GNU PDF Library Hackers Guide

Updated for version trunk.

Free Software Foundation

This is the *GNU PDF Library Hackers Guide*, updated for **libgnupdf** version **trunk**.

Copyright © 2008, 2011 Free Software Foundation, Inc.

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Table of Contents

1 Information for Newcomers 1

2 The build system 3
2.1 Third-party m4 macros 3

3 Development procedures 4

4 Tasks management 7
4.1 The tasks pool 7
4.2 Task originators 9
4.3 Task types 9
4.4 Tasks workflow 9
4.5 The flyspray task manager 11

5 Coding Conventions 12
5.1 File Headers 12
5.2 Inclusion of Header Files 12
5.3 Testing for Preprocessor Symbols 13
5.4 Spaces vs. Tabs 13
5.5 Naming Functions 13
5.5.1 Public functions in a module 13
5.5.2 Private functions in a module 14
5.5.3 Platform specific functions 14
5.6 Abstract Data Types 14
5.6.1 Implementation Files For ADTs 14
5.6.2 Data Structures For ADTs 15
5.6.3 Access Functions For ADTs 15
5.6.4 Opaque Pointers 15
5.7 The layer header files 17

6 Writing Documentation 18
6.1 Generating pictures with ditaa 18

7 Sending Patches 19
7.1 Documenting Your Changes 19
7.2 Generating a Bazaar Merge Directive 19
7.3 Syntax Check 19
7.3.1 Skipping syntax tests 20
7.4 Patch Safety Dispatcher 20
7.5 Sending your Patch 21

8 Testing the library 22
- 8.1 The test specification document 23
- 8.2 Unit testing 23
  - 8.2.1 Designing unit tests 25
  - 8.2.2 Test files 25
  - 8.2.3 Test suite files 26
  - 8.2.4 The runtests program 26
  - 8.2.5 Running the unit tests 27
  - 8.2.6 Using gdb to debug check tests 27
- 8.3 Test Data Files 29
- 8.4 The tortutils Library 29

9 Cross-compiling libgnupdf for Windows under GNU/Linux 30

10 Updating the AUTHORS file 33

1 Information for Newcomers

First of all, many thanks for your interest in collaborate in the development! Here you will find some useful information to get started.

Getting a copy of the sources

The first step to perform is to get a local copy of the development sources. Just install a GNU Bazaar client and issue the following commands.

To get the sources of the GNU PDF Library from the trunk:

```
$ bzr branch bzr://bzr.savannah.gnu.org/pdf/libgnupdf/trunk
```

Getting familiar with GNU Bazaar

We are using the GNU Bazaar[1] version control system. Before to be able to contribute code to this project you should get familiar to the usage of Bazaar.

The GNU Bazaar User Guide[2] is very good. In the Bazaar site (http://www.bazaar-vcs.org) you will also find useful tips for people coming from other VCSs such as CVS or subversion.

The "Bzr For Emacs Devs" page in the Emacs wiki provides detailed information on the workflow used in the Emacs project. Even if we don't follow that exact workflow, most of the guidelines there apply in the GNU PDF development. You can find the document at `http://www.emacswiki.org/emacs/BzrForEmacsDevs`.

Subscribing to the development mailing list

GNU PDF developers communicate using the `pdf-devel@gnu.org` mailing list. Most of the work in the development is discussed there, so you definitely want to subscribe to it if you are going to write some code.

Goto the pdf-devel mailman web interface in `http://lists.gnu.org/mailman/listinfo/pdf-dev` and set up your subscription. Then, send a first email introducing yourself.

There is another mailing list `pdf-tasks@gnu.org` that is used by the Flyspray installation to send change notifications of tasks. You can subscribe to this mailing list at `http://lists.gnu.org/mailman/listinfo/pdf-tasks`.

Getting familiar with Savannah

Savannah is the central point of development for GNU software. The GNU PDF project has a savannah project in `http://savannah.gnu.org/projects/pdf`

Getting familiar with the library

Take a look to the design of the library and the source code. The documents to look at are the "GNU PDF Library Reference Guide" (available in doc/gnupdf.texi) and the "GNU PDF Library Architecture Manual" (available in doc/gnupdf-arch.texi).

There is useful information in the '`README-dev`' file located at the root of the source distribution. In particular the dependencies that you need to have installed in your system are detailed there.

[1] http://bazaar-vcs.org

[2] http://doc.bazaar-vcs.org/bzr.dev/en/user-guide/index.html

If you have doubts about any aspect of the library or the implementation, just ask in pdf-devel.

Getting familiar with the GNU standards

The GNU PDF Library is a GNU package. That means we should follow the GNU Standards to ensure quality and compatibility with other parts of the GNU Operating System.

You should be familiar with the GNU Coding Standards before to write code for the GNU PDF library. You can read it online in `http://www.gnu.org/prep/standards`.

Getting familiar with our coding conventions

Read our Hackers Guide (this very document) for several conventions we use when writing source code.

Taking a task to work on

Please read the documentation on tasks management in Chapter 4 [Tasks management], page 7.

If you want to work in a NEXT task, please be sure to state your interest in the pdf-devel mailing list before. The developers in the pdf-devel mailing list can guide you towards the more convenient task for you to take and can give valuable advice about how to implement it.

Signing papers

The GNU PDF codebase is copyrighted by the Free Software Foundation. This is a way to strength the GPL. That means that we need you to sign papers before to be able to integrate your code in the distribution. There are several ways to get this done. Please ask in pdf-devel for the details.

Sending patches for inclusion

If you have written a patch you want to be included in GNU PDF, please send it to the pdf-devel mailing list. Your patch will be then discussed and maybe accepted.

2 The build system

The build system of the GNU PDF software uses the GNU build utilities (otherwise known as the GNU Autotools). This chapter contains some guidelines to apply when incorporating changes into the codebase.

2.1 Third-party m4 macros

Sometimes it is useful to use third-party m4 macros provided by some build dependency (such as libgcrypt that provides an `AM_PATH_LIBGCRYPT` macro).

In that situation we are introducing a dependency in *bootstrap* time, and it is not desirable: the dependencies should be checked in *configure* time.

Any third-party m4 file should be copied in '`libgnupdf/m4/`' and put under version control. In that way we avoid the dependency in *bootstrap* time.

3 Development procedures

The GNU PDF Library is composed by several layers. Each layer is in turn composed by several modules. The combination of the modules provide the functionality to layers. See ⟨undefined⟩ [Top], page ⟨undefined⟩.

The development of the library involves the design of the overall architecture (determination of the layers that composes the library), the development of each layer and the design and implementation of system tests.

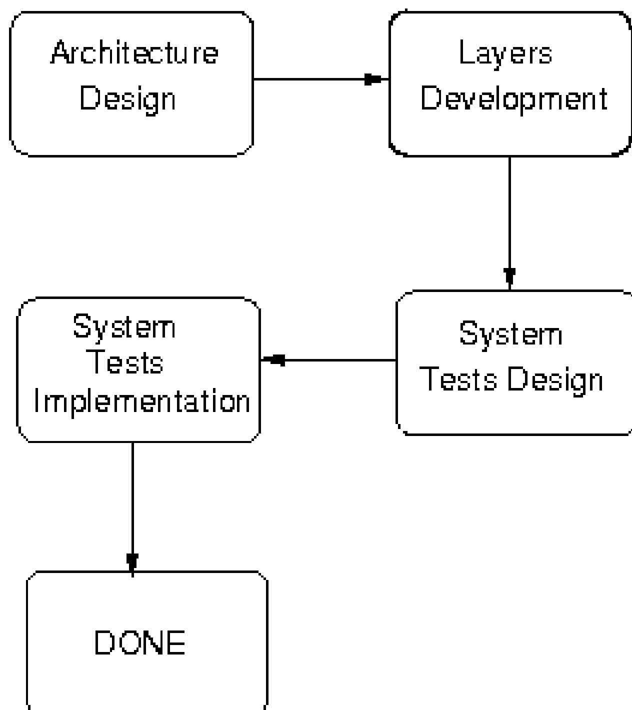

Figure 3.1: Library Development Procedure

The development of each library layer involves the design of the overall architecture of the layer (determination of the modules that composes the layer), the development of each module and the design and implementation of subsystem tests.

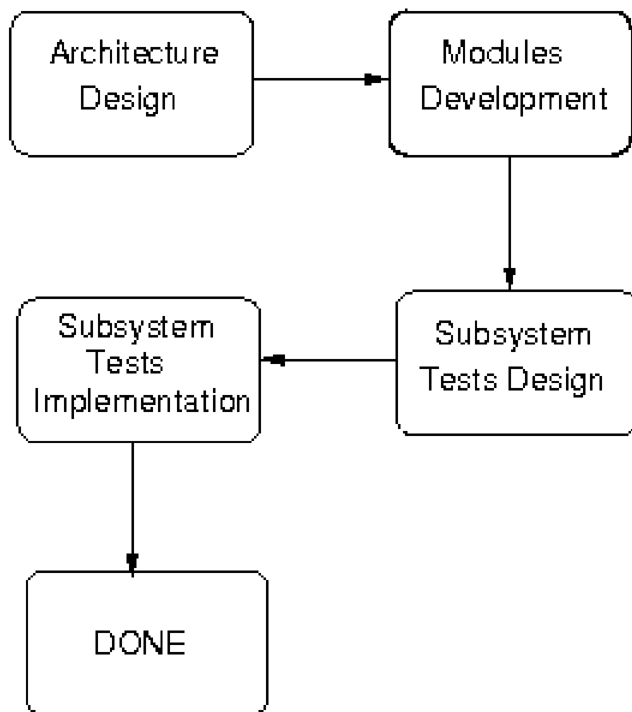

Figure 3.2: Layer Development Procedure

The development of each module involves the design of the API offered by the module and the architectural details, the definition of development tasks needed to complete the implementation of the module and the design and implementation of unit tests.

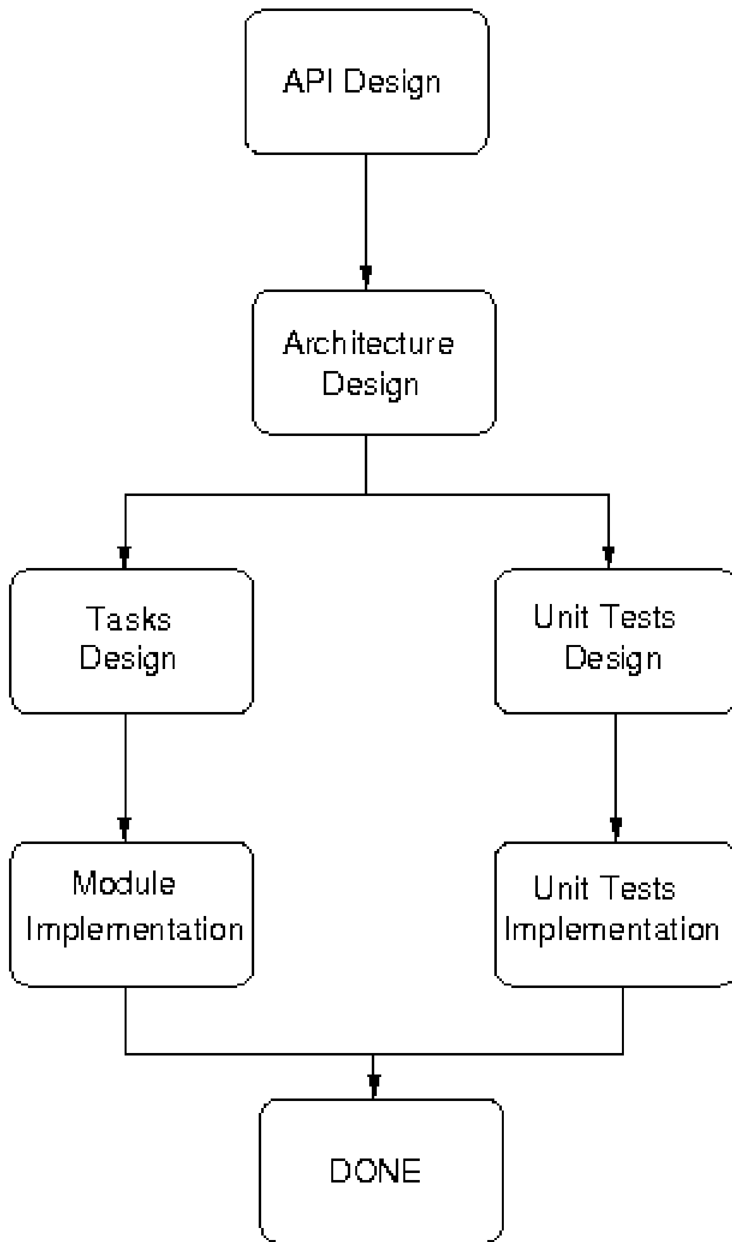

Figure 3.3: Module Development Procedure

4 Tasks management

This chapter contains information about the tasks management we use in the development of the GNU PDF Library.

4.1 The tasks pool

The real tasks being worked out in the project are contained in what we call the tasks pool.

The tasks pool contain a set of tasks that should be performed out by some agent. The pool do not contain all the tasks for the project: there is an approximation in the project plan. Instead there is a constant flow of tasks being introduced in the pool by originators and being consumed out by the developers. That means that when a task is performed by a developer it is pulled out of the pool and archived.

An archived task can be reactivated and inserted again in the tasks pool.

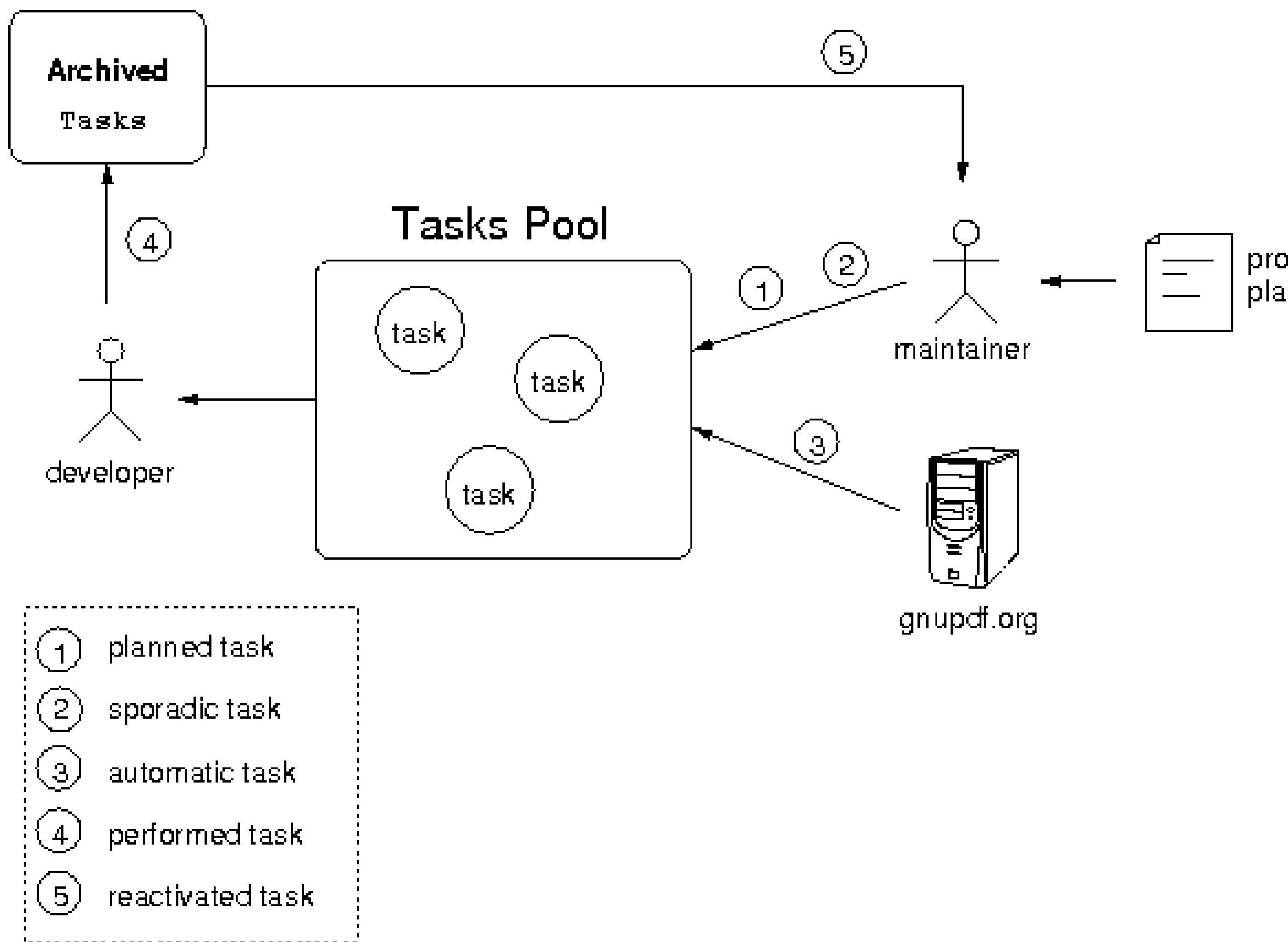

Figure 4.1: The tasks pool

A task is attributed with a priority while it is into the tasks pool. The priority is a number between 1 (less priority) to 9 (higher priority). Some priority levels have names:

- 1 - Later
- 2
- 3 - Low
- 4
- 5 - Normal
- 6
- 7 - High
- 8
- 9 - Immediate

> **Note:** The maintainer may change the priority of a task to avoid it to starve.

> **Note:** A reactivated task gets a new priority number when it is reinserted into the tasks pool.

4.2 Task originators

There are several possible originators of tasks:

The maintainer
The maintainer is the responsible of maintaining the tasks pool, introducing new tasks when needed and setting the priorities. The source of the tasks is either the project plan, bug reports or reactivated tasks.

Cron jobs running in gnupdf.org
There are several cron jobs running in gnupdf.org that perform checks on a nightly build of the sources and executes the unit tests. These jobs can automatically generate new tasks to fix some problem (a high cyclomatic complexity in a module or a fail in a test case).

4.3 Task types

We distinguish between some types of tasks. Each task type has an initial priority defined:

task type	**originator**	**source**	**initial priority**
planned task	maintainer	project plan	5 (Normal)
sporadic task	maintainer	bug report	4
automatic task	gnupdf.org	test fail or bad software metric	6

4.4 Tasks workflow

The following diagram depicts the possible states for a task and the allowed transitions between states:

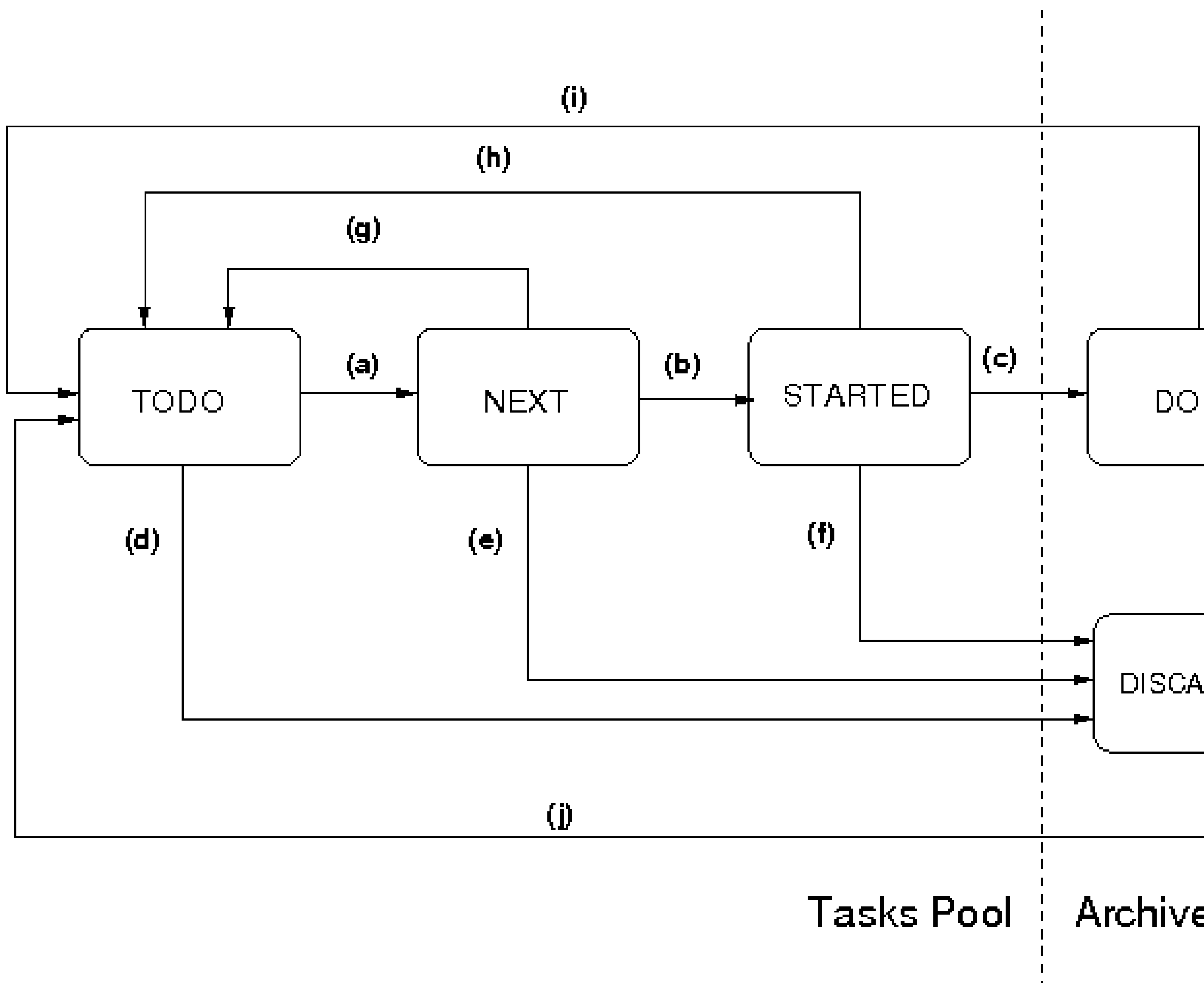

Figure 4.2: Tasks workflow

TODO The task is into the tasks pool but it not ready to be performed.

NEXT The task is into the tasks pool and it is ready to be performed by a developer.

STARTED
The task has been started by a developer but it is not finished.

DONE The task is archived and succesfully performed.

DISCARDED
The task is archived but it has been discarded.

Note that a task may also be assigned to someone or unassigned:

TODO and unassigned
Dependency issues, no one in charge.

TODO and assigned
Dependency issues and someone showed interest.

NEXT and unassigned
Can be started, nobody interested.

NEXT and assigned
Someone is interested but can't work on it for personal issues.

STARTED and assigned
Someone is working on the task (no matter the time it takes)

4.5 The flyspray task manager

We use a flyspray task tracker installed in `http://gnupdf.org/flyspray` in order to implement the tasks pool.

Each flyspray task has the following attributes:

Type The type of the task:
- planned
- sporadic
- automatic

Category he context of the task. It may be the name of a library layer (such as base layer), the name of a library module or a more general context (such as build system).

Priority The priority of the task.

Status The status of the task:
- TODO
- NEXT
- STARTED
- DONE
- DISCARDED.

Summary A summary of the task.

5 Coding Conventions

Like in any other GNU package, the code in the GNU PDF Library follows the coding conventions documented in the GNU Coding Standards.

In this section we complement the guidelines of the GHM with some specific conventions that we follow in the development of the library. It is quite important to follow these guidelines to maintain a good level of coherence in the codebase.

5.1 File Headers

The standard file header to be used in any source file in the library is the following:

```
/* -*- mode: C -*-
 *
 *       File:         FILE_NAME
 *       Date:         CREATION_TIME
 *
 *       GNU PDF Project - SHORT_DESCRIPTION
 *
 */
```

The entries in the template are:

FILE_NAME
: The basename of the file.

CREATION_TIME
: A time stamp string in the format:

```
Fri Feb 22 21:05:05 2008
```

Note that if you are writing your code using Emacs then you will get the appropriate creation date running the `current-time-string` elisp command. If you are using the `gnupdf-c-file-header` skeleton template then you will get the creation date in template-expansion time.

SHORT_DESCRIPTION
: A one-sentence brief description of the contents of the file. This description should not exceed one physical line of text.

5.2 Inclusion of Header Files

The order of the `#include` directives in any '`.c`' file shall be:

```
/*
 * ... file header ...
 */

#include <config.h>

#include <system-file-1.h>
#include <system-file-2.h>
...
```

```
#include <system-file-N.h>

#include <lib-file-1.h>
#include <lib-file-2.h>
...
#include <lib-file-N.h>
```

Note the empty lines separating the several blocks.

In particular:

- Always include 'config.h', and do it in the first position.
- Never use the "foo.h" kind of inclusion. Use always <foo.h>.

If some headers shall be included conditionally depending on the host platform, the macros PDF_HOST_* defined in 'config.h' can be used. Those macros are set using autoheader.

5.3 Testing for Preprocessor Symbols

Please don't use parentheses like this while testing for pre-processor symbols:

```
#if defined (FOO) && defined (BAR) || !defined (BAZ)
  do something
#endif
```

Instead, write something like:

```
#if defined FOO && defined BAR || !defined BAZ
  do something
#endif
```

5.4 Spaces vs. Tabs

It is preferable to use blank characters instead of tabs to indent the source code: the interpretation of the actual width of a tab is up to the viewer program.

Please use blank characters when writing code to be included in GNU PDF software.

If you use Emacs you can tell it to insert spaces instead of tabs including:

```
(setq-default indent-tabs-mode nil)
```

in your '.emacs'.

If you use GNU indent to indent your sources you can use the --nut option:

```
$ indent --nut [rest-of-parameters] [source-files]
```

5.5 Naming Functions

5.5.1 Public functions in a module

All the public functions inside a module should use the following name convention:

```
pdf_MODULE-NAME_...
```

where *module-name* is the canonical name of the module (e.g. alloc or text).

Some modules are composed by more than one compilation unit. In that case the public functions should follow the following name convention:

```
pdf_MODULE-NAME_PART-NAME_...
```

where *part-name* is the canonical name of that part of the module implementation (e.g. `pdf_stm_filter_...` where `filter` is the part name.

5.5.2 Private functions in a module

The private ("static") functions used in a module implementation should follow the same naming conventions as the public ones.

5.5.3 Platform specific functions

The names for functions (both public and private) intended to be used if compiling for a specific platform should use the following name convention:

```
pdf_MODULE-NAME[_PART-NAME]_PLATFORM_...
```

where `platform` is the canonical name for the target platform:

`gnu` For GNU systems.

`posix` For POSIX systems.

`win32` For Windows systems.

`macos` For Macos X systems.

5.6 Abstract Data Types

The GNU PDF Library codebase is written using the C programming language. C does not support the notion of *object* as used in object-oriented programming.

Instead of objects we are using a kind of data-control abstraction called *abstract data types*. This abstraction provides high encapsulation of the implementation details of the data types and thus allow the definition of *opaque* types.

An ADT is composed by:

- A *data structure* containing the private data that characterizes each instance of the ADT.
- A set of *access functions* that implement actions on the ADT.

5.6.1 Implementation Files For ADTs

Each Abstract Data Type shall be implemented in source files following this naming convention:

```
pdf-FOO-*.[ch]
```

where *FOO* is the name of the ADT; for example, '`pdf-text-context.c`'.

A general header file for the ADT should always be present and should be named after:

```
pdf-FOO.h
```

where *FOO* is again the name of the ADT; for example, '`pdf-text.h`'.

5.6.2 Data Structures For ADTs

There are two different approaches that shall be used to define the data structures containing the private data for an ADT:

A pointer to a structure

In this case a C structure should be defined to hold the private data:

```
/* Definition of the pdf_foo_t ADT */
struct pdf_foo_s
{
   int data_a;
   int data_b;
};
```

and then a typedef that defines `pdf_foo_t` as a pointer to that structure:

```
typedef struct pdf_foo_s *pdf_foo_t;
```

A structure

In this case a C structure (not a pointer to it) is used to represent the ADT:

```
typedef struct pdf_foo_s pdf_foo_t;
```

This alternative is indicated in the case where the private data of the ADT is small, allowing the developer to allocate instances of the ADT in the stack and thus avoiding fragmentation of the heap.

Note that both alternatives allow to copy a **reference** using the C assignation operator, like in:

```
reference_to_adt_instance1 = adt_instance1;
```

5.6.3 Access Functions For ADTs

Every access function implemented by an ADT should have a prototype conformant to the following convention:

```
RETURN_TYPE pdf_FOO_* (pdf_FOO_t adt, args...)
```

where *FOO* is the name of the ADT.

The following standard functions shall be defined:

`pdf_status_t pdf_FOO_new (args..., pdf_FOO_t *adt)`

This is the function used to create a new instance of the ADT. The last parameter of the function should be a pointer to a `pdf_FOO_t` value. The returned status value should indicate the state of the operation.

`pdf_FOO_destroy (pdf_FOO_t adt)`

This is the function used to destroy an instance of the ADT. The memory occupied by the ADT data structure is freed.

5.6.4 Opaque Pointers

One way to achieve high encapsulation for ADTs is to publish “opaque pointers” in the public header files. The usage of opaque pointers also improves binary compatibility.[1].

[1] this technique is also known as the PIMPL idiom. See http://en.wikipedia.org/wiki/Pimpl_idiom

The mechanism is quite simple. Suppose that we want to publish an opaque ADT to the user: `pdf_foo_t`. A first approximation would be to mark the full definition of `pdf_foo_t` as public in the header file, like:

```
/* -*- mode: C -*-
 *
 *       File:         pdf-foo.h
 ...
 */

...

/* BEGIN PUBLIC */

struct pdf_foo_s
{
   int a;
   int b;
};

typedef struct pdf_foo_s *pdf_foo_t;

/* END PUBLIC */

...
```

Despite the user is not supposed to access the internal structure of `pdf_foo_t`, she **can** actually do it using the exported structure `struct pdf_foo_s`.

An additional problem is that the user can allocate `pdf_foo_s` structs in the stack, and thus the binary compatibility would be break if the binary links with a more recent version of the library exporting more fields (like a third integer `c`).

The solution to both problems is to export a "opaque pointer": we simply do not export the details about the structure:

```
/* -*- mode: C -*-
 *
 *       File:         pdf-foo.h
 ...
 */

...

/* BEGIN PUBLIC */

typedef struct pdf_foo_s *pdf_foo_t;

/* END PUBLIC */

struct pdf_foo_s
```

```
{
   int a;
   int b;
};

...
```

5.7 The layer header files

Each layer in the library provides a header file that gathers the public headers of its modules. The header is named after the layer. `pdf-LAYER.h` would contain:

```
#include <pdf-MODULE1.h>
#include <pdf-MODULE2.h>
...
```

`pdf-MODULEn.h` being the header files of the modules composing the layer having `BEGIN_PUBLIC`, `END_PUBLIC` sections.

The rules regarding those header files are:

- Never use `pdf-LAYER.h` in a module of the same layer.
- Always use `pdf-LAYER.h` in modules of another layer.

The second rule avoids the propagation of problems when we make changes in the module structure of a given layer. As long as it is stated that the object layer provides `pdf_obj_t`, for example, it does not matter which specific module provides it.

For example, if we wanted to use the tokeniser `pdf_token_read_t` in the object layer, we would include `pdf-base.h` instead of `pdf-token-read.h`.

6 Writing Documentation

This chapter contains some useful information on writing documentation in the GNU PDF project.

6.1 Generating pictures with ditaa

When generating pictures with ditaa, please make sure to not use implicit shape separation (`-E`) nor shadows (`-S`).

So, for example:

```
$ ditaa -S -E [YOUR-OPTIONS] figure.txt
```

Don't forget to regenerate both '`png`' and '`eps`' image files after an update to a picture.

7 Sending Patches

This chapter contains some useful information to send patches to be integrated in the trunk.

7.1 Documenting Your Changes

Everyone loves writing documentation! :D

Please update the dates in the copyright notices in each of the files you have modified, if needed.

Please update the file ChangeLog with a summary of which files have been changed, along with what the changes were for.

If your change includes new files please update the '`MANIFEST.wiki`' in the directory containing the new files.

If your change is not trivial please take a look to the appropriate architecture page in the wiki. Maybe it is needed to update the page with more information?

7.2 Generating a Bazaar Merge Directive

The Bazaar version control system supports the notion of *merge directives*. A merge directive is a kind of "superpatch" that contain an ascii-encoded binary block describing the patch (changes to file contents, addition of new files, etc) and a preview that is much like a regular diff.

A merge directive can be merged into a given branch much like any other branch.

To create a merge directive out of your bazaar branch just type the following command:

```
$ bzr send -o my-patch
```

Then send the file '`my-patch`' in an email to pdf-devel@gnu.org in order to be reviewed by the development team.

Note that you dont need to specify extra parameters to the `bzr send` command: it will use the appropriate format for the patch by default (unidiff).

7.3 Syntax Check

The maintainer-makefile gnulib module provides some more make targets, useful for the maintainership of the package.

One of the targets is 'syntax-check'. It performs a check of common pitfalls on the source code and GCS conformance.

Please do a make syntax-check before to send a patch, or alternatively use the Patch Safety Dispatcher (see the next section).

Additionally, if you created more tests under 'torture/', please make sure the new test headers are correct by running:

```
$ perl ../build-aux/generate-tsd.pl | grep "BAD FORMAT"
```

If the `BAD FORMAT` mark appears in the output it means that some of the headers were not properly processed by the '`generate-tsd.pl`' script.

7.3.1 Skipping syntax tests

Sometimes it is not desirable to run an specific test in an specific source file. Some typical situations are:

- A false positive of the test.
- A positive on code that we are not maintaining (for example, any source file in the '`lib/`' directory).

In order to disable the execution of a syntax check in an specific file, the name of the source file should be added to the file '`.x-RULE`', where '`RULE`' is the name of the syntax check. An example is the file '`.x-sc_avoid_if_before_free`', that affects the syntax check implemented in '`build-aux/sc_avoid_if_before_free`'.

Note that the '`.x-RULE`' files should not contain empty lines.

In order to disable the execution of a syntax check for any file, just add the name of the syntax check to the `local-checks-to-skip` variable in '`cfg.mk`'.

7.4 Patch Safety Dispatcher

Before sending a patch to the list to be included in the trunk you can run the patch safety dispatcher, which is a script that runs a few more scripts, like the syntax check mentioned in this chapter.

In fact, the Patch Safety Dispatcher is a bzr plugin that is run before a commit is applied to your working copy. In order to execute it you need to tell bzr where the plugin is located. There are two ways to do it:

1. Copy the script located in "prmgt/patch-safety-dispatcher.py" at the projects root directory to your bazaar plugins directory "~/.bazaar/plugins".

2. Add the "prmgt" directory to the BZR_PLUGIN_PATH variable. For example, doing "export BZR_PLUGIN_PATH=/your/path/to/libgnupdf/prmgt" (alternatively you can add it to your ~/.bashrc).

After telling bzr where your plugins are, you can test it doing: "bzr hooks" (from the projects root directory) . You should find it in the list as "Patch safety scripts hook" in the pre_commit section.

That's all. Now when you do a "bzr commit" a small report will tell if your patch is correct in terms of the QA scripts we run daily. If it is the commit will be applied, otherwise it won't.

Note that the current hook only works with Bazaar 1.5 or later. To make it work on the older bazaar, you just need to replace this line:

```
branch.Branch.hooks.install_named_hook('pre_commit', pre_commit_hook,
                                       'Patch safety scripts hook')
```

with these lines:

```
branch.Branch.hooks.install_hook('pre_commit', pre_commit_hook)
branch.Branch.hooks.name_hook(pre_commit_hook, 'Patch safety scripts hook')
```

NOTE: Make sure you run "bzr commit" from your working copy root directory. Bazaar will fail with some error or don't even run the script otherwise. Until now we have no solution for this problem.

7.5 Sending your Patch

Before to send the patch to be considered for merging, please use the following checkpoints list to make sure that everything is in place:

- Is the patch including a ChangeLog entry?
- Is the patch including an update to the `AUTHORS` file documenting any new modified/created file by you?
- What about the copyright headers? Did you copy the header from other file and therefore it is needed to change the copyright years?

If everything is ok please send the patch to `pdf-devel@gnu.org`. Note that it is preferable to include the merge directive in the message body than to send it as an attachment.

8 Testing the library

We are following a bottom-up testing strategy. The verification of the library is performed in the following steps:

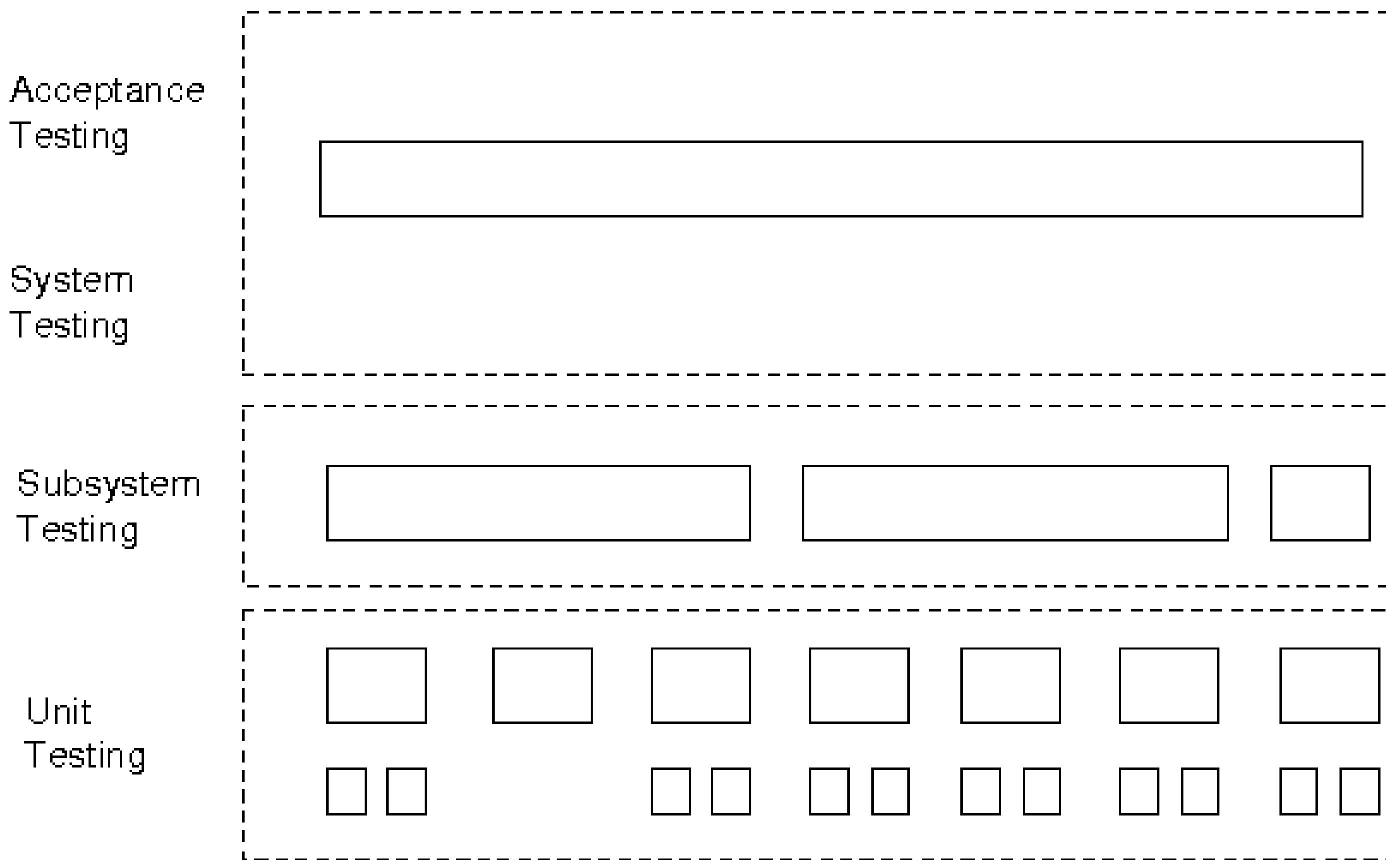

Figure 8.1: Test types

1. **Unit testing** is performed in order to verify the low-level modules of the library.
2. **Subsystem testing** is performed in order to verify the combination of several subsystems. i.e. to test each library layer.
3. **System testing** is performed in order to verify the whole system. i.e. the GNU PDF Library.

We use check[1] to implement our testing infrastructure. Please read the check manual in order to become familiar with its concepts such as test suite, test case, etc.

At some point we wrote a simple replacement for libcheck in Windows systems: nocheck (‘`torture/unit/nocheck`’). The usage of the replacement is now deprecated, since the latest version of libcheck should support windows.

[1] A testing framework for C. See http://check.sf.net

8.1 The test specification document

The tests for the GNU PDF Library are documented in a document called "Test Specification Document" or TSD. It is available as a texinfo file in the sources distribution in the 'doc/gnupdf-tsd.texi' file.

There is an online version of the TSD in http://www.gnupdf.org/manuals/gnupdf-tsd.html. It is automatically updated daily from the bazaar trunk.

8.2 Unit testing

We organize unit tests using the following structure:

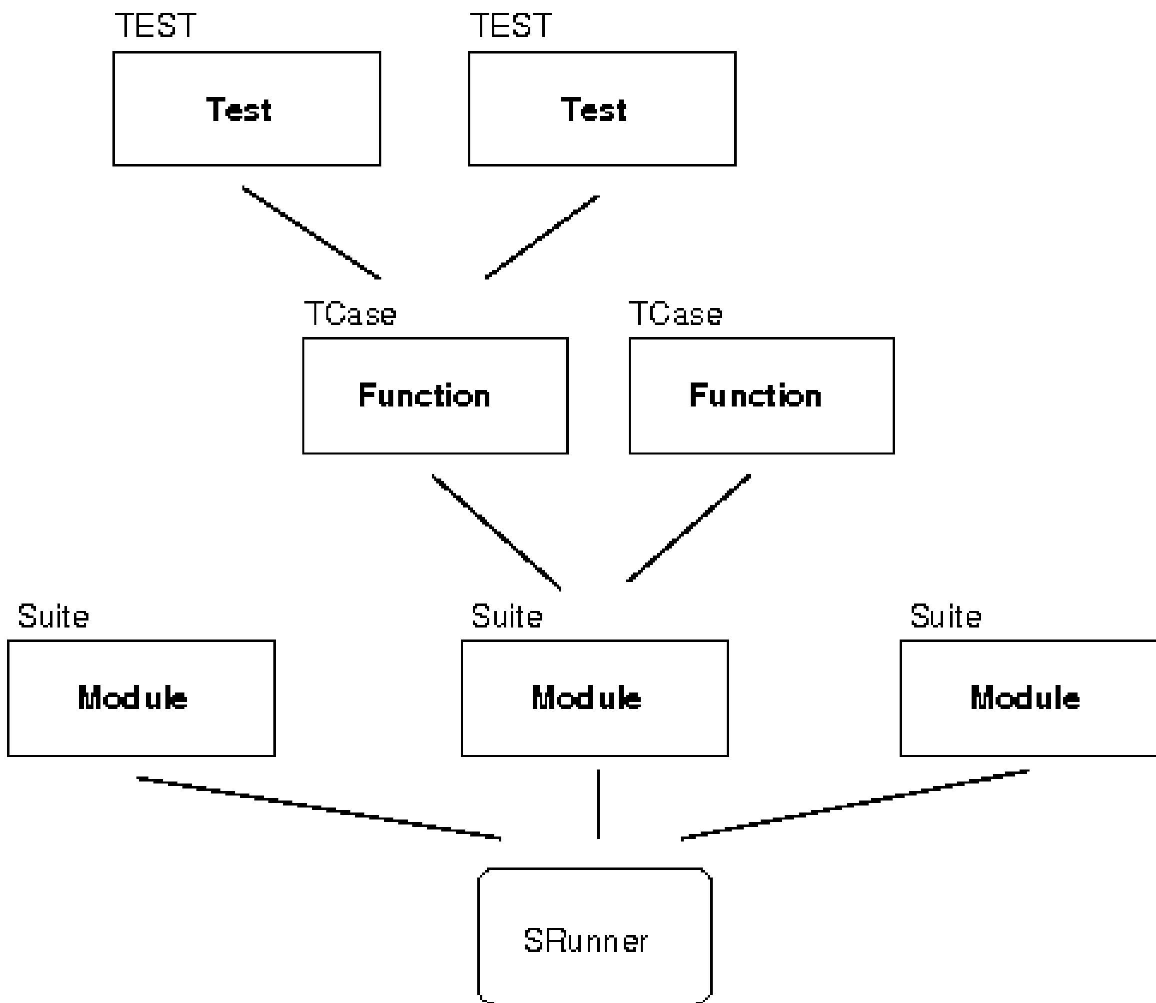

Figure 8.2: Unit Testing Architecture

Test suites are used to collect unit tests for a given module. In turn each test suite contain a collection of test cases. Each test case identifies a function implemented in the module. Several tests can then be defined to test the function capabilities.

The unit tests are stored in the 'torture/unit/' directory.

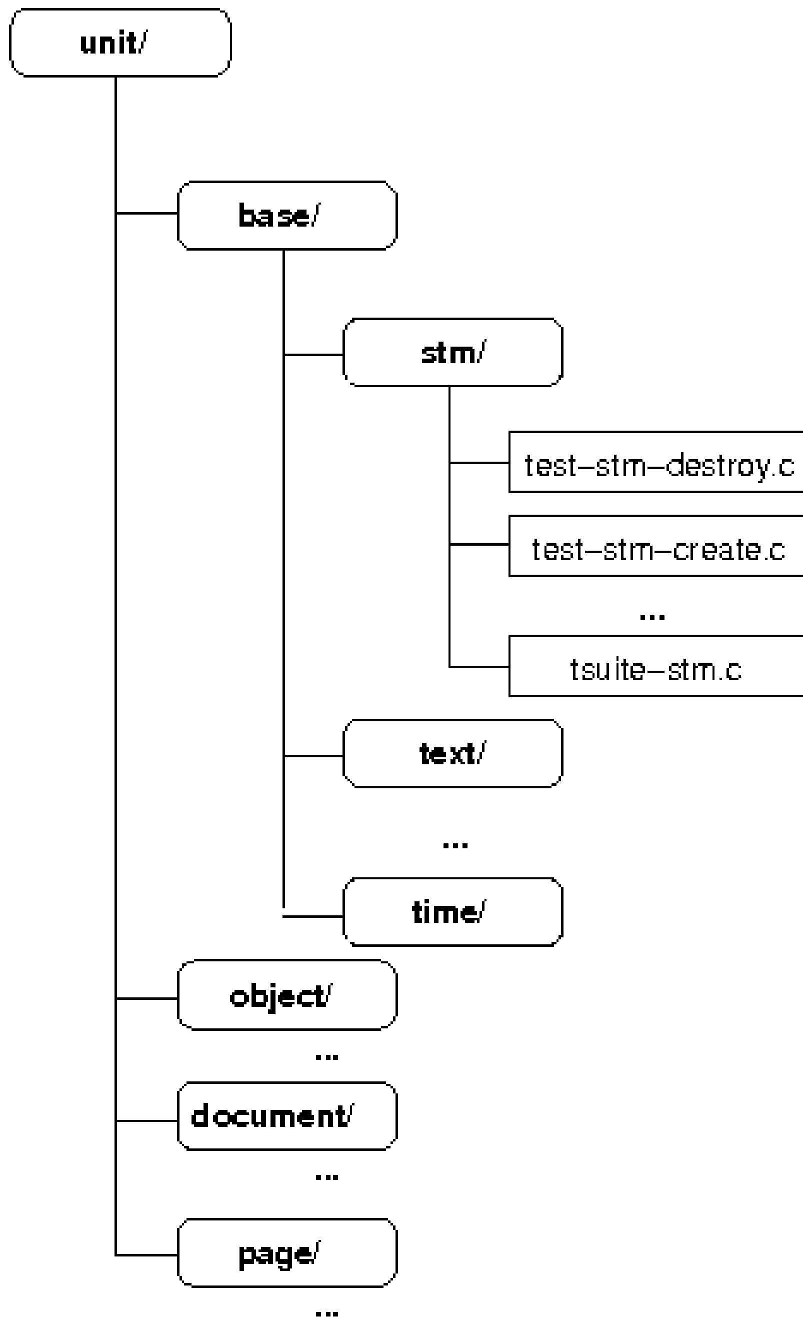

Figure 8.3: Unit Testing Sources

8.2.1 Designing unit tests

Please keep in mind the following considerations when designing unit tests for the functions of a library module:

For any given function we would like to define **positive**, **negative** and **stressing** unit tests:

- **Positive tests** are used to test the function with valid input data. The return value (or any documented side effect) of the function should be checked with the expected effect.
- **Negative tests** are used to test the function with invalid input data.
- **Stressing tests** are used to test the function with strategic or "interesting" valid input data (such as `MAXINT` or `MININT` in a function accepting an integer value).

Many functions are simple enough to only require a single positive test running the function (no parameters). It is important to have this unit test since the execution may fail even for such simple functions.

While designing the unit tests you may find that the API contain errors or that it may be improved in any way. That is fine and it is quite welcomed. You can propose any change of the API in this list.

8.2.2 Test files

The test files contain collections of unit tests associated with a function. Each test file is named '`function-name.c`', where '`function-name`' is the name of the function under testing with underscores replaced with dashes.

For example, the source file containing unit tests for the pdf_stm_new function would be called '`stm-new.c`'.

A minimal test file looks like this:

```
/*
 * Include the check library.
 */
#include <check.h>

/*
 * Test: FUNCTION_NAME_NNN
 * Description:
 *   Description of the test.  Can be of several lines long,
 *   indenting the text like this.
 * Success conditions:
 *   A list of success conditions.
 * Data files:
 *   A list of data files used in this test.
 */
START_TEST(FUNCTION_NAME_NNN)
{
  /* Check a condition.  One of several types of check
     available in the check library.
   */
  fail_if(0 == 1);
```

```
}
END_TEST


/*
 * Provide this function to gather all the tests together.
 */
TCase* test_FUNCTION_NAME (void)
{
  TCase* tc = tcase_create ("FUNCTION_NAME");
  tcase_add_test(tc, FUNCTION_NAME_NNN);
  return tc;
}
```

Note the `test_FUNCTION_NAME` function. It is the function called by the test driver in order to perform all the tests implemented in the test file.

Note also that the comments heading tests are written in a fixed format to allow the 'build-aux/generate-tsd.pl' script to generate the bulk of the Test Specification Document from the comments.

8.2.3 Test suite files

Once you add a new test file, that defines a testcase for some module function, you need to integrate it into the test suite that defines the module itself.

Each module directory (such as 'torture/unit/base/stm') has a test suite definition file such as 'tsuite-stm.c'. The suite definition file defines a function called 'tsuite_MODULE-NAME' that return a `Suite *` object.

New test cases are added using the `suite_add_tcase` function:

```
extern TCase *my_new_tcase (void);

Suite *
tsuite_mymodule ()
{
  Suite *s;
  ...

  suite_add_tcase (s, my_new_tcase ());
  ...

  return s;
}
```

8.2.4 The runtests program

The runtests program is the driver of the testing procedure. It is located in the 'torture/unit/' directory.

In order to integrate new test files into the runtests program the source files of the test files should be added to the value of the `TEST_FILES` variable in 'torture/unit/Makefile.am'.

For example:

```
TEST_FILES = base/stm/pdf-create-file-stm.c \
             base/OTHER-MODULE/OTHER-TEST-FILE.c
```

8.2.5 Running the unit tests

The runtests program will run all the defined unit tests of the library. It can be invoked (and it is the preferred way) by executing:

```
$ make check
```

It is possible to run the unit tests corresponding to a specific library module by using the `MODULE` variable:

```
$ make check MODULE=fsys
```

It is also possible to run the unit tests corresponding to a specific public function by using the `FUNCTION` variable:

```
$ make check FUNCTION=fsys_close_file
```

The test driver display several messages to the standard output and also dump a logfile named '`ut.log`' with details about the test execution.

The runtests program will (by default) output a list of which test suites were run, then a summary line followed by a list of failing lists. You can get a full list of tests (passing and failing) by setting the `CK_VERBOSITY` environment variable to 'verbose'. You can get just the summary line and failing lists by setting the `CK_VERBOSITY` environment variable to 'minimal', and you can produce no output by setting it to 'silent'. Note that tests (in particular, the error reporting tests) may output additional information as part of their normal operation - that isn't really part of the check testing framework, and won't be affected by the `CK_VERBOSITY` environment variable.

If the library was cross compiled in a GNU system using mingw32, `make check` will try to invoke `wine` to execute the '`runtests.exe`' binary. If the compilation was performed in a native windows environment then '`runtests.exe`' will be used directly.

8.2.6 Using gdb to debug check tests

The check testing framework uses fork calls in order to create the processes used to run the single tests. This makes possible to catch unexpected process terminations such as a segmentation fault or a division by zero.

Sometimes we want to debug those failure conditions using gdb. Unfortunately the GNU debugger cannot catch the unexpected termination of the child processess.

The check implementor foresaw this and provides a workaround: to define the `CK_FORK` environment variable to `"no"` and launch the debugger.

The test driver '`torture/unit/runtests`' is a shell script generated by libtool. This means that it is not possible to run it invoking gdb from the command line, as in:

```
$ gdb torture/unit/runtests
```

A solution for this problem is to edit the '`torture/unit/runtests`' and hack it so it will call gdb instead of the program in '`.libs/`'. Just find the line containing something like:

```
exec "$progdir/$program" ${1+"$@"}
```

and change it to something like:

```
exec gdb --return-child-result --quiet --args "$progdir/$program" ${1+"$@"}
```

Note that you need to repeat this change every time 'torture/unit/runtests' is rebuilt by `make`.

The second alternative is to run gdb on the real binary using:

```
$ LD_LIBRARY_PATH=/path/to/libgnupdf/src/.libs:$LD_LIBRARY_PATH gdb torture/unit/.libs
```

When debugging failing tests you may find it useful to set breakpoint to `_fail_unless` function:

```
$ ( export CK_FORK='no'; make check )   # you can use 'make check FUNCTION=...' as well
...
(gdb) break _fail_unless
Breakpoint 1 at 0x804ac44
(gdb) run
...
Breakpoint 1, _fail_unless (result=1, file=0x80b2f94 "base/alloc/pdf-alloc.c",
    line=47, expr=0x80aac30 "Failure 'data == NULL' occured") at check.c:237
237       send_loc_info (file, line);
...
(gdb) list
232     void _fail_unless (int result, const char *file,
233                        int line, const char *expr, ...)
234     {
235       const char *msg;
236
237       send_loc_info (file, line);
238       if (!result) {
239         va_list ap;
240         char buf[BUFSIZ];
241
(gdb) break check.c:238 if (!result)
Breakpoint 2 at 0xb7ea4a7c: file check.c, line 238.
(gdb) delete 1
(gdb) continue
Continuing.
...
Breakpoint 2, _fail_unless (result=0,
    file=0x80a7ef8 "base/text/pdf-text-new-destroy.c", line=44,
    expr=0x80a7f1c "Assertion 'pdf_text_new (&newtext) == PDF_EBADCONTEXT' failed")
    at check.c:238
238       if (!result) {
(gdb) finish
Run till exit from #0  _fail_unless (result=0,
    file=0x80a7ef8 "base/text/pdf-text-new-destroy.c", line=44,
    expr=0x80a7f1c "Assertion 'pdf_text_new (&newtext) == PDF_EBADCONTEXT' failed")
```

```
    at check.c:238
pdf_text_new_destroy_001 (_i=0) at base/text/pdf-text-new-destroy.c:46
46        END_TEST
(gdb) list
41        {
42          pdf_text_t newtext = NULL;
43
44          fail_unless(pdf_text_new (&newtext) == PDF_EBADCONTEXT);
45        }
46        END_TEST
47
48        /*
49         * Test: pdf_text_new_destroy_002
50         * Description:
(gdb) ...
```

8.3 Test Data Files

Some tests require the use of data files:

- to hold input for the software under test
- to hold output to be compared with the output of the software under test

The test data files are distributed in the '`torture/testdata`' directory along with the source code.

The test data files are documented in the "Test Data Files" chapter in the Test Specification Document.

8.4 The tortutils Library

The "torture utils" library provides several utility functions that can be used while writing tests.

See '`torture/tortutils/tortutils.h`' for documentation on the functions provided by the library.

9 Cross-compiling libgnupdf for Windows under GNU/Linux

This chapter documents all the needed steps to setup a mingw build of the GNU PDF library. MinGW provides a complete programming tool set which is suitable for the development of native MS-Windows applications.

Getting and installing MinGW

If you are under debian, you can install mingw32 with the following command:

```
$ apt-get install mingw32
```

If you are not on an usual disto, download mingw32 at http://sourceforge.net/projects/mingw MinGW Build Tool. You follow the README in order to install the package. You can go to http://mingw.org for more details:

```
...
$ sh x86-mingw32-build.sh i686-mingw32
...
```

The dependencies

After having downloaded the sources of the GNU PDF Library, you need to install the required libraries. Below is a list of the required libraries:

- zlib
- libpthread
- libuuid
- libgpg-error
- libgcrypt
- libcheck
- libjbig2dec
- libjpeg

You need to create some directories in order to stock the mingw32 dependencies.

```
$ mkdir $HOME/w32root
$ mkdir $HOME/w32root/lib
$ mkdir $HOME/w32root/include
```

zlib Download zlib either the source or the binaries. In order to compile the source you have to follow to read the page available at http://www.crosscompile.org/static/pages/ZLib.html. The binaries are available at http://zlib.net/zlib125-dll.zip

```
$ unzip path/to/zlib125-dll.zip
$ mv include/* $HOME/w32root/include
$ mv zlib1.dll $HOME/w32root/lib/libz.dll
```

libpthreads

There is a bug in *mingw32-pthreads* (Bug 599227). This bug affects the compilation of libcheck. You need to download the source of the fixed version. You have to log in the pthreads-win32 cvs. The password is *anoncvs*. Then, you download, compile and install.

```
$ cvs -d :pserver:anoncvs@sourceware.org:/cvs/pthreads-win32 login # anoncvs
$ cvs -d :pserver:anoncvs@sourceware.org:/cvs/pthreads-win32 checkout -D 201
$ cd pthreads-w32-20110511
$ make CROSS=i586-mingw32msvc- clean GC
$ cp pthread.h sched.h semaphore.h $HOME/mingw32/include/
$ cp pthreadGC2.dll $HOME/mingw32/lib/libpthread.dll
```

libuuid You can use precompiled *libuuid* binaries available in the following URL: http://mirror.vcu.edu/pub/windows/cygwin/release/util-linux/libuuid-devel/. So, steps for *libuuid* are:

```
$ tar -jxf libuuid-devel-2.17.2-1.tar.bz2
$ cp -r usr/include/uuid $HOME/w32root/include
$ cp -r usr/lib/libuuid.dll.a $HOME/w32root/lib
```

libgpg-error and libgcrypt

Go to http://gnupg.org/download/index.en.html. You have to download:

- 'libgpg-error-1.10.tar.bz2'
- 'libgcrypt-1.4.6.tar.gz'

For *libgpg-error-1.10* and *libgcrypt-1.4.6*, you have to note that the 'autogen.sh' script recognizes *—build-w32* and set the prefix installation to '$HOME/w32root'. You can't modify this prefix. A typical compilation and installation would be:

```
$ tar zxf path/to/libgpg-error-1.10.tar.gz
$ cd libgpg-error-1.10
$ ./autogen.sh --build-w32
$ make
$ make install

$ tar zxf path/to/libgcrypt-1.4.6.tar.gz
$ cd libgcrypt-1.4.6
$ ./autogen.sh --build-w32
$ make
$ make install
```

The libraries are installed in '$HOME/w32root'.

libcheck You need the svn version of *libcheck*:

```
$ svn co https://check.svn.sourceforge.net/svnroot/check/trunk check
```

In order to compile, you need to type:

```
$ cd check
$ autoreconf --install
$ ./configure --host=i586-mingw32msvc --prefix=$HOME/w32root
$ make
$ make install
```

libjbig2dec

Download http://ghostscript.com/~giles/jbig2/jbig2dec/jbig2dec-0.11.tar.gz. Compile and install the library with the following commands. Note that due to http://bugs.ghostscript.com/show_bug.cgi?id=691784, you'll need to generate the DLL manually.

```
$ ./configure --host=i586-mingw32msvc --prefix=$HOME/w32root
$ make
$ make install
$ i586-mingw32msvc-gcc -shared -o .libs/libjbig2dec.dll  jbig2.o jbig2_arith
 jbig2_segment.o jbig2_page.o jbig2_symbol_dict.o jbig2_text.o jbig2_generic
 jbig2_image.o jbig2_image_pbm.o jbig2_metadata.o
$ cp .libs/libjbig2dec.dll $HOME/w32root/lib
```

libjpeg Download *libjpeg* at http://www.ijg.org/files/jpegsrc.v8c.tar.gz. Compile and install the library with the following commands.

```
$ cd jpeg-8c
$ ./configure --host=i586-mingw32msvc --prefix=$HOME/w32root
$ make
$ make install
```

Compiling Now you can run in the GNU PDF trunk:

```
$ sh autogen.sh
$ ./configure --host=i586-mingw32msvc --with-zlib=$HOME/w32root --with-libch
 --with-libjpeg-prefix=$HOME/w32root --with-libuuid-prefix=$HOME/w32root
$ make
$ make install
```

At the time of writing this documentation, there are some errors during `make`.

10 Updating the AUTHORS file

The 'AUTHORS' file is automatically generated using Emacs. The 'prmgt/authors.el' elisp file contains the implementation of the `author` function.

In order to regenerate the file:

1. Launch Emacs
2. Load the 'prmgt/authors.el' file.
3. Call *M-xauthors* and specify the libgnupdf source tree.
4. Update AUTHORS with the contents of the `*Authors*` buffer.