orphan:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\proposals\archive\[swift-main][docs][proposals][archive]ProgramStructureAndCompilationModel.rst, line 5)

Unknown directive type "highlight".

```
.. highlight:: none
```

Swift Program Structure and Compilation Model

Warning

This is a very early design document discussing the features of a Swift build model and modules system. It should not be taken as a plan of record.

Commentary

The C spec only describes things up to translation unit granularity: no discussion of file system layout, build system, linking, runtime concepts of code (dynamic libraries, executables, plugins), dependence between parts of a program, Versioning + SDKs, human factors like management units, etc. It leaves all of this up to implementors to sort out, and we got what the unix world defined in the 60's and 70's with some minor stuff that could be shoehorned into the old unix toolchain model without too much trouble. C also doesn't help with resources (images etc), has a miserable incremental compilation model and many, many, other issues.

Swift should strive to make trivial programs really simple. Hello world should just be something like:

```
print("hello world")
```

while also acknowledging and strongly supporting the real world demands and requirements that library implementors (hey, that's us!) face every day. In particular, note how the language elements (described below) correspond directly to the business and management reality of the world:

Ownership Domain / Top Level Component: corresponds to a product that is shipped as a unit (Mac OS/X, iWork, Xcode), is a collection of frameworks/dylibs and resources. Only acyclic dependencies between different domains is allowed. There is some correlation in concept here to "umbrella headers" or "dyld shared cache" though it isn't exact.

Namespace: Organizational structure within a domain, similar to C++ or Java. Programmers can use or abuse them however they wish.

Subcomponent: corresponds to an individual team or management unit, is one dylib + optional resources. All contributing source files and resources live in one directory (with optional subdirs), and have a single "project file". Can contribute to multiple namespaces. The division of a domain into components is an implementation detail, not something externally visible as API. Can have cyclic dependencies between other components. Components roughly correspond to "xcode project" or "B&I project" granularity at Apple. Can rebuild a "debug version" of a subcomponent and drop it into an app without rebuilding the entire world.

Source File: Organizational unit within a component.

In the trivial hello world example, the source file gets implicitly dropped into a default component (since it doesn't have a component declaration). The default component has settings that corresponds to an executable. As the app grows and wants to start using sub-libraries, the author would have to know about components. This ensures a simple model for new people, because they don't need to know anything about components until they want to define a library and stable APIs.

We'll also eventually build tools to do things like:

- Inspect and maintain dependence graphs between components and subcomponents.
- Diff API [semantically, not "by symbol" like 'nm'] across versions of products
- Provide code migration tools, like "rewrite rules" to update clients that use obsoleted and removed API.
- Pure swift apps won't be able to use SPI (they just won't build), but mixed swift/C apps could (through the C parts, similar to using things like "extern int Z3fooi(int)" to access C++ mangled symbols from C today). It will be straight-forward to write a binary verifier that cross references the NM output with the manifest file of the components it legitimately depends on.
- Lots of other cool stuff I'm sure.

Anyway, that's the high-level thoughts and motivation, this is what I'm proposing:

Program structure

Programs and frameworks in swift consist of declarations (functions, variables, types) that are (optionally) defined in possibly nested namespaces, which are nested in a component, which are (optionally) split into subcomponents. Components can also have associated resources like images and plists, as well as code written in C/C++/ObjC.

A "**Top Level Component**" (also referred to as "an ownership domain") is a unit of code that is owned by a single organization and is updated (shipped to customers) as a whole. Examples of different top-level components are products like the swift standard

libraries, Mac OS/X, iOS, Xcode, iWork, and even small things like a theoretical third-party Perforce plugin to Xcode.

Components are explicitly declared, and these declarations can include:

- whether the component should be built into a dylib or executable, or is a subcomponent.
- the version of the component (which are used for "availability macros" etc)
- an explicit list of dependencies on other top-level components (whose dependence graph is required to be acyclic) optionally with specific versions: "I depend on swift standard libs 1.4 or later"
- a list of subcomponents that contribute to the component: "mac os consists of appkit, coredata, ..."
- a list of resource files and other stuff that makes up the framework
- A list of subdirectories to get source files out of (see filesystem layout below) if the component is more that one directory full of code.
- A list of any .c/.m/.cpp files that are built and linked into the component, along with build flags etc.

Top-Level Components define the top level of the namespace stack. This means everything in the swift libraries are "swift.array.xyz", everything in MacOS/X is "macosx.whatever". Thus you can't have naming conflicts across components.

Namespaces are for organization within a component, and are left up to the developer to handle however they want. They will work similarly to C++ namespaces and aren't described in detail here. For example, you could have a macosx.coredata namespace that coredata drops all its stuff into.

Components can optionally be broken into a set of "**Subcomponents**", which are organizational units within a top-level component. Subcomponents exist to support extremely large components that have multiple different teams contributing to a single large product. Subcomponents are purely an implementation detail of top-level components and have no runtime, naming/namespace, or other externally visible artifacts that persist once the entire domain is built. If version 1.0 of a domain is shipped, version 1.1 can completely reshuffle the internal subcomponent organization without affecting its published API or anything else a client can see.

Subcomponents are explicitly declared, and these declarations can include:

- The component they belong to.
- The set of other (optionally versioned) top-level components they depend on.
- The set of components (within the current top-level component) that this subcomponent depends on. This dependence is an acyclic dependence: "core data depends on foundation".
- A list of declarations they use within the current top-level component that aren't provided by the subcomponents they explicitly depend on. This is used to handle cyclic dependencies across subcomponents within an ownership domain: for example: "libsystem depends on libcompiler_rt", however, "libcompiler_rt depends on 'func abort();' in libsystem". This preserves the acyclic compilation order across components.
- A list of subdirectories to get source files out of (see filesystem layout below) if the component is more that one directory full of code.
- A list of any .c/.m/.cpp files that are linked into the component, with build flags.

Source Files and **Resources** make up a component. Swift source files can include:

- The component they belong to.
- Import declarations that affect their local scope lookups (similar to java import statements)
- A set of declarations of variables, functions, types etc.
- C and other language files are just another kind of resource to be built.

Declarations of variables, functions and types are the meat of the program, and populate source files. Declarations can be scoped to be externally exported from the component (aka API), internal to the component (aka SPI), local to a subcomponent (aka "visibility hidden", the default), or local to the file (aka static). Top-level components also have a simple runtime representation which is used to ensure that reflection only returns API and decls within the current ownership domain: "App's can't get at iOS SPI".

Executable expressions can also be included at file scope (outside other declarations). This global code is run at startup time (same as static constructors), eliminating the need for "main". This initialization code is correctly run bottom-up in the explicit dependence graph. Order of initialization between multiple cyclicly dependent files within a single component is not defined (and perhaps we can make it be an outright error).

File system layout and compiler UI

The filesystem layout of a component is a directory with at least one .swift file in it that has the same name as the directory. A common case is that the component is a single directory with a bunch of .swift files and resources in it. The "large component" case can break up its source files and resources into subdirectories.

Here is the minimal hello world example written as a proper app:

```
myapp/
myapp.swift
```

You'd compile it like this:

```
$ swift myapp
myapp compiled successfully!
```

or:

```
$ cd myapp
$ swift
myapp compiled successfully!
```

and it would produce this filesystem layout:

```
myapp/
myapp.swift
products/
myapp
myapp.manifest
buildcache/
<stuff>
```

Here is a moderately complicated example of a library:

```
mylib/
mylib.swift
a.swift
b.swift
UserManual.html
subdir/
c.swift
d.swift
e.png
```

mylib.swift tells the compiler about your sub directories, resources, how to process them, where to put them, etc. After compiling it you'd keep your source files and get:

```
mylib/
products/
mylib.dylib
mylib.manifest
e.png
docs/
UserManual.html
buildcache/
<more stuff>
```

Swift compiler command line is very simple: "swift mylib" is enough for most uses. For more complex use cases we'll support specifying paths to search for components (similar to clang -F or -L) etc. We'll also support a "clean" command that nukes buildcache/ and products/.

The BuildCache directory holds object files, dependence information and other stuff needed for incremental [re]builds within the component. The generated manifest file is used by the compiler when a client lib/app import mylib (it contains type information for all the stuff exported from mylib) but also at runtime by the runtime library (e.g. for reflection). It needs to be a fast-to-read but extensible format.

What the build system does, how it works

Assuming that we're starting with an empty build cache, the build system starts by parsing the mylib.swift file (the main file for the directory). This file contains the component declaration. If this is a subcomponent, the subcomponent declares which super-component it is in (in which case, the super-component info is loaded). In either case, the compiler verifies that all of the depended-on components are built, if not, it goes off and recursively builds them before handling this one: the component dependence graph is acyclic, and cycles are diagnosed here.

If this directory is a subcomponent (as opposed to a top-level component), the subcomponent declaration has already been read. If this subcomponent depends on any other components that are not up-to-date, those are recursively rebuilt. Explicit subcomponent dependencies are acyclic and cycles are diagnosed here. Now all depended-on top-level components and subcomponents are built.

Now the compiler parses each swift file into an AST. We'll keep the swift grammar carefully factored to keep types and values distinct, so it is possible to parse (but not fully typecheck) the files without first reading "all the headers they depend on". This is important because we want to allow arbitrary type and value cyclic dependencies between files in a component. As each file is parsed, the compiler resolves as many intra-file references as it can, and ends up with a list of (namespace qualified) types and values that are imported by the file that are not satisfied by other components. This is the list of things the file requires that some other files in the component provide.

Now that the compiler has the full set of dependence information between files in a component, it processes the files in strongly connected component (SCC) order processing an SCC of dependent files at a time. Given the entire SCC it is able to resolve values and types across the files (without needing prototypes) and complete type checking. Assuming type checking is successful (no errors) it generates code for each file in the SCC, emits a .o file for them, and emits some extra metadata to accelerate incremental builds. If there are .c files in the component, they are compiled to .o files now (they are also described in the component declaration).

Once all of the source files are compiled into .o files, they are linked into a final linked image (dylib or executable). At this point, a couple of other random things are done: 1) metadata is checked to ensure that any explicitly declared cyclic dependencies match the given and actual prototype. 2) resources are copied or processed into the product directory. 3) the explicit dependence graph is

verified, extraneous edges are warned about, missing edges are errors.

In terms of implementation, this should be relatively straight-forward, and is carefully layered to be memory efficient (e.g. only processing an SCC at a time instead of an entire component) as well as highly parallel for multicore machines. For incremental builds, we will have a huge win because the fine-grained dependence information between .o files is tracked and we know exactly what dependencies to rebuild if anything changes. The build cache will accelerate most of this, which will eventually be a hybrid on-disk/in-memory data structure.

The build system should be scalable enough for B&I to eventually do a "swift macos" and have it do a full incremental (and parallel) build of something the scale of Mac OS. Actually implementing this will obviously be a big project that can happen as the installed base of swift code grows.

SDKs

The manifest file generated as a build product describes (among other things) the full list of decls exported by the top-level component (which includes their type information, not just symbol names). This manifest file is used when a client builds against the component to type check the client and ensure that its references are resolved.

Because we have the version number as well as the full interface to the component available in a consumable format is that we can build an SDK generation tool. This tool would take manifest files for a set of releases (e.g. iOS 4.0, 4.0.1, 4.0.2, 4.1, 4.1.1, 4.2) and build a single SDK manifest which would have a mapping from symbol+type -> version list that indicates what the versions a given symbol are available in. This means that framework authors don't have to worry about availability macros etc, it just naturally falls out of the system.

This tool can also produce warnings/errors about cases where API is in version N but removed in version N+1, or when some declaration has an invalid change (e.g. an argument added or something else "fragile"). Blue sky idea: We could conceivable extend it so that the SDK manifest file contains rewrite rules for obsolete APIs that the compiler could automatically apply to upgrade user's source code.

Future optimization opportunities

The system has been carefully designed to allow fast builds at -O0 (including keeping cached dependence information and the compiler around in memory "across builds"), allowing a very incremental compilation model and allowing carefully limited/understood cyclic dependencies across components. However, we also care about really fast runtime performance (better than our current system), and we should be able to get that as well.

There are several different possibilities to look at in the future:

1. Components are a natural unit to do "link time" optimization. Since the entire thing is shipped as a unit, we know that it is safe to inline functions and analyze side effects within the bounds of the component. This current LTO model should scale to the component level, but we'd need new (more scalable/parallel and memory efficient) approaches to optimize across the entire mac os product. Processing components bottom-up within a large component allows efficient context sensitive (and summary-based) analyzes, like mod/ref, interprocedural constant prop, inlining, and nocapture propagation. I expect nocapture to be specifically important to get stuff on the stack instead of causing them to get promoted to the heap all the time.
2. The dyld shared cache can be seen as an optimization across components within the mac os top-level component. Though it has the capability to include third party and other dylibs, in practice it is rooted from a few key apps, so it doesn't get "everything" in macos and it isn't used for other stuff (like xcode). The proposed (but never implemented) "per-app shared cache" is a straight-forward extension if this were based on optimizing across components.
3. There are a bunch of optimizations to take advantage of known fragility levels for devirtualization, inlining, and other stuff that I'm not going to describe here. Generalization of DaveZ's positive/negative ivar/vtable idea.
4. The low level tools are already factored to be mostly object file format independent. There is no reason that we need to keep using actual macho .o files if it turns out to be inconvenient. We obviously must keep around macho executables and dylibs.