xdc·spec is a special-purpose language for expressing the logical structure of software content in terms of three higher-level constructs – *modules*, *interfaces*, and *packages*. Enforcing a clean separation between software specification and software implementation, an xdc·spec source file effectively serves as a programmatic contract between content suppliers (producers) and their clients (consumers). Incorporating familiar C constructs for defining client-visible constants, types, and functions, a specification expressed in xdc·spec often resembles a “cleaned-up” rendition of a legacy header file.

While comparable to an IDL (Interface Definition Language), the true power of xdc·spec lies in its ability to forge a *single* specification that serves *two* congruent programming domains: [1] a target-domain, where the specified content is implemented in C/C++ and executes on a suitable hardware platform; and [2] a meta-domain, where this same content is potentially configured to match specific system requirements *before* target execution begins. In support of the latter, xdc·spec works hand-in-hand with xdc·script – a general-purpose language based on industry-standard JavaScript – used by producers and consumers alike to implement host-based *meta-programs* that ultimately beget target executables.

lexical-elements

The lexical structure of xdc·spec closely tracks that of JavaScript, which in turn is patterned after familiar conventions originating in C. The following table summarizes the different sorts of lexical elements potentially found in an xdc·spec source file:

Element	Examples
Whitespace	*space*, *tab*, *newline*
single-line comment	`// text`
block comment	`/* text */`
C-style identifier	`main` , `int32` , `_private`
JavaScript number	`123` , `0xff` , `6.02e23`
JavaScript string	`"hello\n"` , `'abc'`

syntactic-elements

The following table summarizes the different sorts of syntactic elements appearing throughout the xdc·spec reference grammar:

Element	Examples
syntactic production	requires-statement , unit-category
language keyword	**`package`** , **`module`**
language identifier	*`dirname`* , *`UnitName`*
literal value	*`number`* , *`string`*
literal symbol	**`[`** , **`}`** , **`;`**
documentation comment	**`//!`** ***`@xdoc`***

optional term	*grammar-term* ?
alternative terms	*grammar-term* | *grammar-term*
zero-or-more occurrences	*grammar-term* *
one-or-more occurrences	*grammar-term* +
group-of terms	[*grammar-term grammar-term*]

package-specification

Within the realm of xdc·spec, a package is a programmatic element that logically contains modules and interfaces – collectively termed *units* – within its scope. Besides introducing its own public name, a package specification may identify other named packages upon which the current package in some way depends. The specification may also declare its (in)compatibility with earlier versions of the same package, as well as further constrain the level of compatibility required of any dependent packages. Specially-formatted comments embedded in package-specification source file are processed by the xdoc utility when generating HTML documentation for a set of packages.

```
package-specification
    requires-statement*
    //! @xdoc
    package qualified-package-name compatibility-key?
    {
        unit-declaration-list*
    } [;]?
    //! @xdoc
requires-statement
    requires qualified-package-name compatibility-key? ;
qualified-package-name
    dirname [.dirname]*
compatibility-key
    [ number [,number]* ]
unit-declaration-list
    unit-category UnitName [,UnitName]* ;
unit-category
    module | interface
```

- A qualified-package-name should in general be globally-unique. Packages are located in a directory with a matching name found along the system package path.
- A package-specification resides in a distinguished source file named `package.xdc`, found in the corresponding package directory.

- Each unit-declaration-list introduces individual modules and interfaces within the scope of the current package, all of which must be uniquely named. Units with the same name may only appear in different packages.
- Each requires-statement designates another package upon which the current package in someway depends. The requires-relation among packages cannot contain cycles.
- A compatibility-key is generally interpreted as an numeric array of the form `[m,s,r,p]` where `m` denotes major functionality, `s` denotes source level, `r` denotes specification radius, and `p` denotes a particular patch. A new release of a package is *source-compatible* with any predecessor in which `m` is the same, and is *binary-compatible* with that predecessor if `s` is the same as well.
- By convention, package names are composed of `lowercase` identifiers whereas units (modules or interfaces) are named with `TitleCase` identifiers.

unit-specification

A unit specification defines all client-visible programmatic features of a module or interface. While virtually identical vis-à-vis their xdc·spec syntax, semantically this pair of programmatic elements are almost opposites: a module is *concrete* and *closed*, comprising a public specification accompanied by a conforming implementation; an interface is *abstract* and *open*, comprising only a public specification which others may import and ultimately implement. In general, the specification of a module or interface can inherit features from exactly one other (interface) specification, which itself can inherit additional features in the same manner – as if the latter's cumulative set of features had been directly defined with the unit specification of its inheritor.

The features defined within the scope of a unit partition themselves into two main groups: *module-wide* features, which are associated with a solitary programmatic object encapsulating the implementation of some concrete module; and *per-instance* features, which are associated with a family of programmatic objects individually created and manipulated by the same underlying module. Aside from auxiliary definitions of supporting constants and types (deemed module-wide for convenience), features with a meaningful run-time presence in the underlying implementation – assignable configuration parameters along with callable functions – can participate in either category. As with packages, special documentation comments can be associated with the module/interface as a whole as well as with any named feature defined within its scope.

In the general case, a single unit specification for a module or interface defines a presence in two congruent programming domains: [1] the *target*-domain, where specified features become accessible within executable programs (written in C/C++) running on some particular platform; and [2] the *meta*-domain, where specified features become accessible within hosted scripts (written in xdc·script) used to configure these very same target programs ahead of their execution. In some cases, the specification of a particular module or interface can be restricted to the meta-domain – useful when constructing configuration "facades" atop legacy content as well as when distributing host-based meta-content for use in a broader range of scripting contexts.

```
unit-specification
    //! @xdoc
    metaonly? unit-category UnitName
       inherited-interface?
    {
       module-wide-feature*
    [ instance:
       per-instance-feature*
    ]?
    } [;]?
    //! @xdoc
inherited-interface
    inherits qualified-unit-name
qualified-unit-name
    [qualified-package-name .]? UnitName
module-wide-feature
    //! @xdoc
    [      auxiliary-definition
       |   config-parameter
       |   function-declaration
    ]
per-instance-feature
    //! @xdoc
    [      config-parameter
       |   function-declaration
    ]
```

- A qualified-unit-name effectively extends the (already) globally-unique name of the unit's containing package. If the qualified-package-name prefix is absent, the current package name is presumed.
- A unit-specification resides in a source file named `UnitName.xdc`, found in the directory of its containing package.
- Each module-wide-feature or per-instance-feature introduces an individual feature with the scope of the current unit, all of which must be uniquely named. Features with the same name may only appear in different units.
- An inherited-interface designates a *single* interface whose features are introduced within the scope of the current unit. Any module or interface can optionally inherit features from another interface, so long as the inherits-relation among all units remains acyclic.
- Each unit manifests itself in both the target- and meta-programming domains, unless designated **`metaonly`**. As a rule, inheritors of metaonly interfaces must themselves be metaonly modules or interfaces.

auxiliary-definition

An xdc·spec auxiliary definition defines a (module-wide) constant or type, often supporting other module-wide or per-instance features defined in the same unit. These definitions are also used by clients who consume this unit, both within any modules or interfaces they may specify as well as within any target (or meta) content they may implement. For the most part, the syntax and semantics of each form of auxiliary definition is patterned after a familiar programmatic construct already found in C. Further semantic restrictions guarantee auxiliary definitions have a meaningful manifestation in the meta-domain as well as in the target-domain.

```
auxiliary-definition
    const | enum | extern | struct
const
    const typed-declaration = initializer ;
enum
    enum EnumName {
       enum-value [, //! @xdoc
       enum-value ]* //! @xdoc
    } [;]?
enum-value
    ENUMVAL_NAME [= initializer]?
extern
    extern typed-declaration = symbolName ;
struct
    struct StructName {
       struct-field*
    } [;]?
struct-field
    typed-declaration ; //! @xdoc
typedef
    typedef typed-declaration ;
```

- A const is restricted to numeric types, either standard or else enumerated. Its persistent value is defined by a statically-evaluated initializer consistent with its typed-declaration.
- An enum is a new numeric type that ranges over a finite set of named values. As in C, integer values beginning with `0` are assigned to each successive enum-value (unless altered by an explicit numeric initializer). Each named value is actually resident in the scope of the containing module or interface, and hence must be unique among all features defined in this unit.

- An extern is a special form of constant that effectively aliases an external program symbol naming a C-language function or variable. Its typed-declaration is restricted to standard C types, including arbitrary pointer-types.
- A struct defines a new aggregate type comprising a set of assignable fields of any type. Each struct-field must be uniquely named within the scope of the enclosing struct.
- A typedef effectively defines a *synonym* for the type specified in its typed-declaration, rather than a new type per se. As in C, typedef names can appear in other typed declarations not unlike previously-defined enum or struct names.
- By convention: `UPPER_CASE` identifiers are used to name a const, enum value, or extern; `TitleCase` identifiers are used to name an enum/struct types as well as in typedefs; and `camelCase` identifiers are used to name struct fields.

config-parameter

A configuration parameter is a feature that behaves like a "property" of the underlying module or instance object – a readable (and sometimes writeable) variable of virtually any type. In the most general case, module-wide configuration parameters are assigned within the meta-domain and then become persistent constants within the target-domain; per-instance configuration parameters are likewise assignable within the meta-domain, but are limited to supporting run-time instance creation within the target-domain. Where appropriate, configuration parameters can be restricted to the meta-domain as well as designated readonly after initialization.

A configuration parameter inherited from some previously specified interface can itself be overridden – typically to (re-)define its initial value. A configuration parameter can also be finalized, effectively freezing its definition and precluding further overrides.

config-parameter
    config-modifiers
    readonly? **config** typed-declaration [= initializer]? ;

config-modifiers
    final? **override**? **metaonly**?

- The optional initializer must yield a value consistent with the typed-declaration, using the rules of assignment-compatibility defined by xdc·script for the meta-domain. If no initializer is supplied, the configuration parameter starts out **undefined** in the meta-domain.
- If **override** is specified among the config-modifiers for a configuration parameter, its typed-declaration as well as its use of **readonly** and **metaonly** must *exactly* match that of the inherited configuration parameter being overridden. Configuration parameters marked **final** cannot be overridden.
- A **readonly** configuration parameter without an initializer can still be assigned a persistent value in the meta-domain, during construction of an underlying module/instance object.

- By convention, configuration parameters names are `camelCase` identifiers.

function-declaration

An xdc·spec function declaration generally stipulates the signature – argument and return types – of a callable routine implemented through a concrete module in either the target-domain or else (if so indicated) in the meta-domain; a target function also manifests itself in the meta-domain as an extern symbol of a function-pointer type derived from the stipulated signature. Following C++, default values can be specified for the last k arguments of an n-ary function, enabling the same routine to be called with as few as $n - k$ inputs; an untyped sequence of optional trailing arguments can be also specified using the familiar `...` notation. For those meta-domain functions wishing to adopt a more "weakly-typed" style supported (but not necessarily encouraged!) by the xdc·script language, their corresponding declaration in xdc·spec can just contain the names for each argument.

Like configuration parameters, a function declaration inherited from a previously-specified interface can be overridden – typically, to alter or extend the set of default argument values or else to allow optional trailing arguments. Note, though, that since interfaces are entirely abstract (void of any "default" implementation), inheritance of functions is limited to their client-visible specification; ultimately, it is concrete modules (or their delegates) that bear responsibility for implementing all functions directly or indirectly declared in their specification. With xdc·spec support for *Design-By-Contract* forthcoming, overriding inherited function declarations becomes an essential technique for weakening pre-conditions and strengthening post-conditions specified previously through executable expressions.

function-declaration
    typed-function-declaration | untyped-function-declaration
typed-function-declaration
    function-modifiers
    typed-declaration **(** typed-arguments? [**, ...**]? **);**
function-modifier
    final? **override**? **metaonly**?
typed-arguments
    arg-declaration [**,**arg-declaration]*
arg-declaration
    typed-declaration [**=** initializer]?
untyped-function-declaration
    function *fxnName***(** untyped-arguments? **);**
untyped-arguments
    argName [**,***argName*]*

- If **`override`** is specified among the function-modifiers for a function, its name and type signature as well as its use of **`metaonly`** must *exactly* match that of the inherited declaration being overridden. Functions marked **`final`** cannot be overridden.
- The optional initializer within an arg-declaration must yield a value consistent with the typed-declaration, using the rules of assignment-compatibility defined by xdc·script for the meta-domain.
- All argument names – whether typed-arguments or untyped-arguments – must be uniquely named on a per-function basis.
- An untyped-function-declaration is implicitly modified **`final`** and **`metaonly`**.
- By convention, function and argument names are `camelCase` identifiers.

typed-declaration

Generic declarations of typed identifiers patterned after the familiar (and sometimes awkward) syntax of C lie at the heart of virtually all xdc·spec feature definitions. These declarations stipulate a type name, either built-in or previously-defined, followed by what is conventionally termed a *declarator* – the name of the feature per se, optionally adorned with other syntactic elements. As in C, use of the **`*`**, **`[]`**, and **`()`** operators within the declarator denotes new types such *pointer-to*`(`*`t`*`)`, *array-of*`(`*`t`*`)`, and *function-returning*`(`*`t`*`)`for some base type *`t`*; extra parentheses are typically used to bind the lower-precedent **`*`** operator to the declared name, especially when defining types of form *pointer-to*`(`*function-returning*`(`*`t`*`))`.

Beyond these familiar C constructs lifted from the target-domain – each given a corresponding meaning in the meta-domain – xdc·spec introduces additional base types as well as more specialized forms of the array: [1] the built-in type **`any`**, which subsumes all other types in the meta-domain; [2] the keywords **`Module`** or **`Instance`**, signifying an opaque type referencing a concrete module or instance object whose visible features are limited to those specified in the corresponding named unit; [3] the keyword **`length`** in conjunction with the **`[]`** operator, signifying the type *vector-of*`(`*`t`*`)` whose length can be altered in the meta-domain and retrieved in the target-domain; and [4] the keyword **`string`** in conjunction with the **`[]`** operator, signifying the type *map-into*`(`*`t`*`)`that effectively overlays direct access via string-valued keys on an underlying *vector-of*`(`*`t`*`)`.

typed-declaration
    [builtin-type | defined-type] declarator

builtin-type
    `standard-C-type` | *`extended-XDC-type`*

defined-type
    [qualified-unit-name `.`]? defined-type-name

defined-type-name
    `EnumName` | *`StructName`* | *`TypedefName`* | **`Instance`** | **`Module`**

declarator

        declared-name ?
    | * declarator
    | declarator [[*number* | **length** | **string**]?]
    | declarator (argument-types ? [...]?)
    | (declarator)

declared-name

        CONST_NAME
    | *EXTERN_NAME*
    | *TypedefName*
    | *argName*
    | *configName*
    | *fieldName*
    | *fxnName*

argument-types

    typed-declaration [,typed-declaration]*

- A builtin-type extends the familiar set of standard C types (`int`, `unsigned char`, etc.) with a set of capitalized equivalents (`Int`, `UChar`, etc.) as well as a handful of scalar types (`String`, `Ptr`, `Int32`, etc.) that promote further portability of C code. [Additional information on these extended types – both their definition in terms of standard C as well as their interpretation in the meta-domain – will be found elsewhere.]
- A defined-type identifies a previously defined type, either in the module or interface designated by a valid qualified-unit-name or else in the current unit.
- The declared-name at the heart of a declarator is only optional within arguments-types, typically used when declaring a type *pointer-to*(*function-returning*(*t*)). C reference grammars usually refer to this syntactic construct as an *abstract-declarator*.

initializer

An initializer is an expression that denotes either a scalar or aggregate value, and whose elementary terms are manifest constants of known types. While certainly C-like in form and substance, xdc·spec scalar initializers are in fact statically-evaluated using JavaScript semantics (upon which xdc·script is based). Building on this foundation, xdc·spec aggregate initializers adopt standard JavaScript notation for denoting object and array values.

initializer

    scalar-initializer | array-initializer | struct-initializer

```
scalar-initializer
        literal
      | defined-constant
      | unary-op scalar-initializer
      | scalar-initializer binary-op scalar-initializer
      | scalar-initializer ? scalar-initializer : scalar-initializer
      | (scalar initializer)
literal
    number | string | true | false | null |undefined
defined-constant
    [qualified-unit-name .]? defined-constant-name
defined-constant-name
    CONST_NAME | ENUMVAL_NAME
unary-op
    *|-|~|!
binary-op
    +|-|*|/|%|<<|>>|==|!=|<|<=|>|>=|&| | |^
array-initializer
    [ ] | [ initializer [,initializer]* [,]?]
struct-initializer
    { } | { field-initializer [,field-initializer]* [,]?}
field-initializer
    fieldName : initializer
```

- A literal `number` or `string` must conform to standard JavaScript, which also tracks standard C in this regard.
- A defined-constant identifies a previously defined constant, either in the module or interface designated by a valid qualified-unit-name or else in the current unit.
- The meaning and precedence of each unary-op and binary-op conforms to standard JavaScript, which likewise mirrors C. Note that JavaScript often overloads operators like **+** and **<** to accept strings as well as numbers.

//! @xdoc

Special comments embedded within a specification source file are available for processing by independent tools used (say) for generation of HTML-based manuals or for interactive package browsing. These documentation comments are identified by an extra leading “bang” character – **`//!`** for single-line comments, **`/*!`** for the block variety – and can be juxtaposed with most named elements in the specification. Whenever multiple comments of either variety are associated with an individual specification element, their bodies are effectively concatenated into a single *documentation block* comprising one or more lines of text.

Markup of the form **@*tag***, when present at the beginning of a line, further punctuates a documentation block into distinct sections comprising various styles of paragraphs. To avoid clutter in the source file due to excessive markup, most commentary can be written as “plain text” that follows some simple conventions to indicate (say) change-of-font or end-of-paragraph. Each documentation block generally comprises: [1] an untagged *summary* section, which is typically a “one-liner”; [2] an optional untagged *details* section, which may contain multiple paragraphs with additional information; and [3] a series of *tagged* sections, which further compartmentalize information about the associated specification element.

[Additional information on **@*xdoc*** – covering the current set of supported tags as well as textual hints – will be found elsewhere.]

```
//! @xdoc
    summary-section
    details-section?
    tagged-section*
summary-section
    comment-paragraph
details-section
    comment-paragraph+
tagged-section
    @sectTag(ident) comment-paragraph+
comment-paragraph
    [@p[(style)]?]? textLines*
```