from the time initialization was complete to when the SCO finishes. A SCO-specific script may also call `SCOReportSessionTime()` at any time after initialization and before the session finishes.
- Score data sanity: Calls to `LMSSetValue` for score are managed to ensure that score min and score max are reported along with score raw. By default, raw score is assumed to be normalized in the range `0..100`. If you use another range in your SCO, overwrite the values of `SCO_ScoreMin` and `SCO_ScoreMax` by setting them to other values in your page's

script. Note that this version of the script does not check whether the runtime service provides other values for score min and max, which would also require a more complicated adjustment of the actually reported score range and raw value.

- Passed/failed status update: If a `masteryScore` value is available from the runtime service, whenever your SCO reports a score the passed/failed status is reported automatically. The default value of the `SCO_MasteryScore` pseudo-constant will be used if no mastery score is available from the runtime service. If `SCO_MasteryScore` does not evaluate to a number, passed/failed status won't be set at all.
- Completion status logic. Because in the CMI data model "failed" or "passed" status implies "completed", the script includes some logic to preserve "failed" or "passed" status if set.

Although it is much more complex than the script in Sample listing 3.2, this script can also be used by the simplest SCO. A simple, single page SCO that uses this complex script is shown in Sample listing 7.1. The only difference with Sample listing 3.3 are the URL of the included script, and the addition of a function to set the status to completed:

Sample listing 7.1 – Simplest SCO reusing a more complex script and reporting completion

```
<html>
<head>
<script src="SCORMGenericLogic.js" type="text/javascript" language="JavaScript">
</script>
<title>A very simple SCO</title>
<script type="text/javascript" language="JavaScript">
function SCOSaveData(){
  // this function will be called when this SCO is unloaded
  SCOSetStatusCompleted()
}
</script>
</head>
<body onload="SCOInitialize()"
      onunload="SCOFinish()"
      onbeforeunload="SCOFinish()">
<h1>Hello world</h1><p>My status will be set to completed when I am unloaded.</p>
</body>
</html>
```

This SCO will automatically update the status and session time information when it runs.

Recording completion with the more complex reusable SCO script

A generic, reusable script should not automatically set `cmi.core.lesson_status` to `completed` because there is no way to guess the intent of the designers of different SCOs.

The simple SCO shown above can be considered completed if the user saw it. To record this, we can just add a `SCOSaveData` function to the SCO to set the status. When `SCOFinish` is called as the SCO is unloaded, the generic script will call `SCOSaveData` before it calls `LMSFinish`. The SCO in Sample listing 7.1 includes a SCOSaveData function that in turns calls a function of the generic script to set the status to "completed". It calls that function rather than setting the status directly because some additional generic validation logic is involved.

Sample listing 7.2 shows the reusable script that does all the work. In addition to more logic, it includes better error handling than the simple generic script in Sample listing 3.2.

Sample listing 7.2 – Complex script to manage SCO state (long listing runs over several pages)

```
// Saved as file "SCORMGenericLogic.js"
// SCORM 1.2 SCO Logic management script sample
// Copyright 2001,2002,2003 Click2learn, Inc.
// This version concocted by Claude Ostyn 2003-02-22
//
// This script implements various aspects of
```

```
// common logic behavior of a SCO.
// The SCO can be a HTML document or a frameset.
//
// Change these preset values to suit your taste and requirements.
var g_bShowApiErrors = true;
var g_strAPINotFound = "Management system interface not found.";
var g_strAPITooDeep = "Cannot find API - too deeply nested.";
var g_strAPIInitFailed = "Found API but LMSInitialize failed.";
var g_strAPISetError = "Trying to set value but API not available.";

var g_nSCO_ScoreMin = 0;      // must be a number
var g_nSCO_ScoreMax = 100;    // must be a number > nSCO_Score_Min
var g_SCO_MasteryScore = 100; // value by default; may be set to null
var g_bMasteryScoreInitialized = false;

////////// API INTERFACE INITIALIZATION AND CATCHER FUNCTIONS ////////
var g_nFindAPITries = 0;
var g_objAPI = null;
var g_bInitDone = false;
var g_bFinishDone = false;
var g_bSCOBrowse = false;
var g_dtmInitialized = new Date(); // will be adjusted after initialize

function AlertUserOfAPIError(strText) {
  if (g_bShowApiErrors) alert(strText)
}

function FindAPI(win) {
  while ((win.API == null) && (win.parent != null) && (win.parent != win)) {
    g_nFindAPITries ++;
    if (g_nFindAPITries > 500) {
      AlertUserOfAPIError(g_strAPITooDeep);
      return null;
    }
    win = win.parent;
  }
  return win.API;
}

function APIOK() {
  return ((typeof(g_objAPI)!= "undefined") && (g_objAPI != null))
}

function SCOInitialize() {
  var err = true;
  if (!g_bInitDone) {
    if ((window.parent) && (window.parent != window)){
      g_objAPI = FindAPI(window.parent)
    }
    if ((g_objAPI == null) && (window.opener != null))  {
      g_objAPI = FindAPI(window.opener)
    }
    if (!APIOK()) {
      AlertUserOfAPIError(g_strAPINotFound);
      err = false
    } else {
      err =  g_objAPI.LMSInitialize("");
      if (err == "true") {
        g_bSCOBrowse = (g_objAPI.LMSGetValue("cmi.core.lesson_mode") == "browse");
        if (!g_bSCOBrowse) {
          if (g_objAPI.LMSGetValue("cmi.core.lesson_status") == "not attempted") {
              err =  g_objAPI.LMSSetValue("cmi.core.lesson_status","incomplete")
          }
```

```
        }
      } else {
        AlertUserOfAPIError(g_strAPIInitFailed)
      }
    }
  }
  if (typeof(SCOInitData) != "undefined") {
    // The SCOInitData function can be defined in another script of the SCO
    SCOInitData()
  }
  g_dtmInitialized = new Date();
  return (err + "") // Force type to string
}

function SCOFinish() {
  if ((APIOK()) && (g_bFinishDone == false)) {
    if (typeof(SCOSaveData) != "undefined"){
      SCOReportSessionTime()
      // The SCOSaveData function can be defined in another script of the SCO
      SCOSaveData();
    }
    g_bFinishDone = (g_objAPI.LMSFinish("") == "true");
  }
  return (g_bFinishDone + "" ) // Force type to string
}

// Call these catcher functions rather than trying to call the API adapter directly
function SCOGetValue(nam) {return ((APIOK())?g_objAPI.LMSGetValue(nam.toString()):"")}
function SCOCommit(parm)  {return ((APIOK())?g_objAPI.LMSCommit(""):"false")}
function SCOGetLastError(parm){return ((APIOK())?g_objAPI.LMSGetLastError(""):"-1")}
function SCOGetErrorString(n){return ((APIOK())?g_objAPI.LMSGetErrorString(n):"No API")}
function SCOGetDiagnostic(p){return ((APIOK())?g_objAPI.LMSGetDiagnostic(p):"No API")}

//LMSSetValue is implemented with more complex data
//management logic through the SCOSetValue function below

var g_bMinScoreAcquired = false;
var g_bMaxScoreAcquired = false;

function SCOSetValue(nam,val){
// Special logic to manage some special values
  var err = "";
  if (!APIOK()){
      AlertUserOfAPIError(g_strAPISetError + "\n" + nam + "\n" + val);
      err = "false"
  } else if (nam == "cmi.core.score.raw") err = privReportRawScore(val)
  if (err == ""){
      err =  g_objAPI.LMSSetValue(nam,val.toString() + "");
      if (err != "true") return err
  }
  if (nam == "cmi.core.score.min"){
    g_bMinScoreAcquired = true;
    g_nSCO_ScoreMin = val
  }
  else if (nam == "cmi.core.score.max"){
    g_bMaxScoreAcquired = true;
    g_nSCO_ScoreMax = val
  }
  return err
}
function privReportRawScore(nRaw) { // called only by SCOSetValue
  if (isNaN(nRaw)) return "false";
  if (!g_bMinScoreAcquired){
```

```
    if (g_objAPI.LMSSetValue("cmi.core.score.min",g_nSCO_ScoreMin+"")!="true"){
      return "false"
    }
  }
  if (!g_bMaxScoreAcquired){
    if (g_objAPI.LMSSetValue("cmi.core.score.max",g_nSCO_ScoreMax+"")!="true"){
      return "false"
    }
  }
  if (g_objAPI.LMSSetValue("cmi.core.score.raw", nRaw)!= "true") return "false";
  g_bMinScoreAcquired = false;
  g_bMaxScoreAcquired = false;
  if (!g_bMasteryScoreInitialized){
    var nMasteryScore = parseInt(SCOGetValue("cmi.student_data.mastery_score"),10);
    if (!isNaN(nMasteryScore)) g_SCO_MasteryScore = nMasteryScore
  }
  if (isNaN(g_SCO_MasteryScore)) return "false";
  var stat = (nRaw >= g_SCO_MasteryScore? "passed" : "failed");
  if (SCOSetValue("cmi.core.lesson_status",stat) != "true") return "false";
  return "true"
}

function MillisecondsToCMIDuration(n) {
//Convert duration from milliseconds to 0000:00:00.00 format
  var hms = "";
  var dtm = new Date();  dtm.setTime(n);
  var h = "000" + Math.floor(n / 3600000);
  var m = "0" + dtm.getMinutes();
  var s = "0" + dtm.getSeconds();
  var cs = "0" + Math.round(dtm.getMilliseconds() / 10);
  hms = h.substr(h.length-4)+":"+m.substr(m.length-2)+":";
  hms += s.substr(s.length-2)+"."+cs.substr(cs.length-2);
  return hms
}

function SCOReportSessionTime() {
// SCOReportSessionTime is called automatically by this script,
// but you may also call it at any time also from another SCO script
  var dtm = new Date();
  var n = dtm.getTime() - g_dtmInitialized.getTime();
  return SCOSetValue("cmi.core.session_time",MillisecondsToCMIDuration(n))
}

function SCOSetStatusCompleted(){
// Since only the designer of a SCO knows what completed means, another
// script of the SCO may call SCOSetStatusComplete to set completed status.
// The function checks that the SCO was not launched in browse mode, and
// avoids clobbering a "passed" or "failed" status since those imply "completed".
  var stat = SCOGetValue("cmi.core.lesson_status");
  if (!g_bSCOBrowse)
    if ((stat!="completed") && (stat != "passed") && (stat != "failed")){
      return SCOSetValue("cmi.core.lesson_status","completed")
    }
  } else return "false"
}
```

The next two examples will show how the complex script can be used to greatly simplify the creation, testing and maintenance of any SCO that reports other kinds of tracking data to an LMS.

A more complex SCO that reports tracking data when it is unloaded

The SCO in Sample listing 7.3 contains a simple assessment, in the form of a multiple-choice question with different scores associated with different responses. After the user responds, it sets the score value for the SCO to a value associated with the response. In this example, the user may change the response as many times as she wants. The status and score are reported in real time, and may be reported more than once with a different value.

This is only a fairly crude example, of course, and you could implement it in very different ways. However, this shows how a reusable script and simple conventions can hide the complexities of managing the communication state for the SCO and tracking score and status data.

As the user responds, the score is recorded by the SCO in a variable. The score is reported only when the SCO is unloaded, just before the generic script calls `LMSFinish`. This happens when the generic script seeks and finds the `SCOSaveData` function in the script of the SCO.

As a result, if the SCO uses `LMSGetValue` to get the status from the run time service, it will only get the last reported value, which may be out of sync with what the SCO knows is the current score. How to solve that problem? The answer is shown in the following example.

Sample listing 7.3 – Single-page SCO that sets SCORM data when the SCO is unloaded

```
<html>
<head>
<script src="SCORMGenericLogic.js" type="text/javascript" language="JavaScript">
</script>
<script type="text/javascript" language="JavaScript">
var znScore = 0;
function EvaluateMultipleChoiceItem(obj) {
  znScore = obj.value
}
function SCOSaveData() {
  // this function is called by the generic script before calling LMSFinish
  if (!isNaN(znScore)) {
    SCOSetValue("cmi.core.score.raw", znScore); // this will set pass/fail status
    SCOCommit() // to make sure that, no matter what happens, this has been recorded
  }
}
function ShowSCORMStatus(){
  alert('Current status is ' + SCOGetValue('cmi.core.lesson_status'))
}
</script>
<title>One question test</title>
</head>
<body onload="SCOInitialize()"
      onunload="SCOFinish()"
      onbeforeunload="SCOFinish()">
<form>Which of these is the correct answer?<br />
<input type="radio" name="q1" value="60"
onclick="JavaScript:EvaluateMultipleChoiceItem(this)" />This, kind of.<br />
<input type="radio" name="q1" value="100"
onclick="JavaScript:EvaluateMultipleChoiceItem(this)" />This, absolutely.<br />
<input type="radio" name="q1" value="0"
onclick="JavaScript:EvaluateMultipleChoiceItem(this)" />Definitely not this.<br />
</form>
<p>
<form>
<input type="button" onclick="JavaScript:ShowSCORMStatus()" value="Check Status" />
</form>
</p>
</body>
</html>
```

Sample listing comments

Notice that there is no submit action because that would cause `onbeforeunload` to be called in Internet Explorer, which in turn triggers a premature call to `LMSFinish`.. The use of a construct like `<a href="javascript:xxx()">` was also avoided because it too causes `onbeforeunload` to be called when the link is clicked. This is only a problem if the HTML document where those constructs occur is also the SCO itself. Of course, one could just not use `onbeforeunload` in the `<body>` tag, but at the risk of maybe making the SCO a little less reliable.

When you make this design choice, consider that it is good practice to make your Web page accessible. Assistive features in the browser may not trigger the onclick script events associated with text fragments; also, text fragments do not get displayed automatically as hyperlinks.

Less than optimal

Generally speaking, the approach to SCORM tracking shown in Sample listing 7.3, where data is reported only when the SCO is unloaded, is less than optimal for several reasons:

- Reporting a lot of data with unload as the trigger event may not be robust. Experiments have shown that in some implementations latency in a back end database may exceed the time allowed by the browser to finish executing the scripts of the unloaded pages.
- If there is a catastrophic event, all data recorded by the SCO during the session is lost.
- The information recorded by the run time service and the information in the SCO may get out of sync. This is typically not a serious problem, but the example shown in Sample listing 7.3 shows how it may restrict some applications because the information available through the API is out of date.

Fortunately, there is a more robust approach, which is to report significant data whenever it occurs, as shown in the next example.

A more complex SCO that reports tracking data as it occurs

Like the SCO in Sample listing 7.3, the example in Sample listing 7.4 contains a simple assessment, in the form of a multiple-choice question with different scores associated with different responses. The difference is that, as the user responds, the score is not only recorded internally, but also reported immediately through the SCORM API. So, unlike in the previous example, querying the status will return the last reported value, which will be in sync with what the SCO knows is the current score.

Sample listing 7.4 – Single-page SCO that sets SCORM data in real time

```
<html>
<head>
<script src="SCORMGenericLogic.js" type="text/javascript" language="JavaScript">
</script>
<script type="text/javascript" language="JavaScript">
function EvalMultipleChoiceItem(obj) {
  var v = obj.value;
  if (!isNaN(v)) {
    SCOSetValue("cmi.core.score.raw",v); // this will also set pass/fail status
    SCOCommit() // to make sure that, no matter what happens, this has been recorded
  }
}
function ShowSCORMStatus(){
  alert('Current status is ' + SCOGetValue("cmi.core.lesson_status"))
}
</script>
<title>One question test</title>
</head>
<body onload="SCOInitialize()"
      onunload="SCOFinish()"
      onbeforeunload="SCOFinish()">
<form>Which of these is the correct answer?<br />
<input type="radio" name="q1" value="60"
onclick="JavaScript:EvalMultipleChoiceItem(this)" />This, kind of.<br />
<input type="radio" name="q1" value="100"
onclick="JavaScript:EvalMultipleChoiceItem(this)" />This, absolutely.<br />
<input type="radio" name="q1" value="0"
onclick="JavaScript:EvalMultipleChoiceItem(this)" />Definitely not this.<br />
</form>
<p>
<form>
<INPUT TYPE="BUTTON" onClick="JavaScript:ShowSCORMStatus()" NAME="CheckStatus"
VALUE="Check Status">
</form>
</body>
</html>
```

The ambiguities of status

The SCORM 1.2 specification inherited much of the data model used for communication between the content and the runtime service from the older AICC Computer Managed Instruction (CMI) Guidelines and Recommendations. The names of the elements reflect this heritage. For example, "`cmi.core.lesson_status`" means "the lesson_status element defined in the core data model of the CMI data model."

One odd aspect of this specification is that the SCO status ("`cmi.core.lesson_status`") can have only one value, but that value must represent different kinds of status information. The allowed values that represent completion are `"not attempted"`, `"incomplete"`, `"completed"`. The allowed values that represent mastery are `"passed"` or `"failed"`. There is no way to express "incomplete and passed," for example, since that would require two values and only one is allowed.

Future versions of the SCORM specification will fix this problem, but in the meantime you must assume that "`passed`" or "`failed`" implies "`completed`", at least where any single SCO is concerned.

How and when to set status information

When a SCO is first launched, its status is "`not attempted`". SCORM specifies that this is the value that should be passed to the SCO in this case. As soon as the user can interact with it, the status of the SCO can become "`incomplete`". If the user has done everything the designer of the SCO intended her to do, the status can become "`completed`". If there is a measure of satisfaction involved, which can be summarized as pass/fail value, the status becomes either "`passed`" or "`failed`", which also means "`completed`".

Note that there is at least one case where the status may change beyond the control of the SCO: If the SCO reports a score, and the runtime service knows of a mastery score, the status is changed to "`passed`" or "`failed`" by the runtime service by comparing the reported score with the mastery score. The SCO can get the value of that mastery score and use it to determine pass/fail status also, but the runtime service will always have the last word.

The reusable script in Sample listing 7.2 manages the status information according to these simple rules, which is why the scripts for the SCOs themselves can be so simple.

Chapter 8 - A layered architecture for SCOs

You may have begun to detect a trend in the examples above: The SCO page scripts are doing only lightweight work; reusable scripts do the heavy lifting. Also, the reusable scripts themselves tend to isolate the details communication with the API from the logic of what to communicate and when.

This layered approach is summarized in Figure 8.1.

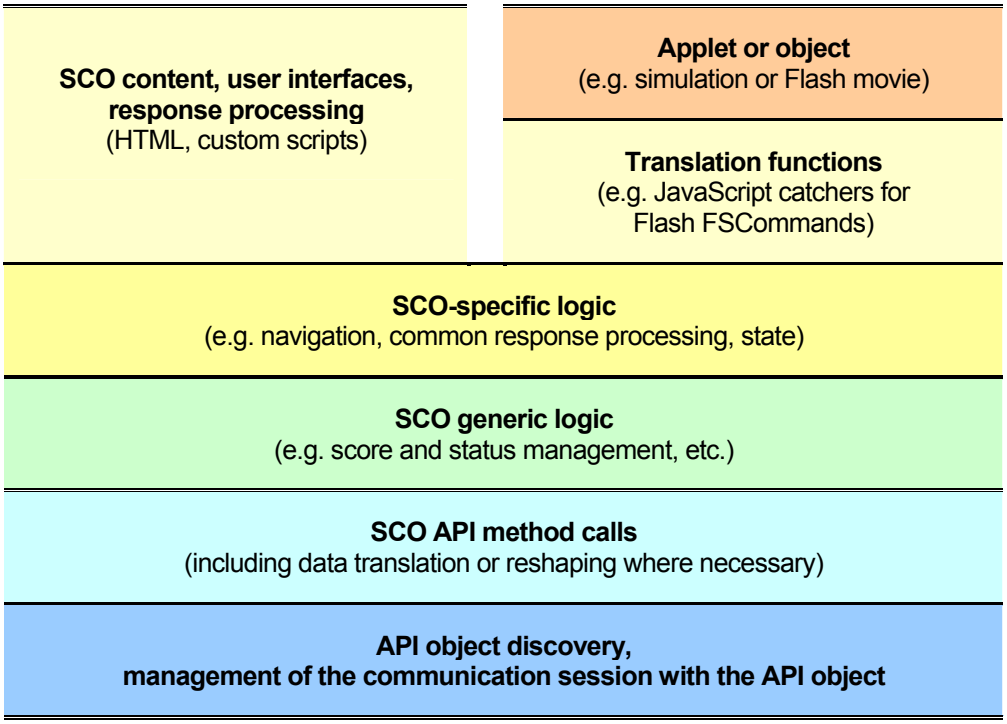


Figure 8.1 – An approach SCO scripting layers

There are three main reasons why this approach was chosen for these examples:

- Easier customization of the example SCOs. Once one of the lower layers has been debugged, it can be reused again and again and it saves time to concentrate on the upper layers.
- Ease of maintenance. By separating the functions into separate layers, layers can be maintained separately. For example, adding data translation logic to one layer does not affect the other layers.
- Future proofing. Future versions of SCORM will probably introduce subtle or not so subtle changes in the API, the API protocol, the communication data model, or error codes. Revising only the lower, reusable layers can accommodate most if not all of these changes. For example, if a future version of SCORM starts using XML to communicate data "packages" rather than relying on individual calls, the generic, reusable scripts can be modified to provide the necessary translations. If a future version of SCORM starts using SOAP instead of the DOM object API interface, this can be managed by replacing only the bottom two layers in Figure 8.1, thus preserving the investment in the more expensive custom layers of the content.

The examples that follow will build on this layered approach to add more functionality.

Chapter 9 - Suspend and resume

An attempt to complete a SCO may require more than one session. SCORM 1.2 allows the delivery of a SCO to be suspended and resumed in a later session..

Before the SCO session terminates:

- The SCO must report that its exit mode is "`suspend`".
- The SCO must ask the API adapter to store either suspend data or some designation of a location. The form and content of suspend data or location data are defined by the SCO. An LMS or runtime service does not attempt to interpret this information; it must only store it and make it available to the SCO when it is launched again in a later session. The size of the suspend data is limited to 4096 characters.

The suspend information is contextual. If a SCO is launched in a different context, i.e. for a different user, in a new attempt, or because the same SCO is used in a different learning activity, the suspend information will not carry over to that other context.

When the SCO is launched again in the same context:

- The SCO asks the API adapter whether there was a previous suspended session.
- If the response is "`resume`", then the SCO asks the API adapter for the suspend and/or location data that were stored during the previous session.

Note that nothing prevents a SCO from immediately declaring that it wants to be suspended whenever it is unloaded, and from storing suspend or location data at every significant juncture throughout the delivery of the SCO. Especially in brittle delivery environments, this allows the SCO to resume from that point if for some reason delivery was interrupted abnormally without an opportunity to do an orderly shutdown.

A multi-page SCO that can be suspended at any page

The following example is a multi-page SCO that can be suspended at any time, and resume from the page where it was suspended. For the sake of brevity, the example will only use the location data element and very simple suspend data, but you can easily expand it to store other meaningful suspend data as needed.

This example contains a few more components and behaviors than the previous examples, so we will provide a little more upfront explanation before showing the listings.

Structure

This SCO is built as a frameset, following the model in a previous chapter. However, the page structure is a little different. The page structure includes a load page, to which the user will not be able to navigate, and an end page, from which the user cannot navigate to another page.

Page 1 could be a table of content with links to other pages within the SCO, or it could be the first in a straightforward sequence of pages.

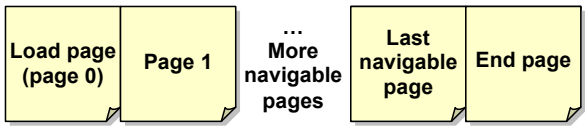


Figure 9.1 – Structure template for multiple pages

How it works

The frameset is the SCO and scripts to communicate with the API are managed by the frameset. The SCO calls the API only after it is fully loaded. Therefore, it cannot find out whether it is resuming or starting *ab initio* until a page has already been displayed. If resuming, it may have to show a page other than page 1. Therefore, when the SCO is launched, it always starts with a load page, which is not part of the normal page sequence. That page is

shown as "Page 0" in Figure 9.1. Normal navigation through the pages of the SCO will never go to page 0. It is used only to display something other than page 1 while the SCO is initializing. Using a load page also solves a problem with the timing of scripts triggered by load events, because when the browser initializes the frameset it typically trigger the load event of the pages loaded into frames before triggering the load event for the frameset itself. If this first loaded page did attempt to call the API through its parent frameset when it is loaded, it would fail.

What happens next depends on the data the SCO gets through the API adapter. If the SCO is not being resumed, or if no location was stored during the previous session, it then goes immediately to page 1. If it is being resumed, and a valid location is available from the previous session, it goes immediately to that location without ever showing page 1.

Figure 9.1 also shows an "end page." You may or may not use such an end page in your own application. If the user does something that you consider "terminal," navigation would go to that page and the user would not be able to navigate back to another page. For example, if your page set is a series of questions in a quiz, the end page could display a final score after the user clicks a "Done" button on another page. Unloading the SCO when that page is displayed would not set the exit value to "`suspend`", but it would set the exit to "" (an empty string), and report a score and possibly a status of "`passed`" or "`failed`". On the other hand, unloading the SCO when any other page is displayed would set the exit mode to "`suspend`", but would also leave the status with a value of "`incomplete`."

In this example, each time a navigable page is displayed, the SCO sets some suspend information. If the SCO is unloaded while on a navigable page, and is later launched again in the same context, this suspend information will be provided by the runtime service and used to return to the same page. The end page is used as a destination if the user clicks "All done" on the last page. If the user chose "All done," the end page is displayed. When the SCO is unloaded while on the end page, it sets the exit value to "" (an empty string). The next time the SCO is launched it will be as if it was a new beginning.

Reusing generic logic

This example reuses the script shown in Sample listing 7.2. It adds its own logic to the logic defined in that script, in order to manage the suspend and resume functionality. For the sake of brevity, this example does not include things such as tracking and setting scores, but only sets completion once all pages have been displayed to the user. Which pages have been visited is stored from session to session in suspend data. The most current page is stored from session to session as the value for the location data.

This frameset is almost exactly the same as the one in Sample listing 6.2, with the addition of logic to keep track of which page have been seen, setting completion, and suspend/resume behavior.

Sample listing 9.1 – HTML Source: Resumable multi-page SCO frameset

```
<html>
<head>
<script src="SCORMGenericLogic.js" type="text/javascript"
   language="JavaScript">
</script>

<script type="text/javascript" language="JavaScript">
// Manage page navigation
var znNavigablePages=3;
var znThisPage=0;
var zaVisitedPages = new Array(znNavigablePages)
function NextPage() {
  if (znThisPage < znNavigablePages){
    znThisPage++;
    myStage.location.href = "page" + znThisPage + ".htm"
  }
}
```

```
function PreviousPage() {
  if (znThisPage > 1){
    znThisPage--;
    myStage.location.href = "page" + znThisPage + ".htm"
  }
}
function GoToPage(n) {
  if ((!isNaN(n)) && (n >= 1) && (n <= znNavigablePages)){
    myStage.location.href = "page" + n + ".htm";
  }
}
function SetThisPage(n) { // called by each navigable page when displayed
  var i = 0; var nCnt = 0
  znThisPage = n;
  zaVisitedPages[n-1] = true;
  for (i = 0 ; i < znNavigablePages; i++){
    if (zaVisitedPages[i]) { nCnt++ }
  }
  if (nCnt == znNavigablePages) {
    SCOSetStatusCompleted()
  }
  // Make sure this is recorded, just in case
  SCOSetValue("cmi.core.exit","suspend");
  SCOSetValue("cmi.core.lesson_location", znThisPage);
  SCOSetValue("cmi.suspend_data",zaVisitedPages.join(","));
  SCOCommit()
}
function AllDone() {
  znThisPage = znNavigablePages + 1;
  myStage.location.href = "endpage.htm"
}
function SCOInitData() {
 var loc = 1;
  if (SCOGetValue("cmi.core.entry") == "resume"){
    var SuspendData = SCOGetValue("cmi.suspend_data")
    if (SuspendData.length > 0) {
      zaVisitedPages = SuspendData.split(",")
    }
    var loc =(parseInt(SCOGetValue("cmi.core.lesson_location")));
   if (isNaN(loc)) loc = 1;
  }
 GoToPage(loc)
}
function SCOSaveData() {
  if (znThisPage > znNavigablePages) {
    SCOSetValue("cmi.core.exit","") // this is the endpage, no need to resume
  }
}
</script>

<title>Sample Multiple Page SCO with Suspend and Resume</title>
</head>
<frameset rows="100%,*"
      onload="SCOInitialize()"
      onunload="SCOFinish()"
      onbeforeunload="SCOFinish()">
>
<frame name="myStage" title="Learning Object display window" src="page0.htm" />
<frame src="dummypage.htm" />
</frameset>
</html>
```

Sample listing 9.2 – HTML Source: Dummy page in a multi-page resumable SCO frameset

```
<html><!-- saved as file "dummypage.htm" -->
<head><title>empty</title></head>
<body> </body>
</html>
```

Sample listing 9.3 – HTML Source: Page 0 of a multi-page resumable SCO frameset

```
<html><!-- saved as file "page0.htm" -->
<head><title>---</title></head>
<body>One moment, please...</body>
</html>
```

Sample listing 9.4 – HTML Source: Page 1 of a multi-page resumable SCO frameset

```
<html><!-- saved as file "page1.htm" -->
<head><title>Page 1</title></head>
<body onload="window.parent.SetThisPage(1)">
<p>This is page 1</p>
<p align="right">
<a href="JavaScript:window.parent.NextPage()">Next</a>
</p>
</body>
</html>
```

Sample listing 9.5 – HTML Source: Page 2 of a multi-page resumable SCO frameset

```
<html><!-- saved as file "page2.htm" -->
<head><title>Page 2</title></head>
<body onload="window.parent.SetThisPage(2)">
<p>This is page 2</p>
<p align="right">
<a href="JavaScript:window.parent.PreviousPage()">Previous</a>
<a href="JavaScript:window.parent.NextPage()">Next</a>
</p>
</body>
</html>
```

Sample listing 9.6 – HTML Source: Last navigable page of a multi-page resumable SCO frameset

```
<html><!-- saved as file "page3.htm" -->
<head><title>Page 3</title></head>
<body onload="window.parent.SetThisPage(3)">
<p>This is page 3 (the last navigable page)</p>
<p align="right">
<a href="JavaScript:window.parent.PreviousPage()">Previous</a>
<a href="JavaScript:window.parent.AllDone()">All done</a>
</p>
</body>
</html>
```

Sample listing 9.7 – HTML Source: End page of a multi-page resumable SCO frameset

```
<html><!-- saved as file "endpage.htm" -->
<head><title>---</title></head>
<body>All done!</body>
</html>
```

Chapter 10 - Tracking objectives within a SCO

SCORM 1.2 allows a SCO to track objectives. A few facts about objectives:

- The SCO objectives are really just small records of the status for something called an objective that is private to the SCO. There may be one or more objectives tracked by a SCO. For example, if your SCO includes 3 activities, you can define a SCO objective for each activity and use that objective data to keep track of whether an activity has been done already.
- The status information you can use for an objective includes completion and score. You may use one, both or none of these data elements depending on what the objectives mean to you.
- The status of objectives is expected to be maintained by the LMS at least from session to session within an attempt to complete the SCO. But if a new attempt is started, SCORM does not specify whether this status persists or not. This is a matter of LMS implementation policy.
- The SCORM specification does not define whether SCO objectives are or are not related to a learning objective an author might associate with the use of the SCO.
- There is no defined relationship between the status of an objective and the overall status of the SCO, or between the status of an objective and the status of interactions in a SCO. If you choose, you may create such a relationship, but what they are and what they mean is not defined by SCORM. For example, you might decide that a SCO will be considered completed when all the objectives you defined for the SCO are completed. Or you might decide that a SCO will be considered completed when a score of 80% has been achieved for 67% of the objectives. Or you might use the objectives just to keep track of whether certain pages have been seen and not define any relationship with the SCO as a whole.
- It is not defined whether the objectives correspond to actual educational objectives that may be tracked outside the context of delivering this particular SCORM package. One could envision an implementation where the tracking status information for an objective may be preset before launching the content, based on existing information in the learner's profile. But there is currently no standard way to specify or expect this.
- Not every LMS keeps track of objective data. SCORM 1.2 conformance statements define this feature as optional, which means that a conforming LMS may not implement it. It has been verified that Click2learn Aspen LMS does implement it.

How objectives differ from suspend data

Suspend data and location data are opaque to an LMS. The LMS is not expected to be able to interpret the information in those data elements. Objectives, on the other hand, use a well-defined, standardized data model. This allows an LMS to read and possibly even set the objective data.

Another difference is that suspend data and location should not be maintained by an LMS between attempts, but only between sessions during the same attempt. As soon as the attempt is deemed completed, the suspend data and location can be discarded. Objective data, on the other hand, may or may not persist beyond the life span of an attempt. This is determined by the policies that govern the LMS.

A practical example

The example in this chapter is almost identical to the previous example that was using suspend data and location. The difference is that it uses objectives to keep track of whether the user has performed a couple of specific actions while traversing the SCO. With actual content, this action might be completing a practice exercise or playing a demonstration, or demonstrating mastery of a particular objective. To keep the example simple, we will just make the action the clicking of some page elements on pages 2 and 3.

What it does

As the user navigates from page to page, the appearance and behavior of pages 2 and 3 change, depending on the status of objectives related to those pages. On page 2, the user is invited to click a link if she has not done it before. If the user arrives on page 2 and that link had been clicked previously, the invitation is no longer displayed but rather a confirmation that the action had been done previously. Page 3 does the same thing, but with two actions which it tracks as two different objectives. Page 3 reuses the objective from page 2, so you can see how completing the same objective in different pages can work even when objective data is recorded out of sequence. The user may navigate backward and forward between pages 1, 2, and 3, and forward only from page 3 to the end page, which shows the final status of the objectives.

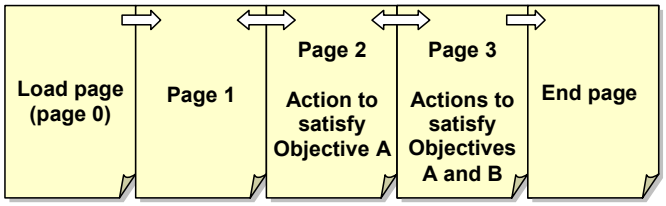


Figure 10.1 – The pages in a SCO that keeps track of some individual objectives

Technical notes about the example

This example relies on the LMS to keep track of the status of the objectives. If this SCO is run in an LMS environment that does not support SCO objectives, then it will not work properly.

This example uses the same files as the previous example, except pages 2 and 3, which have additional behavior, and the frameset itself that includes another script fragment to manage SCO objectives. Only that script fragment, which should be included in the frameset, and pages 2 and 3 will be shown entirely in the listings below. Refer to the previous example for the other components of the SCO.

Technical notes about the reusable script

Each objective has a unique identifier, but a record about each objective is stored in the equivalent of an array. Because SCORM does not specify whether the order of the records will be the same from run to run, the unique identifier should be used instead of the position in the array. Note also that an objective record can only be added to the array by specifying a position that is in sequence with existing objective records for the SCO. In other words, you cannot add an objective at position 3 unless there are already objectives at positions 1 and 2.

The SCORM 1.2 specification states that the objective ID is optional. That was a mistake that will be corrected in SCORM 1.3. Without the objective ID, you would have to rely on the relative position of the data in the objective array, which may not be reliable, in order to get the objective data. The reusable script avoids this pitfall by using the ID as the key every time you get or set objective data. If you get data about an objective, it looks for the objective record that has that ID. If you set data for an objective, it looks for the record for this objective. If it exists, it updates it. If it does not exist, it adds a record for this objective. You can see by inspecting the scripts for pages 2 and 3 and the end page how this simplifies the managing of objectives. The only thing you need to remember is to give different objectives distinct values for the ID.

Some LMS implementers have interpreted the Objectives information model as a journaling model, i.e. they create a new record for each change to an objective, rather than updating an existing record. The script works with either implementation, but always attempts to update an existing record first, and if that fails it attempts to append a record. In case the implementation already has an objective array instantiated before the SCO starts, searching for an existing objective data record among the existing records is done in reverse, to always find the latest record for that objective in case there is more than one record with the same ID value.

Sample listing 10.1 – A JavaScript fragment to manage SCO objectives

```
// Saved as file "SCORMObjectiveLogic.js"

function SCOSetObjectiveData(id, elem, v) {
  var result = "false";
  var i = SCOGetObjectiveIndex(id);
  if (isNaN(i)) {
    i = parseInt(SCOGetValue("cmi.objectives._count"));
    if (isNaN(i)) i = 0;
    if (SCOSetValue("cmi.objectives." + i + ".id", id) == "true"){
      result = SCOSetValue("cmi.objectives." + i + "." + elem, v)
    }
  } else {
    result = SCOSetValue("cmi.objectives." + i + "." + elem, v);
    if (result != "true") {
      // Maybe this LMS accepts only journaling entries
       i = parseInt(SCOGetValue("cmi.objectives._count"));
      if (!isNaN(i)) {
        if (SCOSetValue("cmi.objectives." + i + ".id", id) == "true"){
          result = SCOSetValue("cmi.objectives." + i + "." + elem, v)
        }
      }
    }
  }
  return result
}

function SCOGetObjectiveData(id, elem) {
  var i = SCOGetObjectiveIndex(id);
  if (!isNaN(i)) {
    return SCOGetValue("cmi.objectives." + i + "."+elem)
  }
  return ""
}

function SCOGetObjectiveIndex(id){
  var i = -1;
  var nCount = parseInt(SCOGetValue("cmi.objectives._count"));
  if (!isNaN(nCount)) {
    for (i = nCount-1; i >= 0; i--){ //walk backward in case LMS does journaling
      if (SCOGetValue("cmi.objectives." + i + ".id") == id) {
        return i
      }
    }
  }
  return NaN
}
```

Sample listing 10.2 – Fragment of a frameset modified to include objective logic

```
<html>
<head>
<script src="SCORMGenericLogic.js" type="text/javascript" language="JavaScript">
</script>
<script src="SCORMObjectiveLogic.js" type="text/javascript" language="JavaScript">
</script>
...
```

Sample listing 10.3 – HTML and JavaScript: Get and set data for an objective

```
<html><!-- saved as file "page2.htm" -->
<head><title>Page 2</title></head>
<body onload="window.parent.SetThisPage(2)">
<p>This is page 2</p>
<p align="right">
<a href="JavaScript:window.parent.PreviousPage()">Previous</a>
<a href="JavaScript:window.parent.NextPage()">Next</a>
</p>
<script type="text/javascript" language="JavaScript">
var s = "<p>";
var objectiveID = "myUniqueObjectiveName001";
function MarkADone(){
  window.parent.SCOSetObjectiveData(objectiveID, "status", "completed");
  window.parent.SCOCommit();
  window.location.href=window.location.href;
}
if (window.parent.SCOGetObjectiveData(objectiveID, "status") != "completed") {
  s = 'Please click <a href="JavaScript:MarkADone()">here</a> for objective A.'
}
else { s = 'You already completed objective A.' }
document.write('<p>' + s + '</p>')
</script>
</body>
</html>
```

Sample listing 10.4 – HTML and JavaScript: Get and set data for more than one objective

```
<html><!-- saved as file "page3.htm" -->
<head><title>Page 3</title></head>
<body onload="window.parent.SetThisPage(3)">
<p>This is page 3 (the last navigable page)</p>
<p align="right">
<a href="JavaScript:window.parent.PreviousPage()">Previous</a>
<a href="JavaScript:window.parent.AllDone()">All done</a>
</p>
<script type="text/javascript" language="JavaScript">
function MarkObjectiveCompleted(objectiveID){
  window.parent.SCOSetObjectiveData(objectiveID, "status", "completed");
  window.parent.SCOCommit();
  window.location.href=window.location.href;
}
var s = "";
var objectiveID = "myUniqueObjectiveName001";
if (window.parent.SCOGetObjectiveData(objectiveID, "status") != "completed") {
  s = 'Please click <a href="JavaScript:MarkObjectiveCompleted(\'';
  s += objectiveID +'\')">here</a> to complete objective A.'
}
else {
  s = 'You already completed objective A.'
}
document.write(s + '</p><p>');
objectiveID = "myUniqueObjectiveName002";
if (window.parent.SCOGetObjectiveData(objectiveID, "status") != "completed") {
  s = 'Please click <a href="JavaScript:MarkObjectiveCompleted(\'';
  s += objectiveID + '\')">here</a> to complete objective B.'
}
else { s = 'You already completed objective B.' }
document.write('<p>' + s + '</p>')
</script>
</body>
</html>
```

Sample listing 10.5 – HTML and JavaScript: Get and display data for more than one objective

```
<html><!-- saved as file "endpage.htm" -->
<head><title>---</title></head>
<body>
<p>Summary of how you did on objectives:</p>
<script type="text/javascript" language="JavaScript">
var s = "";
var stat = "";
// Get the status for one objective, and turn it
// into some text to display
var objectiveID = "myUniqueObjectiveName001";
stat = window.parent.SCOGetObjectiveData(objectiveID, "status");
if (stat != "completed") stat="not completed";
s += '<p>Status for objective A: ' + stat + '</p>';
// Get the status for the other objective
// and append that to the text to display
objectiveID = "myUniqueObjectiveName002";
stat = window.parent.SCOGetObjectiveData(objectiveID, "status");
if (stat != "completed") stat="not completed";
s += '<p>Status for objective B: ' + stat + '</p>';
document.write(s)
</script>
</body>
</html>
```

Chapter 11 - Displaying a SCO in full screen mode

A SCO may not assume that it will run in a top-level window, or attempt to force itself to run into a top-level window. It must be designed so as to be compatible with being launched in a frame. However, there are some situations where it is desirable for a SCO to run full-screen. Certain types of content, such as a game, might work best as a full-screen experience.

Since the SCO may be launched by the runtime service in a frame, the only way to provide a full-screen experience is for the SCO to open a full-screen window. Nothing in the SCORM specification prevents a SCO from opening such a dependent popup window, but this can be risky. In any case, the SCO cannot leave such a window open after it terminates, even if the termination is abnormal. Also, experience has shown that users can get very confused if they inadvertently activate another window.

This example shows a way to handle this problem in a reasonably robust way. It is a full-screen version of the example in *Chapter 10 -Tracking objectives within*.

There are many ways to do this. In this case, since most of the required scripts are already available, part of the solution is to run a SCO within a SCO, as shown in Figure 11.1.

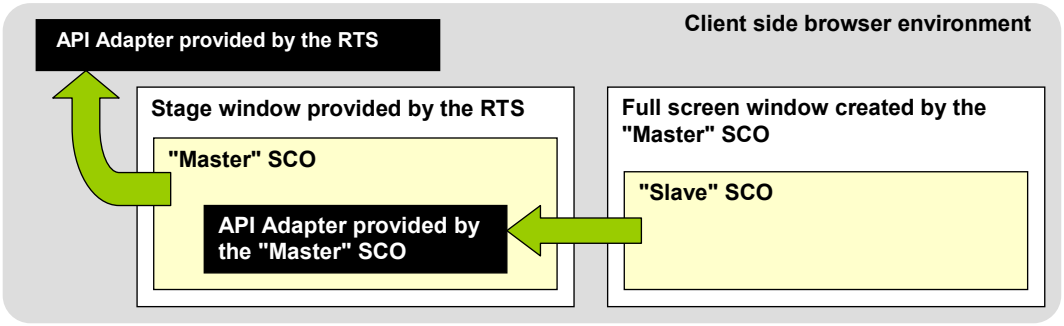


Figure 11.1 – Launching a slave SCO in a full-screen window

The actual SCO that is launched by the runtime service—let us call it the "master" SCO—acts as a mini runtime service for another SCO—let us call this one the "slave" SCO. The "master" SCO launches the "slave" SCO in a window it creates. When the "slave" SCO is launched, it seeks and finds an API adapter, which is actually part of the "master" SCO. This API adapter that is instantiated by the "master" SCO intercepts API calls from the "slave" SCO. Most of those calls just get forwarded to the "real" API adapter that is provided by the runtime service. Because the "master" SCO does initialization and finish in its own way, when the "slave" SCO calls `LMSInitialize` and `LMSFinish` the fake API adapter just returns "`true`" but does nothing.

The SCORM rules do not allow a SCO to link to another SCO, or to go to another SCO within the same window. In this case, however, the rules are not violated because the "slave" SCO does not replace the SCO launched by the runtime service, but is launched in a window that is "owned" by that SCO. In fact, the runtime service does not even know that the "slave" SCO exists, because the only SCO it sees running is the "master" SCO. And the "slave" SCO sees only the runtime service provided by the "master" SCO and is totally unaware of another runtime service beyond the "master" SCO.

User interface considerations

When the browser opens a full-screen window, it does not provide any means of closing that window; the browser's page stage fills the entire screen. Therefore you must provide a way to close the window, otherwise the user is "stuck" in that full-screen window. The example shown here just adds an "Exit" command on the end page. Clicking Exit actually closes the window. The user may also use Alt+F4 to close the window, but that is rather inelegant. A more elaborate example might use a different method. Note that, if the "master" SCO is unloaded for any reason, its script will ensure that the full screen window is closed automatically.

There is also the problem of window activation. In the Windows environment, for example, you cannot prevent the user from using Alt+Tab to activate another window. Also, you cannot prevent the user from closing the full-screen window by using Alt+F4. The script in this example uses a function triggered by a recurring timer to watch over the full-screen window. This ensures that a relevant prompt is always displayed in the "master" SCO's stage window.

SCO structure

This SCO uses two framesets. The main frameset is the "master" SCO which displays an empty page that gets updated dynamically. The second frameset and the associated pages make up the "slave" SCO. The second frameset will be displayed in a separate, full-screen window when the first frameset is launched in the runtime service's stage window.

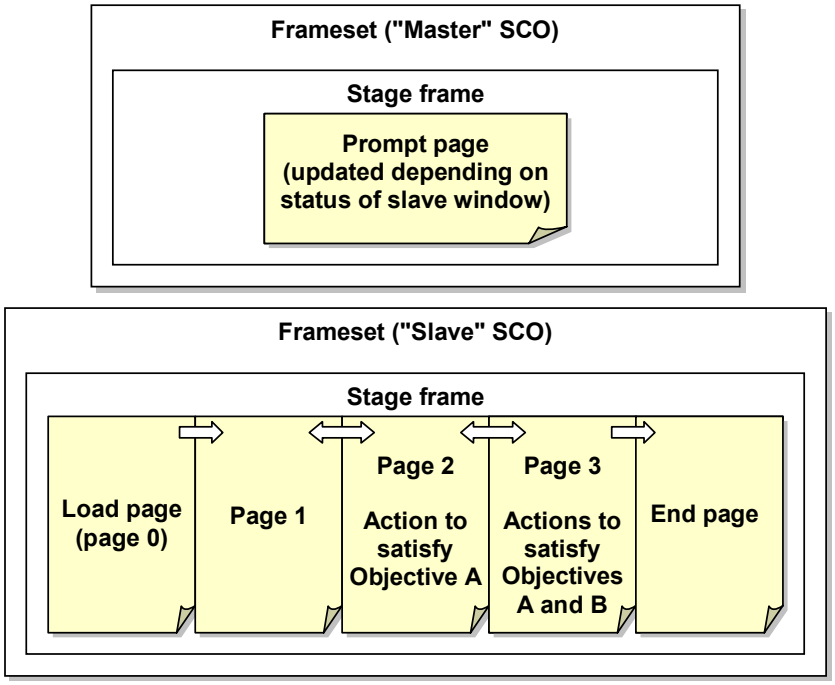


Figure 11.2 – Pages in a SCO that keeps track of some individual objectives

The main frameset—the "master" SCO—is shown in Sample listing 11.1. The prompt page used by the "master" SCO to inform the user is in Sample listing 11.2.

All the pages that make up the "slave" SCO are exactly identical to those in *Chapter 10 - Tracking objectives within*, and will not be repeated here. There is one exception: The end page, which is shown in Sample listing 11.2, is modified from that previous example because it must contain a user interface device to close the full-screen window.

Sample listing 11.1 – The "master" SCO

```
<html>
<head>
<script src="SCORMGenericLogic.js" type="text/javascript" language="JavaScript">
</script>

<script type="text/javascript" language="JavaScript">
// If all goes well this local object named API will become the
// SCORM API adapter found and used by the "slave" SCO
var API = null;
function dummyInitialize(){ return "true" }
function dummyFinish(){ return "true"}

// This part of the script ensures that the user knows
// what's going on in 3 different states:
```

```
// (1) while the SCO is creating the full frame popup window
// (2) while the full frame window is displayed, in case the
//     user inadvertently activates another window
// (3) after the full frame window closes.
// It uses a timer to control the display of a prompt in the "main"
// window.

var zPopupStageTimerID = null;
var zbOtherWindowPromptShown = false;

function DisplayPrompt() {
  var s = '<html><head>';
  s += '<style type="text/css">body {background-color: silver; color: blue;';
  s += 'font-size: small;font-family: Verdana, Arial, Helvetica, Sans-Serif}';
  s += '</style></head><body><h3>%s</h3><p>%s</p>';
  s += '</body></html>';
  var re = /%s/
  for (i=0; i < DisplayPrompt.arguments.length; i++){
    s = s.replace(re,DisplayPrompt.arguments[i]);
  }
  frameMyStage.location.href = "dummypage.htm";
  frameMyStage.document.open();
  frameMyStage.document.write(s);
  frameMyStage.document.close()
}

function PopupStageFocus() {
  if ((zwndSlave) && (!zwndSlave.closed)) {
    zwndSlave.focus()
  }
}

function PopupStageFocusTimer() {
  // called by timer while a popup stage is open
  if ((zwndSlave) && (!zwndSlave.closed)){
    if (!zbOtherWindowPromptShown){
        DisplayPrompt('This content was launched in another window.',
          'Click <a href="javaScript:window.parent.PopupStageFocus()">'+
            'here</a> to reactivate it.');
        zbOtherWindowPromptShown = true;
    }
  } else {
    KillStageFocusTimer();
    frameMyStage.location.href = "dummypage.htm"
    if (zbOtherWindowPromptShown){
        DisplayPrompt("Finished",
          "The full screen content window is now closed.")
        zbOtherWindowPromptShown = true;
    }
    zbOtherWindowPromptShown = false;
  }
}

function StartStageFocusTimer() {
  if (zPopupStageTimerID!= null) {
    clearInterval(zPopupStageTimerID)
  }
  zbOtherWindowPromptShown = false;
  zPopupStageTimerID= setInterval("PopupStageFocusTimer()", 500);
}

function KillStageFocusTimer() {
  if (zPopupStageTimerID != null) clearInterval(zPopupStageTimerID);
```

```
}

// The actual work begins here
function init() {
  DisplayPrompt("One moment, please...","Preparing a window...");
  SCOInitialize();
  if (APIOK()){
    API = new Object();
    // We are handling overall SCO initialize and finish in this script
    // Make the methods called by the slave no-ops
    API.LMSInitialize = dummyInitialize;
    API.LMSFinish = dummyFinish;
    // Add the functions instantiated by the generic script to this API adapter
    API.LMSSetValue = zobjAPI.LMSSetValue;
    API.LMSGetValue = zobjAPI.LMSGetValue;
    API.LMSCommit = zobjAPI.LMSCommit;
    API.LMSGetLastError = zobjAPI.LMSGetLastError;
    API.LMSGetErrorString = zobjAPI.LMSGetErrorString;
    API.LMSGetDiagnostic = zobjAPI.LMSGetDiagnostic;
  }
  // Now launch the actual content in full screen window
  var url = "frameset.htm" ; // Note: could be read from a parameter.
  // Open window with a blank page because it may take a long
  // time to load the actual content; psychologically it is
  // better to see the window come up as fast as possible.
  zwndSlave = window.open("dummypage.htm","SCOFullScreenStage","fullscreen=yes");
  zwndSlave.location.href = url;
  zwndSlave.focus();
  StartStageFocusTimer()
}

function SCOSaveData(){
  KillStageFocusTimer();
  // Make sure that the slave window is not left behind
  if ((zwndSlave) && (!zwndSlave.closed)) {
    zwndSlave.close()
  }
}

</script>

<title>Sample Full Screen SCO</title>
</head>
<frameset rows="100%,*" border="0"
      onload="init()"
      onunload="SCOFinish()"
      onbeforeunload="SCOFinish()">
>
<frame name="frameMyStage" title="Explanation" src="loadpromptpage.htm" />
<frame src="dummypage.htm" />
</frameset>
</html>
```

Sample listing 11.2 – Initial prompt page displayed while the full-screen window is opening

```
<html><!-- saved as file "loadpromptpage.htm" -->
<head>
<style type="text/css">
body {background-color: silver;
 color: blue;
 font-size: small;
 font-family: Verdana, Arial, Helvetica, Sans-Serif}
</style>
<title></title>
</head>
<body>
<h3>One moment, please...</h3>
<p>Preparing a window...</p>
</body>
</html>
```

Sample listing 11.3 – End page with a user-interface device to close the full-screen window

```
<html><!-- saved as file "endpage.htm" -->
<head><title>---</title></head>
<body>
<script type="text/javascript" language="JavaScript">
var s = "<p>Status: ";
var stat = "";
// Get the status for the SCO
stat = window.parent.SCOGetValue("cmi.core.lesson_status");
switch(stat) {
  case "": s += window.parent.SCOGetDiagnostics("");break;
  case "completed": s += "completed (you visited every navigable page)";break;
  default: s += stat
}
s += "</p><p>Summary of how you did on objectives:</p>"
// Get the status for one objective, and turn it
// into some text to display
var objectiveID = "myUniqueObjectiveName001";
stat = window.parent.SCOGetObjectiveData(objectiveID, "status");
if (stat != "completed") stat="not completed";
s += '<p>Status for objective A: ' + stat + '</p>';
// Get the status for the other objective
// and append that to the text to display
objectiveID = "myUniqueObjectiveName002";
stat = window.parent.SCOGetObjectiveData(objectiveID, "status");
if (stat != "completed") stat="not completed";
s += '<p>Status for objective B: ' + stat + '</p>';
document.write(s)
</script>
<p><a href="JavaScript:window.parent.close()">Exit</a></p>
</body>
</html>
```

Chapter 12 - Packaging SCORM-compliant content

Having built some SCOs, it would be nice to see them in action in a real delivery environment. SCORM specifies how to package your SCOs so that they can be aggregated, stored, copied, moved, archived, uploaded and eventually delivered to a user by any SCORM-compliant management system. A package may contain one SCO, or many SCOs.

IMS & SCORM content organization model

The IMS Global Learning Consortium (www.imsglobal.org) has developed a packaging specification for learning content that provides a useful blueprint to generic content organization in the form of a manifest included with the package. The manifest is used to inventory the content of the package, but also to describe it through metadata. The manifest may also be used to show how the content is organized. SCORM 1.2 content packaging is based on the IMS specification.

An IMS packaging manifest is an XML document that contains several parts:

- Metadata that describe the package.
- Organizations: Zero, one or more structural maps that describe how the content is organized. Each item in such a map can reference a resource in the package. For SCORM content for delivery to an end user, the manifest must contain at least one organization.
- Resources, which specify actual chunks of content that can be used. More than one organization item can reference the same resource. A resource can also have its own metadata. A SCO is typically described by a resource.
- Sub-manifests (nested manifests), which describe a subset of the content in a package. A sub-manifest can have its own metadata, organizations and resources.

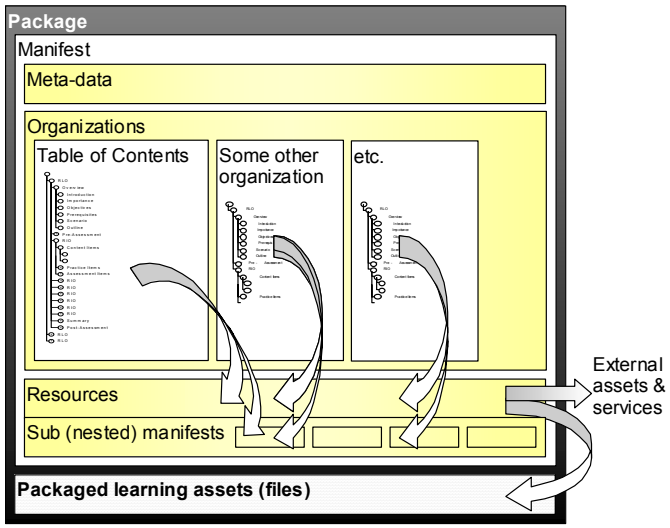


Figure 12.1 – The IMS Content Packaging model

For example, if you were to display an organization as a table of contents, by selecting an item in the table of contents you could launch or open the corresponding resource.

Instead of an atomic resource, an item in an organization may reference a sub-manifest, and thus an entire other organization. This can be very useful if the same organized "chunks" of content are used in more than one place in a course, for example. This also makes the packaging of organizations in the shape of a directed graph more convenient. Because they are self-contained and can have metadata, sub-manifests can also be useful as a way to identify the parts of a package that can be extracted and reused in another context.

This packaging model is the foundation for the SCORM 1.2 content structure and organization. SCORM 1.2 extends the package definition by specifying additional data elements. In the XML document, those extensions are identified by the namespace prefix *adlcp*:

Where to put the metadata

SCORM requires metadata for the package. You put this metadata at the top level of the manifest. SCORM also adds the option to reference a separate metadata file included in the package. To conform to SCORM 1.2, you may either include the metadata in the manifest directly, or use the SCORM-defined extension to reference an external metadata file. You cannot do both, i.e. you may not have both inline metadata in the manifest and a reference to a metadata file. SCORM also allows additional metadata as defined by the IMS schema.

To create a SCORM package

Create an XML *manifest*. This is an XML file with a header as specified in SCORM 1.2.

- Add a *resources* section and describe each learning object (SCO) as a *resource* element in the resources section.
- Unless you are referencing external resources, identify all the files that are needed for each learning object as *file* elements in each *resource*.
- If several learning objects use the same files (for example, the same graphic), consider using the *dependency* element, and making that element reference a common *resource* that is a collection of assets, rather than repeating the lists of files several times.
- Add an *organizations* section above the resources section.
- Create one or more *organization* trees that contain *item* elements that reference the resources.
- Add a *metadata* section above the organizations section.
- Add metadata elements as specified in SCORM 1.2 to describe the package and its content.

Create a zip file that contains the manifest, the XML control files (XSD, etc.) and all the files included in the package. You can use subdirectories, but the manifest must be in the base directory of the directory tree. See the IMS Content Packaging Best Practices guide for instructions. SCORM conformance and interoperability requires that you include the XSD files specified in SCORM 1.2. Those are not the most recent XSDs posted on the IMS Web site.

Content paths and directories

When the SCORM content you created is installed in an LMS or repository, it will probably not end up in the same directory as on your development system, or have the same access rights associated with it. Therefore you must follow these rules in the organization of any internal links in your Web content:

- All files used, linked or referenced by any web content in the package must be in a directory that is either the base directory of the package, or a descendant of that directory.
- All references must be relative. You may not hard wire paths relative to either the disk root directory or a specific disk drive or volume. For example, the values of *href* and *src* attributes in HTML files may not begin with a "/" (forward slash), or specify a relative path that goes further toward the root than the base directory of the package.

SCORM 1.2 defines the metadata element Location as mandatory. However, since a package can be moved from system to system, the only value that makes sense for that element is ".", meaning "wherever this metadata happens to be."

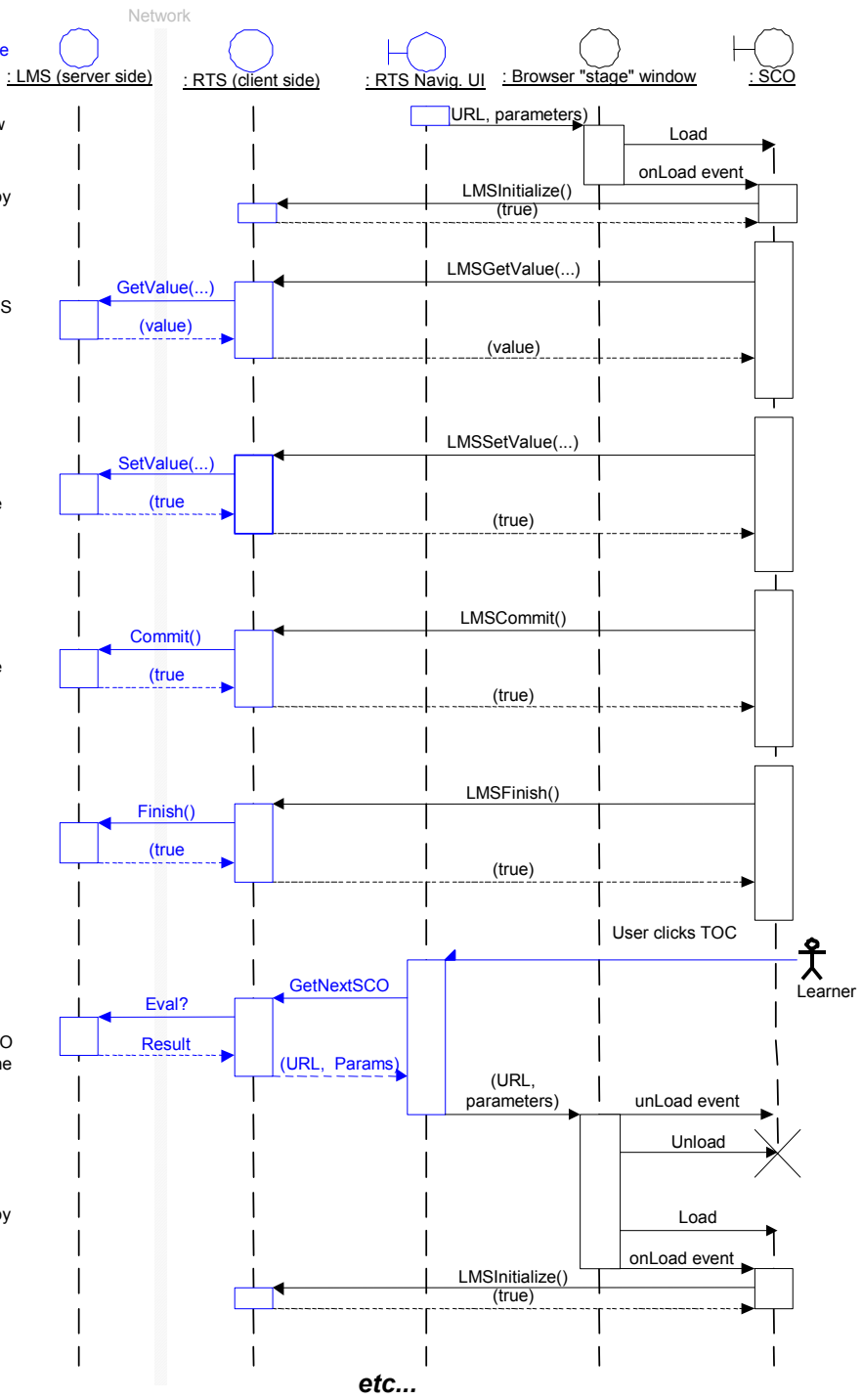
Organizing SCOS

You can associate a SCO with any item in the organization tree. Note however that the Simple Sequencing model for SCORM 1.3 will probably allow SCOs only for leaf nodes in the tree.

Appendix: Miscellaneous resources and implementation notes

Sequence diagram

Note: The implementation of the LMS and RTS (a.k.a. API adapter) can vary and is included in diagram only to indicate possible latency and other dependencies.



The RTS opens the SCO in a stage window

SCO calls LMSInitialize (usually triggered by onload event)

SCO calls LMSGetValue one or more time throughout its execution. Depending on LMS implementation, client returns value from local store or calls into server side.

SCO calls LMSSetValue one or more time throughout its execution. Depending on implementation, client side RTS caches the value or forwards the Setvalue call

SCO calls LMSCommit one or more time throughout its execution. Depending on implementation, client side RTS caches the values or forwards all unsaved data to the LMS in a private commit process.

SCO calls LMSFinish once when it determines that is has finished execution. Depending on implementation, client side RTS manages the event or or forwards all unsaved data to the LMS in a private commit and finish process.

Some navigation event in the client side RTS navigation user interface is going to force unloading of the SCO by closing the stage window or loading another SCO.

But first the RTS must determine which SCO to launch next.This may require querying the LMS. If no valid SCO to go to, navigation is canceled at this point.

The RTS opens another SCO in the stage window, thus forcing the current SCO to unload.

SCO calls LMSInitialize (usually triggered by onload event)

Etc. until the next RTS navigation event.

Figure 12.2 – SCO execution sequence with no surprises

Figure 12.2uses a flavor of UML to represent the life cycle of a SCO launched by a runtime service. This example is probably a rare one. It assumes that the SCO finishes normally (whatever that means to the author of that SCO) and reports all its data to the LMS before the user thinks of going to another SCO. In reality, a SCO may be forcibly unloaded at any time, as shown in Figure 12.3. Note: RTS stands for "Runtime service".

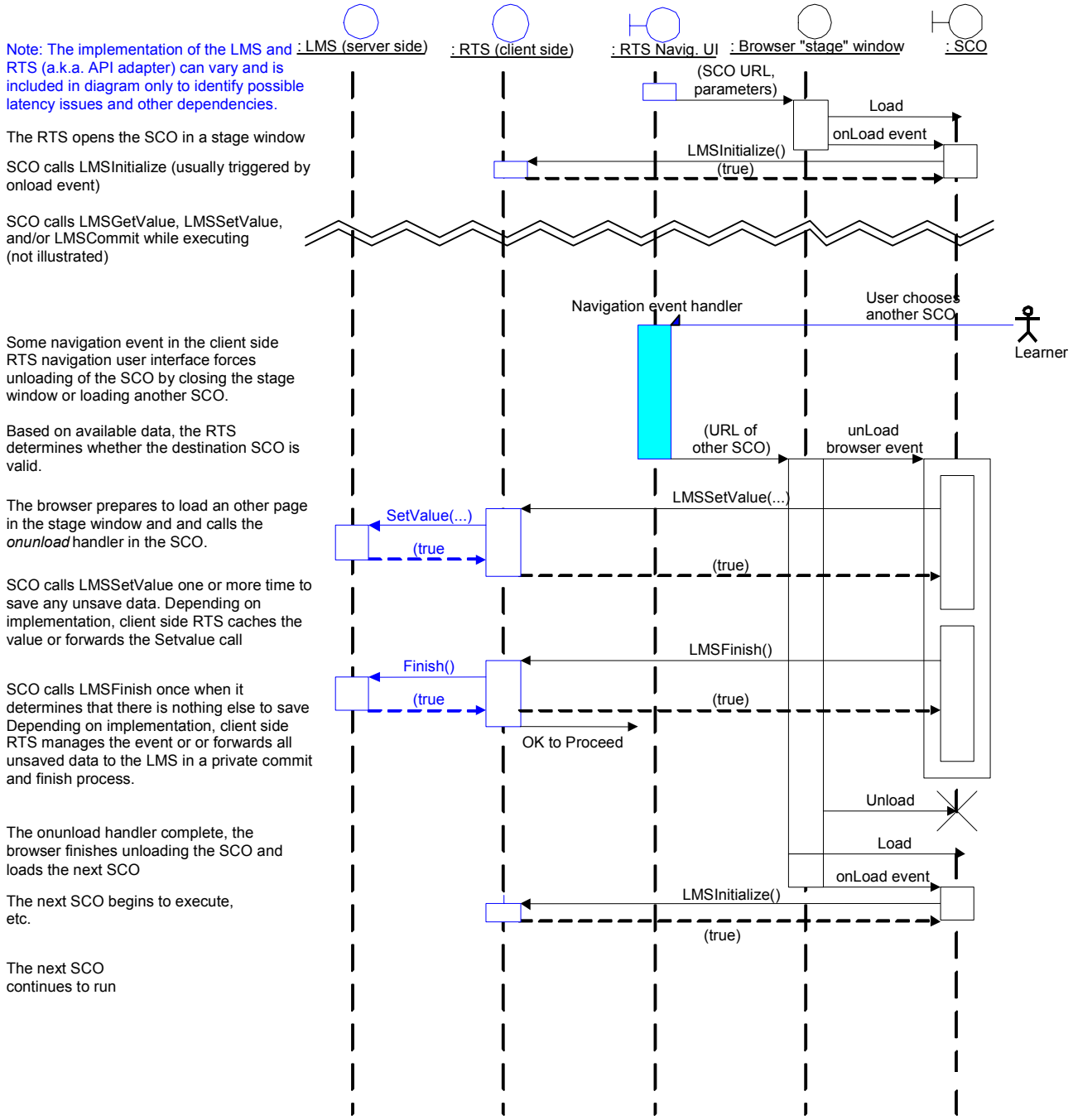


Figure 12.3 – SCO execution sequence with forced unloading of the SCO

The most likely scenario is that the browser summarily unloads the SCO whenever some browser event occurs, over which the SCO has no control. For example, this happens when the user chooses another SCO. The diagram below shows how such a scenario might occur in an implementation. Note that the browser is already processing the URL of the other SCO—and who knows what else—even as the SCO is being notified that it is being unloaded. This is why it may be a good idea to save any critical data while the SCO is being executed, rather than waiting for an unload event.

Sample SCORM 1.2 package manifest

XML manifest for a single SCO

This manifest can be used together with the files that make up the multiple-page SCO example above. All that is needed to complete the package is the set of XML schema files that are referenced in xsi:schemaLocation, and you have a complete SCORM 1.2 package. Note that every file used by the SCO must be listed in the manifest. The SCORM test suite does not actually check for this—it only checks whether the files listed here do exist. It is good practice to create a complete inventory of the required files in the manifest. Note that SCORM 1.2 requires a number of data elements that are optional in the IMS Content Packaging and Metadada specifications.

You can reuse this manifest to package any single SCO, such as a single-page SCO and the graphics it uses, or a single-page SCO wrapper in which a Flash file would be embedded, for example. The parts of the manifest you would replace with your own data are indicated in bold in the sample.

Note that the <location> element is required in for SCORM metadata, but should always be "."

```
<?xml version="1.0"?>
<manifest identifier="SampleContentManifest" version="1.2"
 xmlns="http://www.imsproject.org/xsd/imscp_rootv1p1p2"
 xmlns:adlcp="http://www.adlnet.org/xsd/adlcp_rootv1p2"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.imsproject.org/xsd/imscp_rootv1p1p2
 imscp_rootv1p1p2.xsd
 http://www.imsglobal.org/xsd/imsmd_rootv1p2p1 imsmd_rootv1p2p1.xsd
 http://www.adlnet.org/xsd/adlcp_rootv1p2 adlcp_rootv1p2.xsd">
  <metadata>
    <schema>ADL SCORM</schema>
    <schemaversion>1.2</schemaversion>
    <lom xmlns="http://www.imsglobal.org/xsd/imsmd_rootv1p2p1">
      <general>
        <catalogentry>
          <catalog>Your favorite Catalog</catalog>
          <entry>
            <langstring>Samples-SCO12-A</langstring>
          </entry>
        </catalogentry>
        <title>
          <langstring>Sample Multiple page SCO</langstring>
        </title>
        <description>
          <langstring>This multi-page SCO implements the following features:
(a) You can navigate back and forth among the SCO pages without leaving the SCO.
(b) Pages do not contain complex scripts.
(c) Can be forcibly unloaded while on any page.
(d) Illustrates how a page can set the SCO status to "completed" when it is
reached.</langstring>
        </description>
        <keyword>
          <langstring>Sample</langstring>
        </keyword>
        <keyword>
          <langstring>SCO</langstring>
        </keyword>
      </general>
      <lifecycle>
        <version>
          <langstring>1.0</langstring>
        </version>
```

```
        <status>
          <source>
            <langstring xml:lang="x-none">LOMv1.0</langstring>
          </source>
          <value>
            <langstring xml:lang="x-none">final</langstring>
          </value>
        </status>
        <contribute>
          <role>
            <source>
              <langstring xml:lang="x-none">LOMv1.0</langstring>
            </source>
            <value>
              <langstring xml:lang="x-none">author</langstring>
            </value>
          </role>
          <centity>
            <vcard>
BEGIN:vCard
FN:Claude Ostyn
ORG:Click2learn, Inc.
END:vCard
            </vcard>
          </centity>
          <date>
            <datetime>2002-02-26</datetime>
          </date>
        </contribute>
      </lifecycle>
      <metametadata>
        <metadatascheme>ADL SCORM 1.2</metadatascheme>
        <language>en-US</language>
      </metametadata>
      <technical>
        <format>text/html</format>
        <location type="URI">.</location>
        <requirement>
          <type>
            <source>
              <langstring xml:lang="x-none">LOMv1.0</langstring>
            </source>
            <value>
              <langstring xml:lang="x-none">Browser</langstring>
            </value>
          </type>
          <name>
            <source>
              <langstring xml:lang="x-none">LOMv1.0</langstring>
            </source>
            <value>
              <langstring xml:lang="x-none">Microsoft Internet Explorer</langstring>
            </value>
          </name>
          <minimumversion>4.01</minimumversion>
        </requirement>
      </technical>
      <rights>
        <cost>
          <source>
            <langstring xml:lang="x-none">LOMv1.0</langstring>
          </source>
          <value>
```

```
            <langstring xml:lang="x-none">No</langstring>
          </value>
        </cost>
        <copyrightandotherrestrictions>
          <source>
            <langstring xml:lang="x-none">LOMv1.0</langstring>
          </source>
          <value>
            <langstring xml:lang="x-none">Yes</langstring>
          </value>
        </copyrightandotherrestrictions>
        <description>
          <langstring>Can be used freely but author retains copyright.</langstring>
        </description>
      </rights>
      <classification>
        <purpose>
          <source>
            <langstring xml:lang="x-none">LOMv1.0</langstring>
          </source>
          <value>
            <langstring xml:lang="x-none">Educational Objective</langstring>
          </value>
        </purpose>
        <description>
          <langstring>Recognize a SCO Packaging Manifest</langstring>
        </description>
        <keyword>
          <langstring>SCO</langstring>
        </keyword>
        <keyword>
          <langstring>Package manifest</langstring>
        </keyword>
      </classification>
    </lom>
  </metadata>
  <organizations default="One">
    <organization identifier="One">
    <title>Sample Multiple page SCO</title>
      <item identifier="item1" identifierref="SCO1" isvisible="true">
        <title>Sample Multiple page SCO</title>
      </item>
    </organization>
  </organizations>
  <resources>
    <resource identifier="SCO1" type="webcontent"
      adlcp:SCORMtype="SCO" href="multipage_SCO_sample.htm">
      <file href="multipage_SCO_sample.htm"/>
      <file href="dummypage.htm"/>
      <file href="page1.htm"/>
      <file href="page2.htm"/>
      <file href="page3.htm"/>
      <file href="SCORM1_2Generic.js"/>
    </resource>
  </resources>
</manifest>
```

About the author

Claude Ostyn has been involved with Learning Technology Standards for several years as a contributor and member of various working groups of the IEEE Learning Technology Standards Committee and the IMS Global Learning Consortium. He was also part of the technical consultants team that helped in the genesis of the SCORM specifications.

He has been with Click2learn since the early days when the company was still called Asymetrix. He directed the design and implementation of what was, way back then, a new kind of hypertext help system with state of the art on line tutorials. He later added multimedia widgets to ToolBook and designed ToolBook CBT edition, the first e-learning authoring tool to use wizards, templates and smart, context-sensitive objects rather than coding and flowcharts, which later evolved into Click2learn Instructor.

His background includes a Film and Video degree, extended stints in remote Alaskan places as artist in the schools, ethnographic, training, and commercial video production, stand-up training of adults in areas ranging from filmmaking to spreadsheets, an Education degree from Stanford and the occasional use of an excellent recipe for chocolate mousse that begins with "take some good, dark Belgian chocolate…"

Photo: J.R. Garcia