xdc·script 1.1

xdc·script is a general-purpose programming language based upon the ECMA-262 Edition 3 standard – popularly known as JavaScript 1.5. A central element of XDC, the language primarily serves as an implementation vehicle for hosted *meta-content* used to automate integration of companion *target-content* within executable programs tailored for a specific hardware platform and optimized for a specific application environment. XDC also leverages the langauge to manage the flow of *packages* – standardized containers holding re-usable target content plus its attendent meta-content – between producers and consumers. And of course, the language is readily suited for standalone scripting tasks.

Working in tandem with xdc·spec – a special-purpose language for defining client-visible programmatic features of target- [and meta-] content through **`module`**, **`interface`**, and **`package`** specifications – xdc·script augments standard JavaScript with modular, type-safe, and expressive mechanisms that meld nicely with the familiar style and substance of the C-centric target programming domain. References to terms defined in the xdc·spec grammar are written-this-way.

« Base »

xdc·spec incorporates all core JavaScript 1.5 classes, as defined in the ECMA-262 Edition 3 standard.

```
Arguments     Error        Number
Array         Function     Object
Boolean       Global       RegEx
Date          Math         String
```

« Root »

All xdc·script properties and methods are ultimately rooted in a distinguished (read-only) object referenced by the global identifier `xdc`. This name along with any identifier containing one or more $'s should be treated as a reserved word.

`xdc`

The distinguished root object.

`xdc.global` « Base »

The original JavaScript `Global` object.

`xdc.jre` « Java »

The Java Runtime Environment.

`xdc.csd`

The directory of the script file currently being loaded.

`xdc.exec(`*`command, attrs, status`*`)`

Executes an arbitrary shell command and returns the command's exit status. The parameter *`command`* can be one string containing the command and the command line parameters, or a string array with the command in the first element of the array.

The optional parameter *`attrs`* controls the command's environment and accepts the following fields: *`envs`* (array of environment variable settings), *`cwd`* (the command's current working directory), *`outName`* (the name of the output file to create, if the file exists, output is appended to the end of the file) and *`filter`* (regular expression used to filter command output).

The optional parameter *`status`* contains the fields *`output`* (string containing all output from command) and *`exitStatus`* (exit status of the command).

`xdc.findFile(`*`fileName`*`)`

Returns a path to *`fileName`* or else `null` if not found. If *`fileName`* is an absolute path or else begins with `./`, this path is returned if the file exists. Otherwise, the file is first searched in the `xdc.csd` directory and then in each directory along the system package path.

`xdc.loadCapsule(`*`fileName`*`)` **« Capsule »**
`xdc.loadTemplate(`*`fileName`*`)` **« Template »**

Loads a capsule (template) from *`fileName`*, and returns a Capsule (Template) object. Calls `xdc.findFile()` internally, and throws an Error if the capsule (template) is not found.

`xdc.loadPackage(`*`pkgName`*`)` **« Package »**

Loads a package whose qualified-package-name is *`pkgName`*, adds the package to the current Object-Model, and returns a Package object. Throws an Error if the package is not found along the system package path.

`xdc.module(`*`modName`*`)` **« Module »**

Returns the Module object whose qualified-unit-name is *`modName`*. If *`modName`* is not qualified with a package name, `xdc.om.$homepkg` is presumed. Calls `loadPackage()`if needed. Throws an Error if the module is not found.

`xdc.om` **« Object-Model »**

The current Object-Model – *program-config, package-build, standalone-scripting,* etc.

`xdc.traceEnable(`*`mask`*`)` **« Capsule »**

This function accepts string-valued masks that designate a set of named capsules for which calls to the special function `$trace(msg)` result in the `msg` string being printed. If called with no arguments, the function returns the current trace mask; otherwise, the function updates the current trace mask and returns its previous value.

The trace mask itself comprises a set of semicolon-separated patterns that are suffix-matched against the canonical filename of the capsule. Occurrences of slashes (forward or backward) within the patterns are appropriately normalized. Wildcards ('*') within the patterns match arbitrary character sequences that themselves do not contain slashes. For convenience, capsules can be designated without any particular file extension (e.g., ".xs").

This function is called upon startup with the value of the environment variable 'xdc.traceEnable', if the latter is defined.

`xdc.useModule(`*`modName`*`)` « Module »

Declares usage of a module whose qualified-unit-name is *modName*, and returns a Module object. If *modName* is not qualified with a package name, `xdc.om.$homepkg` is presumed. Calls `loadPackage()` if needed. Throws an Error if the module is not found.

« Java »

xdc·spec supports connectivity to all classes in the current Java Runtime Environment (JRE). Public fields and functions of static class objects as well as newly constructed class instance objects can be accessed.

```
xdc.jre.javaClass.staticField =
xdc.jre.javaClass.staticFxn()

var jobj = new xdc.jre.javaClass()

jobj.field =
jobj.fxn()
```

« Capsule »

A fundamental unit of implementation. Each capsule exposes a set of public variables and functions to its client.

```
var Cap = xdc.loadCapsule(fileName)

Cap.publicVar =
Cap.publicFxn()
```

« Capsule-Body »

Contains top-level definitions of public and private variables/functions. Also contains top-level statements executed exactly *once* – on first call to `loadCapsule()` naming this file.

```
var publicVar =
var _privateVar =

function publicFxn( ... ) { ... }
function _privateFxn( ... ) { ... }

stmt ...
```

« Template »

A synthesized function that encapsulates a prescribed pattern for generating formatted output. Clients invoke templates with specific arguments and designate a sink for all generated output

```
var Tplt = xdc.loadTemplate(fileName)
```

`Tplt.genFile(`*`fileName, thisObj, args, clobber`*`)`

Generates output to *fileName,* which is created if necessary. Unless *clobber* is **true,** an existing file is overwritten only when its content has truly changed. Internally calls `genStream()`.

`Tplt.genStream(`*`outStream, thisObj, args`*`)`

Generates output to *outStream,* an instance of class `java.io.PrintStream`. All three parameters are passed to the underlying template generation function.

« Template-Body »

Contains lines of text that the synthesized function automatically outputs when invoked. Also contains lines of script to be inserted into the synthesized function. Interpolated script can reference any object in the current environment – with `$out, this,` and `$args` bound to parameters of `genStream()`.

text`` `script `` *text* ... `\n`

A line of text to be output by the synthesized function at this point. All interpolated script is replaced with its (textual) value in the current context.

`%` *script* ... `\n`

A single line of script to be inserted into the synthesized function at this point.

```
%%{                 \n
    stmt ...        \n
    stmt ...        \n
%%}                 \n
```

Multiple lines of script to be inserted into the synthesized function at this point.

« Package »

Reflects a package-specification. A read-only container for Capsule, Template, Interface, and Module objects that implements the `xdc.IPackage` interface. Its `init()` function is invoked *once* – on the first call to `xdc.loadPackage()`naming this package.. Each package is also an element of the current Object-Model.

```
var pkg = somePkgObject
```

`pkg.InterName` « Interface »

A particular interface contained within this package, unless *`InterName`* matches a name in the `xdc.IPackage` interface – in this case the xdc.IPackage interface value is returned. Because of this ambiguity, this method of referencing a package's interfaces is deprecated.

`pkg.ModName` « Module »

A particular module contained within this package, unless *`ModName`* matches a name in the `xdc.IPackage` interface – in this case the xdc.IPackage interface value is returned. Because of this ambiguity, this method of referencing a package's modules is deprecated.

`pkg.$interfaces` « Interface »

An array of all interfaces within this package.

`pkg.$modules` « Module »

An array of all modules within this package.

`pkg.$name`

The qualified-package-name of this package.

`pkg.$vers`

An array of numbers formed from the package's compatibility key array followed by the package's release date in milliseconds since midnight 1/1/1970 GMT. If the compatibility key array has fewer than three elements, the release date is not appended to the array.

« Object-Model »

A specialized execution context for scripts which comprises a model-dependent prologue and epilogue. May also introduce global variables representing "anchor points" within the model as well as codifiy **`metaonly`** interfaces to be implemented by constituent packages.

`xdc.om.$homepkg`

A model-specific home package. Unless **`undefined`**, `xdc.useModule()` resolves unqualified module names within this package.

`xdc.om.$name`
The name of the current model.

`xdc.om[pkgname]` « Package »
The package named *pkgname* within the current model.

`xdc.om.$packages[]` « Package »
An array of all packages within the current model.

« Interface »

Reflects a unit-specification for an **`interface.`**

```
var Inter = someInterObj
```

`Inter.CONST_NAME` « Typed-Value »
A scalar value of the type specified in a const definition.

`Inter.ENUMVAL_NAME` « Typed-Value »
A scalar value of the type specified in an enum definition.

`Inter.EXTERN_NAME` « Typed-Value »
A scalar value of the type specified in an extern definition.

`new` `Inter.StructName` « Struct »
A pointer to an aggregate of the type specified in a struct definition.

`Inter.Module(someModObj)` « Module »
Casts a Module object to the **`Module`** handle type of this interface. Returns **`null`** if incompatible.

`Inter.Instance(someInstObj)` « Instance »
Casts an Instance object to the **`Instance`** handle type of this interface. Returns **`null`** if incompatible.

`Inter.$name`
The qualified-unit-name of this interface.

`Inter.$package` « Package »
The Package object which contains this unit.

`Inter.$super` « Interface »
An interface object corresponding to the inherited-interface for this interface, or **`null`** if not specified.

« Module »

Reflects a unit-specification and its module-wide-feature(s)

`var Mod = someModObj` « Typed-Object »

`Mod.CONST_NAME` « Typed-Value »

A scalar value of the type specified in a const definition.

`Mod.ENUMVAL_NAME` « Typed-Value »

A scalar value of the type specified in an enum definition.

`Mod.EXTERN_NAME` « Typed-Value »

A scalar value of the type specified in an extern definition.

`new Mod.StructName` « Struct »

A pointer to an aggregate of the type specified in a struct definition.

`Mod.configName =` « Typed-Operation »

A property of the type specified in a config-parameter definition. Assignable, unless **`readonly`** in the specification.

`Mod.fxnName( ... )` « Typed-Operation »

A method whose signature is specified in a function-declaration.

`Mod.create(createParams)` « Instance »

A special method that returns a new Instance object managed by this unit.

`Mod.Module(someModObj)` « Module »

Casts a Module object to the **`Module`** handle type of this module. Returns **`null`** if incompatible.

`Mod.Instance(someInstObj)` « Instance »

Casts an Instance object to the **`Instance`** handle type of this module. Returns **`null`** if incompatible.

`Mod.$instances[]` « Instance »

An array of Instance objects managed by this module.

`Mod.$name`

The qualified-unit-name of this module.

`Mod.$package` « Package »

The Package object which contains this module.

`Mod.$private`

A private object used by capsules implementing module-wide functions specified for this module.

`Mod.$super` « Mod »

A Module object of the **`Module`** type designated in the inherited-interface for this module, or **`null`** if not specified.

`Mod.$used`

`true` if this module has been named in a call to `xdc.useModule`, `false` otherwise.

« Instance »

Reflects per-instance-feature(s) of a unit-specification.

`var inst = someInstObj` « Typed-Object »

`inst.configName =` « Typed-Operation »

A property of the type specified in a config-parameter definition. Assignable, unless `readonly` in the specification.

`inst.fxnName( ... )` « Typed-Operation »

A method whose signature is specified in a function-declaration.

`inst.$index`

A serial number assigned when created by its managing module.

`inst.$module` « Module »

The Module object that manages this instance.

`inst.$package` « Package »

The Package object which contains the module that manages this instance.

`inst.$private`

A private object used by capsules implementing per-instance functions specified for the module managing this instance.

`inst.$super` « Instance »

An Instance object of the `Instance` type designated in the inherited-interface for the module managing this instance, or `null` if not specified.

« Module-Body »

A capsule that implements all (inherited) module-wide and per-instance `metaonly` functions declared in a module's unit-specification. These capsule files are named *`UnitName.xs`* and reside in the base directory of the containing package. A special capsule named `package.xs` implements the (module-wide) functions specified in the `xdc.IPackage` interface.

```
function fxnName(arg ... ) {                    « Typed-Assignment »
    this.$private = ...
    return expr                                 « Typed-Assignment »
}
```

Implements a module-wide or per-instance function specified for this module. `this` references the corresponding Module or Instance object, whose `$private` state is manipulated in this capsule. Typed-Assignment rules are enforced upon function entry and exit.

```
function instance$init(createParams) {
    this.$private = ...
}
```

A distinguished function called upon creating a new Instance object, to which `this` is bound. Typically used to initialize the `$private` state for this Instance.

« Struct »

An aggregate of the type specified in a struct definition. Comprises a fixed number of values of different types, selected by name. A constrained variant of a JavaScript object.

```
var str = someStructObj                         « Typed-Object »
```

`str.fieldName =`

An assignable property of the type designated in a struct-field definition.

« Vector »

An aggregate that comprises an expandable ordered set of values of some common base type, indexed by non-negative integers. A constrained variant of a JavaScript array.

```
var vec = someVectorObj                         « Typed Object »
```

`vec.length =`

Gets or sets the number of values in this vector.

`vec[number] =` « Typed-Operation »

Gets or sets an element of this vector whose index is in the range `0..length-1`.

`vec.$add(arg)` «Typed-Assignment»

Appends *arg* to the end of this vector. Equivalent to `vec[length++] = arg`.

`vec.$addrof(number)` «Typed-Object»

Returns an object of the **Ptr** type. The object represents the address of the element `vec[number]` on the target. Such an object can be used at the configuration time to refer to an address that is generally not known at the configuration time.

The expression `vec.$addrof(vec.length)` can be used to refer to the first address after `vec[length-1]`.

« Map »

An aggregate that comprises an expandable ordered set of values of some common base type, indexed by strings as well as non-negative integers. A constrained hybrid of a JavaScript object and array.

`var map = someMapObj` « Typed-Object »

`map.length`

Gets the number of values in this map.

`map[number] =` « Typed-Operation »

Gets or sets an element of this map whose index is in the range `0..length-1`.

`map[string] =` « Typed-Operation »

Gets or sets a map element given a string-value key. Returns **`undefined`** if the designated element does not exist.

The command `delete map[string]` removes the named element from this map and decrements `length`.

« Typed-Object »

Common properties and methods shared by Module, Instance, Struct, Vector, and Map objects. These objects also belong to a *specified type* ultimately defined within a unit-specification.

`tobj``.$copy()` « Typed-Object »

Returns a deep-copy of this object, leaving references intact (Struct, Vector, Map only)

`tobj``.$orig`

Returns an object with the most specific typed-declaration for this object rather than the declared type of *tobj*

`tobj``.$seal(`*`fieldName`*`)`

Renders the specifed field within *tobj* read-only. If *fieldName* is omitted, the entire object is made read-only.

`tobj``.$sealed`

`true` if this object has been sealed.

`tobj``.$self` « Typed-Object »

This object. Useful for bulk assignment when *`tobj`* is a JavaScript variable.

`tobj``.$type`

A string representation of a typed-declaration for this object.

tobj.$unseal(*fieldName*)

Renders the specifed field within *tobj* writable. If *fieldName* is omitted, the entire object is made writable.

« Typed-Operation »

Each category of Typed-Object defines selectors for getting and setting constituent values of some specified type. Unit and Instance objects additionally define callable functions with parameter and return values of some specified type.

tobj.ident → « Typed-Value »
tobj[*number*] → « Typed-Value »
tobj[*string*] → « Typed-Value »

Get a value of a specified type

tobj.ident ← *expr* « Typed-Assignment »
tobj[*number*] ← *expr* « Typed-Assignment »
tobj[*string*] ← *expr* « Typed-Assignment »

Set a value of a specified type. *expr* must yield an assignment-compatible value.

tobj.fxn(*arg* ...) → « Typed-Value »
arg ← *expr* « Typed-Assignment »
return ← *expr* « Typed-Assignment »

Call a function of a specified type. Assignment-compatibility is enforced for each *expr* passed as a function argument and for the value returned from within the function body.

« Typed-Value »

All values belong to either *standard types* supported in the base language or else to *specified types* ultimately defined within some unit-specification. Each is identified by a characteristic syntactic pattern comprising type-constructors of the form *T*<...> to denote a particular type as well as type-variables *t* that range over any of these patterns.

T<**int**> *T*<**float**>... « Base »
T<**boolean**> « Base »
T<**string**> « Base »

Standard scalar types with JavaScript number, boolean, or string values.

T<{ *p*: *t* ... }> « Base »
T<[*t* ...]> « Base »

Standard aggregate types with JavaScript objects or arrays as values

`T<null>` « Base »
`T<undefined>` « Base »

Standard singleton types with `null` or `undefined` as their only values.

`T<struct S>` « Struct »
`T<struct S*>` « Struct »

Specified types with Struct objects or Struct pointers for values, the latter mimicing a JavaScript object reference. The name S comes from a corresponding struct definition.

`T<t[length]>` « Vector »
`T<t[string]>` « Map »

Specified types whose values are Vector or Map objects.

`T<U.Module>` « Module »
`T<U.Instance>` « Instance »

Specified types whose values are opaque handles for Module or Instance objects. The name U comes from a corresponding unit-specification.

`T<void*>`
`T<t*>`

Specified types whose values connote target program addresses.

`T<any>`

A universal type that subsumes all possible standard and specified types.

« Typed-Assignment »

Rules for assignment compatibility.

`t-LValue` ← `t-RValue`

Typed-Assignment pattern expressed in terms of LValue and RValue types. Enforced when assigning a specified selector, binding specified function arguments, and executing a `return` statement from within a specified function.

`t` ← `t`
`t` ← `T<undefined>`
`T<any>` ← `t`

Identical types are assignment compatible; any type can be assigned `undefined`; and any type of RValue can be assigned to an LValue specified `any`.

`T<struct S>` ← `T<struct S>`
`T<struct S*>` ← `T<struct S*>`

Struct objects based on a common definition are assigned field-wise using the rules of Typed-Assignment – that is, *by value*. Struct pointers are assigned to one another without getting or setting any constituent fields of the underlying object – that is, *by reference*.

`T<struct S>` ← `T<struct S*>`
`T<struct S*>` ← `T<struct S>`

Field-wise assignment involving a mixture of Struct objects and pointers. Implicitly gets or sets all fields of the underlying object referenced through a Struct pointer.

`T<struct S>` ← `T<{ fld: t-fld ... }>`
`T<struct S*>` ← `T<{ fld: t-fld ... }>`

Assignment from a standard JavaScript object. Named properties of the latter corresponding to fields specified in the definition of S are assigned using the rules of Typed-Assignment. Each *t-fld* must be assignment-compatible with the type of S.*fld*.

`T<t[length]>` ← `T<t´[length]>`
`T<t[string]>` ← `T<t´[string]>`

Vectors and Maps are assigned by value, by effectively removing all elements from the destination object and then performing an element-wise Typed-Assignment from the source object. The element types *t* and *t´* must be assignment-compatible.

`T<t[length]>` ← `T<[ t-elem ... ]>`
`T<t[string]>` ← `T<[[T<string>,t-elem ] ... ]>`

Assignment to Vectors or Maps from a standard JavaScript array. Elements of the destination object are replaced (in order) by elements in the source array. Each *t-elem* must be assignment-compatible with the base type *t*.

`T<U.Module>` ← `T<U´.Module>`
`T<U.Instance>` ← `T<U´.Instance>`

Assignment of opaque references to Module or Instance objects. The unit-specification(s) for U and U´ are either the same, or else the former must directly/indirectly inherit from the latter.

`T<U.Module>` ← `T<{ cfg: t-cfg ... }>`
`T<U.Instance>` ← `T<{ cfg: t-cfg ... }>`

Assignment from a standard JavaScript object. Named properties of the latter corresponding to module-wide or per-instance configs specified in the definition of U are assigned using the rules of Typed-Assignment. Each *t-cfg* must be assignment-compatible with the type of U.*cfg*.

`T<void*>` ← `T<struct S>`
`T<void*>` ← `T<struct S*>`
`T<void*>` ← `T<t*>`

Assignment to **void*** covers Struct objects/pointers plus arbitrary target addresses.

```
T<struct S*>  ← T<null>
T<U.Module>   ← T<null>
T<U.Instance> ← T<null>
```

Assignment of **null** limited to Struct pointers, target addresses, and Module/Instance handles.