Physical Architecture

Introduction

Most projects will create documentation as to the logical architecture of the endeavour:

- which components compose the system
- how do they interrelate
- which interfaces exist between the components
- further decomposition, if applicable
- some form of interface specification

Such an architectural overview is independent of the implementation constraints of real-world systems, and even partially independent from the implementation language environment.
What is often overlooked is what I call the "physical architecture" of a project:

- what does a component look like
- what mandatory elements must be made available for any component
- what optional elements can be made available for any component
- what hierarchy can or must exist between interrelated components
- how deep a hierarchy is allowed
- how does a component map to a directory structure
- how does a component hierarchy map to a directory structure

Of course, such questions can only be answered in the context of a concrete realisation, and are therefore only of secondary importance when creating an overall architecture. However, such questions had better be explicitly answered before undertaking the implementation of such an abstract architecture.
What I will set out in this article is a set of guidelines as an example of such a physical architecture. The development of an arbitrary logical architecture has created a hierarchy of several components with their interfaces, and I will set out some guidelines for the implementation in C (usable for C++ too), on a Windows- or Unix-based platform.

How to map a hierarchy onto a file system

If we assume a hierarchical file system (like Windows or Linux), the most natural mapping of a hierarchical component structure onto such a file system is to use one directory per component or sub-component. The hierarchy can theoretically be arbitrarily deep. However, most file systems limit the possible length of pathnames, thereby placing an implicit limit on the depth of the hierarchy. In practice it is useful to limit the hierarchy depth to three levels in normal cases, to eight in exceptional cases. That way, having a project root directory path of some 40 characters long is still not a problem.
The names of the components can be arbitrarily chosen. A useful name should not exceed 10 characters and shall not contain any spaces (!). Limiting component names to the old DOS 8.3 naming constraint is no longer state-of-the-art.

Component interfaces to the external world

As the logical architecture defines its components, each component shall advertise at least one (possibly more!) interface to the external world, i.e. to components that are not sub-components.
Interfaces shall be contained in include files (**.h*).
The primary interface (for most components it will usually be the only one) shall be made available in a file called *<component_name>.h*, and be located in the component's directory. That way, the complete component shall be contained in its directory.
To whom should a component advertise its interface(s)? Normally, just to its siblings at the same hierarchy level. And in order to simplify (and possibly automate) finding a component's interface file, a second include file named *<component_name>_i.h* shall be created in a directory called *include* parallel to the component's directory. In that directory all interfaces of the sibling components shall be collected. The *_i* stands for 'interface'.

The interface file shall be used by all sibling components needing the interface. But in order to avoid maintaining every interface twice, the file *<component_name>_i.h* shall only contain a relative reference[1] to *<component_name>.h* like this:

```
#include "../<component_name>/<component_name>.h"
```

Some boilerplate logic like a file comment and proper include-guards shall then complete the file. The full interface shall only be contained in the *<component_name>.h*.
If additional interfaces are defined for a component, those interfaces shall also have two include files, *<component_name>.<subintf>.h* locally in the component's directory, and *<component_name>_i.<subintf>.h* in the *include* directory, containing an `include` directive for the interface proper. The filename part *<subintf>* indicates – abbreviated as appropriate – the additional interface. For instance: *output.diag.h*.

Interface implementation

The interfaces as specified need to be instantiated and implemented within the component. For that, each interface header *<component_name>[.<subintf>].h* is combined with a corresponding C file *<component_name>[.<subintf>].c*. Aside from one project-wide general include file, the first non-comment directive in these C files is to include their corresponding header file. In that way, it can be assured that the interface specification is complete, i.e. not depending on the inclusion of other foreign interface files.
This implies one important rule: **All header files shall include all interfaces they use by themselves**. This makes it irrelevant in which order interface files are included in a source file.
As a consequence, when someone includes a certain interface, he/she shall be able to use all facilities offered by that interface without worrying about other interfaces or include files. One pre-processor peculiarity is to be accommodated: Depending on the coding guidelines in use, if any, an interface may publish one or more macro definitions. Technically, such a macro definition is only expanded upon use. However, if in such a macro facilities from other interfaces are indicated, this other interface must be included in each file using that facility. In such a case, the component interface shall include the used interface's header file, even if it itself doesn't really need or use the interface. Even the interface's implementation file may not need to use the macros so offered. But in order for the user of the component to be able to use all facilities offered, also the macros, the so 'used' interface must be included in the component's interface header.

Internal component structure

A minimal component shall be composed of one header file (the interface) and of one C file (the implementation). A component can be bigger, though.
If the functional decomposition of a component does not warrant creating subcomponents on an architectural level, the component can be internally decomposed at a design level. But also in this case we need interfaces, both declarations and their implementation. File naming is not basically restricted, but in order to globally identify files, a 3 or 4 letter acronym shall be defined for each (sub-) component, and all component-internal files shall start with this acronym. Similar rules as for externally visible files shall apply for component-internal files: Each interface shall be fully contained in a header file; each header file shall have a corresponding C file containing the full implementation of the interface; each C file shall start with including its own interface header (except for one global header file).

File naming

The names of source files can be chosen as deemed appropriate. All relevant operating systems forbid identical names in the same directory, but identical names are allowed if the containing directory differs. However, in order to avoid problems in identifying files throughout a project, no two files in one project configuration shall be named identically.

---

[1] Please be aware that the compiler in use must be able to use the path of the including file as a first base for searching include files, as most current compilers can. If not, it might be possible to use the directory from where the compiler is called as a base; using absolute pathnames is strictly forbidden.

Component-spanning

According to the principle of encapsulation, in a component hierarchy not every component shall be able to use the (interface of) every other component. I call this principle "component-spanning", with reference to the "span of control" a manager has.
Component-spanning follows two easy rules:

1. Any component can use the interfaces of all its direct subcomponents;
2. All components inherit the component-span of their respective parent component

These two rules ensure that the constraints imposed by the (logical) architecture can be enforced by the physical architecture: If a build process (be it automatic or manual) starts at the root and moves toward the component under scrutiny, thereby collecting all subdirectories called *include* along the way, the paths so collected build the search path for compiling the respective component.
If a component needs access to another component's interface that is not in view, this is a sign that the architecture needs to be reviewed: Maybe an interface is incomplete, or the component hierarchy may be up for some rework.

Include-guards

Include-guards are necessary for all header files. `#pragma once` is a possibility, but not supported by all compilers, so to be sure, include-guards shall be provided for all header files.
Include-guards shall be unique within the project; otherwise they make no real sense.
I'd suggest using the (unique) filename as a basis for generating include-guards. At least the way to create include-guards shall be consistent within or even across projects.
As an example, for an include file named *PumpControl.h*:

```
#if !defined(_included_PumpControl_h_)
#define      _included_PumpControl_h_ 1
...
#endif /* _included_PumpControl_h_ */
```

Be aware that macro names starting with two underscores or with one underscore and a capital letter are reserved names.

External include-guards

This is a topic of some controversy. The idea is to improve the speed of compilation by specifying the include-guards from within a header file also outside around the include-directive. To expand on the sample above, suppose some component wants to use the interface of the `PumpControl` component, and therefore wants to include the corresponding header file:

```
#                          if !defined(_included_PumpControl_h_)
#include "PumpControl.h"
#                          endif /* _included_PumpControl_h_ */
```

This construct will include the intended header file, but only if it hasn't been included before. This functionality we have already with the include-guards of the previous section. The difference is the opening, reading and closing of an extra file. If the file has been included before, the include-directive is ignored, and the file isn't touched at all.
Don't underestimate the effects of such a simple construct: it can easily bring up to 30% improvement in compilation times, depending on other conventions and rules on placement of certain pre-processor directives.
The extra indentation between '`#`' and the `ifdef` / `endif` -pair is to visually enhance the include-directive proper. Without the indentation the effective filename of the header to be included would be shrouded and hard to discern.

Extern declarations

Most compilers hardly differentiate between prototypes of functions and external declarations of those functions. A small difference exists, however: An external declaration indicates that in the current compilation unit no definition for this function will be given, whereas a prototype is a form of forward declaration to enable the type-safe usage of a function before its definition has been processed.

Before using a function at least a prototype declaration shall have been processed, to avoid type inconsistencies between formal parameters and actual parameters, or formal return type and actual return type.
An example:

```
/* External declaration */
extern
tErrorCode PMP_StopPump(void);

/* Prototype declaration */
tErrorCode PMP_StopPump(void);
```

The interface header is intended for use by other components. This means that for all functions belonging to the interface an external declaration shall be made available. But when we compile the component itself, the same include file shall provide prototype declarations. For this to work without having to maintain two almost identical declarations, we shall introduce a macro called `_local_export_`. This macro can have one of two values: If we need a prototype, the macro shall expand into an empty string (whitespace), if we need an external declaration, the macro shall expand into the keyword `extern`.
We can achieve this in two ways:

- Either we define `_local_export_` as `extern` in the interface header before including the local, effective interface header from the component's directory, and we un-define it again after the include-directive;
- Or we require that each C file identifies itself to all included header files by defining a filename-based macro like `_cfile_PumpControl_h_`, which is then checked within the header file, and `_local_export_` is set accordingly.

The first solution works fine when the physical layout is consistently kept in sync with all recommendations contained in this document. The second solution is more locally contained, and not quite as dependent on the physical structure described in this document. Therefore I'd suggest using the second solution. Refer to the final section in this document where I provide a small set of sample files based on the partial samples presented in this document.

Samples

The indentations in the samples do not reflect the indentation I would suggest using, because of the limited space available.

Implementation file

```
/**
 * C-file comment PumpControl.c
 */
#define _cfile_PumpControl_h_ 1
#                     if !defined(_included_PumpControl_h_)
#include "PumpControl.h"
#                     endif /* _included_PumpControl_h_ */

#                     if !defined(_included_OtherComponent3_i_h_)
#include "OtherComponent3_i.h"
#                     endif /* _included_OtherComponent3_i_h_ */
#                     if !defined(_included_OtherComponent4_i_h_)
#include "OtherComponent4_i.h"
#                     endif /* _included_OtherComponent4_i_h_ */

/**
 * Function comment PMP_StopPump
 */
tErrorCode PMP_StopPump(void)
{
  HW_Stop_Pump();
  return (E_OK);
}
```

Interface file

```
/**
 * H-file comment PumpControl.h
 */
#if !defined(_included_PumpControl_h_)
#define      _included_PumpControl_h_ 1

/* Put all include directives here */
#                     ifndef _included_OtherComponent1_i_h_
#include "OtherComponent1_i.h"
#                     endif /* _included_OtherComponent1_i_h_ */
#                     ifndef _included_OtherComponent2_i_h_
#include "OtherComponent2_i.h"
#                     endif /* _included_OtherComponent2_i_h_ */

/* From here onwards, no include directives shall be used! */
/* Differentiate extern declarations and prototypes */
#if defined(_cfile_PumpControl_h_)
#  define _local_export_
#else
#  define _local_export_ extern
#endif /* _cfile_PumpControl_h_ */

_local_export_
tErrorCode PMP_StopPump(void);

/* Make sure _local_export_ is not carried over
 * into the next include */
#undef _local_export_

#endif /* _included_PumpControl_h_ */
```

Interface stub

```
/**
 * H-file comment PumpControl_i.h
 */
#if !defined(_included_PumpControl_i_h_)
#define      _included_PumpControl_i_h_ 1

#                          if !defined(_included_PumpControl_h_)
#include "../PumpControl/PumpControl.h"
#                          endif /* _included_PumpControl_h_ */

#endif /* _included_PumpControl_i_h_ */
```