FFTW_Ada Version 1.2

User and Reference Manual

FFTW_Ada is an Ada 95 binding to the FFTW library written at MIT by Matteo Frigo and Steven G. Johnson. FFTW is a library for computing the Fast Fourier transform, which is both fast, and copes with arbitrary sizes of array, in multiple dimensions. FFTW is written in C and the purpose of FFTW_Ada is to allow calls to FFTW from an Ada 95 program. It includes a thin binding (a fairly direct interface to the C code, using Ada 95 types, but C concepts) and a thick binding (an interface in Ada 95 style, which makes FFTW look like an Ada 95 package). There is also a package which makes the Ada 95 real time clock (if available) usable to FFTW for timing purposes, and a test program. FFTW_Ada, like FFTW, is released under the GNU General Public License.

July 1, 2002

Contents

1 Introduction

This is the user guide and reference manual for FFTW_Ada, an Ada 95 binding to the FFTW library of Fast Fourier Transform (FFT) code written at MIT by Matteo Frigo and Steven G. Johnson. FFTW is written in C. The purpose of FFTW_Ada is to make it possible to use FFTW from an Ada 95 program. This is achieved by providing an Ada 95 interface to the C code (known as a *binding*). The binding is compiled by an Ada 95 compiler and linked with object code produced by compiling the FFTW C code. When linked into an Ada 95 application program, it appears as if the FFTW library were written in Ada 95. All of this is made possible by the clean design of FFTW and the advanced interfacing features of Ada 95 as defined in the Ada 95 Language Reference Manual, Appendix B [1]. This means that the binding is portable across machines and operating systems.

FFTW was written by Steve Sangwine while at the University of Reading, UK.

1.1 Some words of caution

FFTW_Ada does not install itself 'out of the box'. Compilation of the Ada 95 code is straightforward, but getting it fully working with the C code of FFTW will take some methodical work. Don't rush it. I cannot provide support for installing the package on numerous operating systems and with various Ada 95 compilers. The package is written to be portable, in the sense that the Ada 95 code uses only standard Ada 95[1]. If you install FFTW_Ada on a system for which there is currently no advice given, and you are willing to write a description of what you did and any pitfalls you overcame, send me the details and I will make them available to other users.

The current version of FFTW_Ada (1.2) should not be considered fully tested. A test program is included, and this has been used to run a fairly comprehensive, and random, test of the high level facilities of FFTW_Ada (the thick binding). There are some facilities in the thin binding (low-level Ada 95) that have not been tested, because they are not exercised by the test program. There is not much code here to be wrong, but it should be tested before use in an application. FFTW_Ada has been tested with tasking, using a simple test in the test program provided (three tasks, testing 1D, 2D and 3D FFTs respectively), but the task safety of FFTW_Ada should not be considered totally reliable without more extensive testing.

1.2 FFTW itself

This manual gives only minimal information about FFTW itself. You must refer to the manual for FFTW for most of the information about it, including details of what it computes, and how, and how to install/compile it.

FFTW_Ada does not provide any help with installing FFTW – you have to do that before you can make use of FFTW_Ada. The section on installation on page 10 gives some advice on this for those without much previous experience of compiling and building C code.

[1] A harmless exception is `pragma Assert` which may not be implemented by all compilers. If it is not, the compiler is required to give a warning (LRM 2.8(13)), but otherwise ignore the pragma (LRM 2.8(11)).

FFTW determines how to compute FFTs of given size by timed experiment. The results are specific to the machine on which the experiment was done and they can be saved to a file, so that applications using FFTW only have to experiment once with a given transform size, and thereafter can use the previous experimentally determined method. The timed experiments require FFTW to have access to a clock. The clock facilities available in C are platform dependent, and FFTW's code includes numerous options for this. Ada 95 provides an optional real-time clock which, if available, provides a much easier option than configuring the C code. FFTW_Ada provides FFTW with access to the Ada 95 real-time clock by providing an Ada 95 package which exports the real-time clock for use by FFTW's C code.

1.3 Components

FFTW_Ada is structured as a hierarchy of packages and consists of the following main components:

- A root package `FFTW_Ada`. This defines some low-level types and values and an exception for handling of fatal errors in FFTW.
- A thin binding to the FFTW complex FFT code. This consists of a generic Ada 95 package called `FFTW_Ada.Generic_Thin_Binding` which is generic on the real type used in order to support both single and double precision floats (using these terms in their C sense). An instantiation of `Ada.Numerics.Generic_Complex_Types` is also required as a generic parameter.

  The thin binding provides a fairly direct interface to the C code of FFTW and provides the foundation on which more Ada-like interfaces may be (and have been) constructed. It is not intended that the Ada 95 user should utilise this binding directly from an application program. If the FFTW_Ada thick bindings do not provide what is needed (and there is no provision for dimensions higher than three in the thick binding) the user should construct a thick interface by extending FFTW_Ada and thereafter access FFTW through the thick interface.
- Library unit instantiations of the package `Ada.Numerics.Generic_Complex_Types` for the two floating-point types provided in `Interfaces.C`. These are the only real types which can be guaranteed to be C-compatible on any platform/compiler. Of course, on a specific platform/compiler (especially one using IEEE floating-point arithmetic) it is possible that standard Ada 95 types such as `Float` and `Long_Float` will also work. These instantiations are called:

  1. `FFTW_Ada.Single_Complex_Types` and
  2. `FFTW_Ada.Double_Complex_Types`.
- Library unit instantiations of the thin binding for the C types `float` and `double`, using the instantiations of `Ada.Numerics.Generic_Complex_Types`. These instantiations are called:

  1. `FFTW_Ada.Single_Thin_Binding` and

2. `FFTW_Ada.Double_Thin_Binding`.

- Library unit instantiations of the thin binding for the standard Ada floating-point types `Float` and `Long_Float`. These instantiations are *not* guaranteed to work with all Ada compilers. (The reasons are documented in the headers of the source code files.) These instantiations are called:

  1. `FFTW_Ada.Thin_Binding` (for `Float`) and
  2. `FFTW_Ada.Long_Thin_Binding` (for `Long_Float`).

- A thick binding to FFTW.

  This consists of an Ada 95 package called `FFTW_Ada.Thick_Binding` and three generic children called `FFTW_Ada.Thick_Binding.Generic_{1|2|3}D`. The thick binding hides the detailed parameters needed by FFTW and presents a high-level interface using complex arrays (*not* pointers or access values, as used in the thin binding).

  The package `FFTW_Ada.Thick_Binding` provides a high-level mechanism to represent and manipulate FFTW plans.

  The three generic child packages for 1, 2 and 3 dimensions provide the means to create plans specific to that number of dimensions, and they operate on complex arrays. Because the complex array types are generic parameters the user is free to use any index types and ranges. The complex type must be the same complex type used to instantiate the thin binding (enforced by making the thin binding a generic parameter). Each generic package provides for the calculation of both in-place and out-of-place transforms.

- Packages to handle wisdom. The first is called `FFTW_Ada.Wisdom` and provides a high-level (Ada 95 style) interface to FFTW's wisdom string import and export procedures. The second is a child package of the first and is called `File_IO`. It provides procedures to read and write wisdom to and from a file of user-specified name.

- An interface to the Ada 95 real-time clock, for use by FFTW's C code. This is in a package called `FFTW_Ada.Real_Time`. FFTW can be configured to make use of clock facilities available to C, but if your Ada 95 implementation provides the optional facilities of the Real-Time Systems annex, you have access, in Ada 95 to a real-time clock which is likely to be as good as or better than that available to C (and more importantly, much easier to get working because all you need to do with FFTW's code is insert a `#include` to a file generated by a component of FFTW_Ada and compile the interface code.

- A stand-alone Ada 95 program called `Make_RTC_Header`. This generates a customized file called `fftw_ada.h` which is a C header file. This file can be `#included` into FFTW's configuration file to make the Ada 95 real-time clock available to FFTW. Because it checks the timing resolution of the Ada 95 real-time clock, FFTW will make use of whatever resolution is available.

- A generic test program (for single or double precision) with instantiations for the standard Interfaces.C single and double precision types. The test program uses the Ada real-time clock to log the time taken to compute each transform. If the package `FFTW_Ada.Real_Time`

is not available, this aspect of the code will have to be modified. The test program verifies randomly chosen FFT results against a very simple (and slow) DFT implementation computed using the maximum precision available. It accepts command line parameters to specify the maximum array sizes, the proportion of results to be verified, whether to enable tasking, and the filenames to use for import/export of wisdom and logging of results. It tests the 1D, 2D and 3D FFTs through the thick binding, exercising the forward/backward directions, in place and out of place computation, with and without the use of wisdom, using both the estimate and measure planning options for randomly chosen sizes of array. Any results which differ from the DFT implementation by more than a certain tolerance are logged.

The test program does not exercise all the facilities of the thin binding.

- A test program called `FFTW_Ada_Test_FFTW_Die` which exercises the handling of FFTW's fatal error routine. The test program calls FFTW's fatal error routine. Correct behaviour is for the FFTW routine to pass control to FFTW_Ada which then raises the exception `FFTW_Dead`. The test program handles this exception and outputs appropriate text to standard output to indicate correct operation of the error handling.

At present there is no support for the FFTW real FFT code. This could be added if there is demand for it. A thin binding would be simple to produce. If you are interested in this let me know.

1.4 Tasking safety

FFTW is not thread safe. Although it can compute FFTs re-entrantly, it can compute plans only one at a time. The FFTW_Ada thin binding takes care of this so that multiple planning calls from a tasking application will not cause problems. (Subsequent calls are queued until the first completes using a protected flag – transparent to the user, of course.) The thick binding makes all its planning calls through the thin binding and therefore it can also be used safely in a tasking application.

Of course, tasking may mess up the timing of the FFT experiments used in planning, so it would be advisable to generate measured plans using non-tasking code. (One way to do this is to run the test program *with tasking disabled* to generate wisdom, ensuring that the test program is not time-sharing with other applications.)

2 Implementation details and design rationale

2.1 The thin binding

The thin binding is written as a generic package with two generic parameters: the real type which is to be used as a component of the complex type to be passed to FFTW, and an instantiation of the package `Ada.Numerics.Generic_Complex_Types`. This means that the thin binding can be generated in single and double precision versions, according to the version of FFTW that is to be used.

The single precision thin binding instantiation (in `FFTW_Ada.Single_Thin_Binding`) uses the C `float` type (called `C_float` in Ada 95).

The double precision thin binding instantiation (in `FFTW_Ada.Double_Thin_Binding`) uses the C `double` type (called `double` in Ada 95).

Although it is possible to instantiate the thin binding for other types, it is not possible to use arbitrary types. The Ada 95 floating-point type and the C floating-point types must have the same representation: the thin binding just passes the floating-point arrays to the C code without a change of representation. If your compiler uses IEEE arithmetic this will not be a problem. On non-IEEE machines, the only C-types guaranteed to work are those in the Ada 95 standard package `Interfaces.C`. For convenience, and for use only if your compiler provides compatibility between C and Ada floating-point types, there are two instantiations provided for the standard Ada types `Float` and `Long_Float`.

2.1.1 FFTW facilities supported

The thin binding provides mostly direct translations of FFTW facilities into Ada 95, but uses exclusively Ada 95 types.

The names used in the thin binding match those in the FFTW code fairly obviously, but use has been made of overloading where this gives shorter and more natural names. It is straightforward to understand the thin binding by reference to the FFTW manual where the C code is documented. If in doubt, look at the code in the body, where the correspondence between the FFTW_Ada and FFTW subprogram names is clear.

Plans are handled using Ada 95 private types. There is no access to the details of the plan. The reasoning behind this is simply that wisdom provides a suitable high-level way for an Ada 95 application to save and restore information about how to compute a particular FFT.

The thin binding provides access to the one-dimensional, and multi-dimensional FFTs in FFTW. The 2D and 3D FFTs are likely to be usable from an Ada 95 application fairly easily (and the thick binding provides this). For dimensions higher than 4, there is a difficulty in providing an Ada 95 thick binding in a generic manner, since Ada 95 arrays must have a fixed number of dimensions. If you really need 7 dimensions, you can easily build your own thick binding to the N-dimensional code through the thin binding.

2.2 The thick binding

The thick binding consists of four packages: a ‘root’ package and three generic children for 1, 2 and 3 dimensions. The generic child packages require an instantiation of the thin binding as a generic parameter. This means that the 1, 2 and 3D components of the thick binding can be generated in single and double precision versions, according to the version of FFTW that is to be used.

Plans are handled at a higher level in the thick binding than they are in the thin binding. Again, a plan is a private type, but since the thick binding stores in the plan the details of the FFT for which

it was created, error checking is implemented, and if the plan is incorrect, an exception is raised (`Plan_Error`). When an FFT is to be calculated, the plan dictates the transform direction.

The thick binding supports 1D, 2D, and 3D transforms, by providing a generic child package for each. The generic parameters of these packages are: the thin binding; an index type; and a constrained array type of complex values, the complex type being that used to instantiate the thin binding. The user is not forced to use a particular index type (any discrete type is acceptable, but the real gain is that the user is not forced to use `0 .. N - 1`, or `1 .. N`, or any other specific indexing scheme). The choice of a constrained type was dictated by generality. If an unconstrained type was used, the user would have to use unconstrained arrays. Constrained arrays can be declared as subtypes of an unconstrained type, so the choice of constrained arrays does not preclude the use of unconstrained arrays in the user's application (but an instantiation of the generic package for the appropriate number of dimensions will be needed for each constrained subtype). To explain how this operates, for those who have not made extensive use of generics, or those new to Ada 95 consider an image processing application in which images are read from a file and an FFT has to be computed. The image size will not be known until after the file has been opened, and some header information read from the file. At this point, a constrained subtype of an unconstrained 2-dimensional complex array type may be declared to match the image size. Then, the nested generic 2D package in the thick binding may be instantiated, a plan created, and the FFT computed.

The 1D, 2D and 3D transforms support *interleaved* arrays. This is a specific concept implemented using a much more general concept in FFTW. The general concept in FFTW allows the complex numbers making up an array to be stored at non-consecutive addresses (for example, the complex values may be fields within the components of an array of records). The thin binding provides access to this general concept *via* the *stride* parameters. The specific concept supported by the thick binding is that of an *interleaved array*, where a single complex array (of 1, 2 or 3 dimensions) contains more than one set of complex data to be transformed. These multiple sets of data are stored in interleaved manner, so that the first complex number in memory is part of the first set of data, the second is part of the second set of data, and so on. Why is this useful? Well, a particular example may make the reason for providing this facility clear: the author has a specific use for the case of two interleaved arrays, where the data is not actually complex, but quaternion-valued [2]. Quaternions are a type of hypercomplex number with four components. They may be separated into a pair of complex numbers, and Fourier transforms of quaternion arrays may be computed using FFTW (or any other complex FFT code) by separating the quaternions (stored as four consecutive real values in memory) into a pair of complex values. The obvious way to do this is *in place* so that the single quaternion array becomes an interleaved array of two sets of complex data. There are probably other cases where the ability to compute interleaved transforms is useful. (If you know of one, please let me know.)

2.3 Wisdom

Wisdom is implemented in a straightforward way by passing strings in and out of FFTW. This is common to the thin and thick bindings, which is why it is factored out into a separate child package. The facilities in the child package `FFTW_Ada.Wisdom.File_IO` are implemented using `Ada.Direct_IO`.

2.4 The real-time clock interface

The package `FFTW_Ada.Real_Time` provides an interface to the Ada 95 real-time clock in the package `Ada.Real_Time`. This package may not be provided with all Ada 95 implementations (it is part of the optional Real-Time Systems annex).

The purpose of this package is to allow FFTW to make use of the Ada 95 clock, rather than relying on access to the operating system default clock. Since the Ada 95 clock interface is portable, this makes it easier to install FFTW in an Ada 95 environment. It is not essential to install the Ada 95 clock interface – you may prefer to set up FFTW to use its own C clock routines.

The Ada 95 real-time clock interface is simply a C binding to the facilities in the Real-Time Systems annex package `Ada.Real_Time` using **`pragma`** `Export`. See the package source code to see how it is done.

2.5 Fatal error handling

FFTW includes a procedure `fftw_die` which includes a hook (a C pointer) which can be set to point to a user-defined error handling routine. FFTW_Ada sets this hook to point to Ada 95 code in FFTW_Ada so that the exception `FFTW_Dead` can be raised. There is a potential portability problem here, because the Ada 95 LRM (B.3) does not specify interfacing requirements for the case of an exported C pointer being set to point to an Ada 95 procedure, although in all probability it will work with any reasonable Ada 95 compiler. (The LRM does specify interfacing requirements for C pointers passed as parameters, but FFTW does not provide a procedure mechanism for setting the hook.) This is why a test program is provided for testing the fatal exception handling. Fortunately, FFTW exports the fatal error routine, so that it can be called for test purposes (in this case from Ada 95 code).

3 Installation

The recommended way to install FFTW_Ada is to start by installing FFTW by following all the relevant instructions in the FFTW manual, plus any platform specific advice given on the FFTW website at http://www.fftw.org/.

It is best to have a working C system before attempting to get the Ada 95 binding working. Since FFTW comes with a fairly comprehensive test program, it is a good idea to use this test program to make sure all is well. If you are able to make use of FFTW_Ada's Ada 95 real-time clock interface, you can install FFTW to use the basic default timing code, since this will only be needed temporarily for testing the C installation. If your Ada 95 implementation doesn't provide the Ada 95 real-time clock, you should customize the FFTW code as documented in the FFTW manual to get the best possible timing performance, and test to make sure it works.

If you are an Ada 95 programmer without previous experience of mixed Ada 95/C programming, appendix C explains how to compile and build FFTW using the Gnat compiler.

Instructions for building `rtc_test` with Gnat: The following commands will compile, bind and link the C main program `rtc_test` using the Gnat compiler:

```
gcc      -c rtc_test.c
gnatmake    fftw_ada-real_time.adb
gnatbind -n fftw_ada-real_time
gnatlink    fftw_ada-real_time rtc_test.o -o rtc_test
```

The first line compiles the C main program, the second the Ada 95 real-time interface package. The binder step finds all the **`with`**'ed Ada 95 units and outputs an Ada 95 program for compilation and linking. The linker combines all the object code into an executable (with the name `rtc_test`).

3.1 The real-time clock interface

3.1.1 The program `Make_RTC_Header`

In order to use the real-time clock interface within FFTW it is necessary to insert appropriate declarations into the FFTW file `fftw-int.h`. To make this easier FFTW_Ada provides the Ada 95 program `Make_RTC_Header` which should be compiled and run. It determines the real-time clock tick for your Ada 95 run-time system, and outputs the file `fftw-ada.h` with a suitably customized declaration of the minimum measurement time that FFTW should use. The following section explains how to set up use of this new header file.

3.1.2 The C program `rtc_test`

To test that the real-time clock interface has been correctly installed, FFTW_Ada includes the simple C program `rtc_test` which may be compiled and linked with `FFTW_Ada.Real_Time`. To do this, your Ada 95 compiler must support linking of Ada 95 code with a C main program. How this is done will vary from compiler to compiler. Some specific instructions are given in the panel for use with the Gnat compiler to illustrate the method, but in general the steps are: compile the Ada 95 package `FFTW_Ada.Real_Time`; compile the C program `rtc_test`; bind and link the two with the necessary run-time library.

The code of `rtc_test` includes some features specific to the Gnat compiler for initialising the Ada 95 code. These are identified and explained in comments and should be amended for use with other compilers.

When `rtc_test` is run, it should output a list of times at one second intervals, showing that the clock interface is working (the output is being done by C code, the clock lookup by Ada 95 code).

If `rtc_test` works, the next step is to link FFTW itself with the Ada 95 real-time clock interface and test it again. To do this, add the following code to the files shown:

- In the file `config.h`, add the following line somewhere in the sequence of definitions:
  `#define FFTW_ADA`

- In the file `fftw-int.h`, add the following lines just before the section headed `VANILLA UNIX/ISO C SYSTEMS`. (This is part of a long `#if` ... `#elif` ... `#else` ... `#endif` construct which includes code for timers on various machines.)

```
/*********************************************
 *       FFTW_ADA REAL-TIME CLOCK PACKAGE      *
 *********************************************/
#elif defined(FFTW_ADA)

#include <fftw-ada.h>
```

Recompile the FFTW C code and remake the libraries. Then, follow any steps that you used to get the program `rtc_test` working for the FFTW test program, so that it can be linked with the FFTW real-time clock package. You should then be able to run the FFTW test program again, but it will now use the Ada 95 real-time clock instead of the clock provided by the C run-time system.

3.2 Compiling and testing the bindings

This part of the installation is fairly straightforward. All you need to do is compile the various Ada 95 packages. A test program is provided to allow you to verify that the compiled code will link and work with FFTW. As a side effect, the test program will create a wisdom file which could be used in an application. (You could use a long run of the test program to generate a lot of wisdom for many array sizes, and perhaps not need to write your application code to generate wisdom.)

The main complication when linking C code with Ada 95 code is getting the linker to find the C object code. If your Ada 95 implementation supports **`pragma`** `Linker_Options` you can put this pragma into your application code and then just follow your usual steps for compiling Ada 95 code. The pragma has not been used in the distributed code of FFTW_Ada because the parameter of the pragma is not portable. If **`pragma`** `Linker_Options` is not possible, you will need to specify the linker options and object files when you run your linker.

You should also compile, link and run the test program `FFTW_Ada_Test_FFTW_Die` to make sure that your Ada compiler correctly implements the linkage between FFTW and FFTW_Ada.

4 License, Copyright and Lack of Warranty

The following notice is included in all the source files of FFTW_Ada:

```
------------------------------------------------------------------------
------------------------------------------------------------------------
-- FFTW_Ada -- An Ada95 Binding to the FFTW library (www.fftw.org) --
--                                                                    --
-- Copyright (©) 2000 Dr Stephen J. Sangwine (S.Sangwine@IEEE.org) --
--                                                                    --
-- This software was created by Stephen J. Sangwine. He hereby     --
-- asserts his Moral Right to be identified as author of this      --
-- software.                                                          --
------------------------------------------------------------------------
------------------------------------------------------------------------
-- FFTW_Ada is free software; you can redistribute it and/or       --
-- modify it under the terms of the GNU General Public License as  --
-- published by the Free Software Foundation; either version 2 of  --
-- the License, or (at your option) any later version.             --
--                                                                    --
-- FFTW_Ada is distributed in the hope that it will be useful, but --
-- WITHOUT ANY WARRANTY; without even the implied warranty of      --
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the    --
-- GNU General Public License for more details.                    --
--                                                                    --
-- You should have received a copy of the GNU General Public       --
-- License along with this software (in the file gpl.txt); if not, --
-- write to the Free Software Foundation, Inc., 59 Temple Place,   --
-- Suite 330, Boston, MA 02111-1307 USA                            --
--                                                                    --
-- FFTW_Ada was written by Steve Sangwine at The University of     --
-- Reading, United Kingdom.                                        --
--                                                                    --
-- The University of Reading has agreed to the public release of   --
-- this software under the GNU General Public Licence. Enquiries   --
-- concerning commercial licensing of the software should be       --
-- directed to the Research Support Office, The University of      --
-- Reading, Whiteknights, PO Box 217, Reading RG6 6AH, United      --
-- Kingdom.                                                        --
-- WWW: http://www.rdg.ac.uk/RSO/    Email: RSO@Reading.ac.uk      --
-- Tel: +44 118 931 8628             Fax: +44 118 931 8979.        --
------------------------------------------------------------------------
------------------------------------------------------------------------
```

Since FFTW_Ada is only a binding (interface) to FFTW, the license for FFTW_Ada does not grant any rights to use FFTW. Since both FFTW_Ada and FFTW are released under the Gnu General Public License, this will not be a problem for uses covered by that license. Any other use of FFTW_Ada requires licensing from The University of Reading, UK, and of course a separate license from MIT for FFTW, as detailed in the FFTW manual.

5 Comments and bug reports

Please send comments, suggestions or bug reports to: S.Sangwine@IEEE.org

Kindly include the string '`FFTW_Ada`' somewhere in the subject line as this triggers an email filter which separates email on this subject from other mail. PGP encrypted email is welcome. See the next section for details.

References

[1] Ada 95 reference manual, January 1995. ANSI/ISO/IEC-8652:1995.

[2] T. A. Ell and S. J. Sangwine. Decomposition of 2D hypercomplex Fourier transforms into pairs of complex Fourier transforms. In Moncef Gabbouj and Pauli Kuosmanen, editors, *Proceedings of EUSIPCO 2000, Tenth European Signal Processing Conference*, volume II, pages 1061–1064, Tampere, Finland, 5–8 September 2000. European Association for Signal Processing.

A Distribution

FFTW_Ada is available from http://privatewww.essex.ac.uk/~sjs/fftw_ada/fftwa.html. It is distributed in zip format and in gzipped tar format (`tar.gz`).

Zip format is handled by the Info-Zip utilities available from (ftp://ftp.cdrom.com/pub/infozip, or mirror sites (the UK mirror is at: http://www.mirror.ac.uk/sites/ftp.cdrom.com/pub/infozip/. There are also various proprietary zip programs, of course.

FFTW is available from http://www.fftw.org/

The FFTW_Ada distribution consists of the 33 files listed in the table below, plus this manual, making 34 in all. The source code files use the ISO-8859-1 Latin 1 character set, but the actual Ada code (outside comments) is restricted to the ISO-646 subset (originally known as ASCII). The source and text files use the DOS end of line convention (CR + LF).

1	`gpl.txt`	GNU General Public License.
2 3	`fftw_ada.ads` `fftw_ada.adb`	Root package.
4 5	`fftw_ada-generic_thin_binding.ads` `fftw_ada-generic_thin_binding.adb`	Thin binding.
6 7	`fftw_ada-single_complex_types.ads` `fftw_ada-double_complex_types.ads`	Instantiations of complex types.

8
9
10
11
	`fftw_ada-single_thin_binding.ads`
	`fftw_ada-double_thin_binding.ads`
	`fftw_ada-thin_binding.ads`
	`fftw_ada-long_thin_binding.ads`
		Instantiations of the thin binding for C and Ada floating types.

12
13
14
15
16
17
18
19
	`fftw_ada-thick_binding.ads`
	`fftw_ada-thick_binding.adb`
	`fftw_ada-thick_binding-generic_1d.ads`
	`fftw_ada-thick_binding-generic_1d.adb`
	`fftw_ada-thick_binding-generic_2d.ads`
	`fftw_ada-thick_binding-generic_2d.adb`
	`fftw_ada-thick_binding-generic_3d.ads`
	`fftw_ada-thick_binding-generic_3d.adb`
		Thick binding.

20
21
22
23
	`fftw_ada-wisdom.ads`
	`fftw_ada-wisdom.adb`
	`fftw_ada-wisdom-file_io.ads`
	`fftw_ada-wisdom-file_io.adb`
		Wisdom.

24 25	`fftw_ada-real_time.ads` `fftw_ada-real_time.adb`	Access to the Ada real-time clock.

26
27
28
29
	`fftw_ada_test.ads`
	`fftw_ada_test.adb`
	`fftw_ada_single_test.adb`
	`fftw_ada_double_test.adb`
		Test code.

30	`fftw_ada_test_fftw_die.adb`	Tests fatal error handling.

31
32
33
	`make_rtc_header.adb`
	`rtc_test.c`
	`copyright.txt`
		Real-time clock header and test program.

FFTW_Ada distributed files are accompanied by detached PGP signature files (extension `.sig`). To verify these (and it is not essential to do so) you will need a version of PGP which supports DH encryption (older versions support only RSA), and my public key, which is given below (copy and paste it to a file and import into PGP) and is also available from key servers and from my web page at http://privatewww.essex.ac.uk/~sjs.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGPfreeware 6.0.2i

mQGiBDU3GkARBADyn6tgErL8GsO/+UkVIMmMR65FgaFhsUue+/90hIRBqRJGF0Xd
RbiW9dm8MNb/2vl2XAcuGxrvokWzdt1fullFJAUuwa/tKOWX8fXuThUjdfS0GpJS
saqlJf74z1R8FT+YTJe2JCOl4vk733ctUGsSYc9hvCgFRsu9e1d9Qvhe8wCg/7Zb
Pazd/WpLuqOFy1WiOzbuN4UEAOOQqo/3Z1/bZeEoCoQNIkSnHzybQLbaZlNlvhcY
IuslKshJT6Moi4ERPpSUJHRh/B+hHcDb8a0dFYdQ89v8EvSYmFW9hL7geH2Wh27z
4giDlTpIpsj+cbdbkp9JV7OYDjuFvblvr+T+r04+D9dUOjeFnVVkwyveYTHovwHh
W4ysA/4vMojMUw2M2hI2dzL6i+eL59przRP40s62G6tQUWvLw2+wlSGw4RncG0/p
xKNCm9Sry/EZimygFU8I6ymBWcZ6FLnjON2xyaVp+VV15nSFUhdA1xcSXY2y41zU
UtEg5706gy/K1On7D1nbrDN/yzjJZ+fLK61R5h3K5Icz6n7Sc7QrU3RldmUgU2Fu
Z3dpbmUgPFMuSi5TYW5nd2luZUBSZWFkaW5nLmFjLnVrPokASwQQEQIACwUCNYpQ
zAQLAwIBAAoJEEGGEV9WsZDtMdsAn1EcgHkoNWUqC6TUZPUQ3S/Az3vGAKDgOdXj
LbvPdh3j/UD8NoDvsqzKMbQpU3RlcGhlbiBKLiBTYW5nd2luZSA8Uy5TYW5nd2lu
ZUBJRUVFLm9yZz6JAEsEEBECAAsFAjU3GkAECwMCAQAKCRBBhhFfVrGQ7QygAKDC
SBp0U+00JiQBV6WHNWA4YrR6CQCgnliLaMqUuEHbvKs8KA3qs9LPkcu0JlN0ZXZl
IFNhbmd3aW5lIDxzanNhbmd3aW5lQGllZS5vcmcudWs+iQBLBBARAgALBQI1ilCg
BAsDAgEACgkQQYYRX1axkO3zJgCg8XMgd/+0+Us2zfsMDfYXdeiY9oQAoOvr7Xk5
k2OAw0qDU5TI7/5iVSAAuQINBDU3GkAQCAD2Qle3CH8IF3KiutapQvMF6PlTETlP
tvFuuUs4INoBp1ajFOmPQFXz0AfGy0OplK33TGSGSfgMg71l6RfUodNQ+PVZX9x2
Uk89PY3bzpnhV5JZzf24rnRPxfx2vIPFRzBhznzJZv8V+bv9kV7HAarTW56NoKVy
OtQa8L9GAFgr5fSI/VhOSdvNILSd5JEHNmszbDgNRR0PfIizHHxbLY7288kjwEPw
pVsYjY67VYy4XTjTNP18F1dDox0YbN4zISy1Kv884bEpQBgRjXyEpwpy1obEAxnI
Byl6ypUM2Zafq9AKUJsCRtMIPWakXUGfnHy9iUsiGSa6q6Jew1XpMgs7AAICB/9P
24ofRoqQVvyRv1julDbGThnmv7BJhxItoh5U1/MVksv9I6WktzgFWqMzSASoEzfs
tc2DSnnmKR9yIiX1jESFHHYkZE9ba6sPM1+de57p301isU6FaTcbcwHOv1lHXG0T
0xz4H4sBw0ZQ+3DzpMoXN248/BZWEaP96WyV1JGNEs9ijC4krDZKY8XwvGDWwc6E
t1XofqiOsR6+QkurMgdg9RrR20OlW4FEraiOn7/RMPFnlGAz/kcds7VwUuAfS1GK
zkyHDLH7NrE1Rtg4rLCHRbt++zIWN7ng7pMY0T3UwBEIQFR5+Xo2/2+MMOKpkYvX
jxu1OTEA+80diyGIGfOmiQBGBBgRAgAGBQI1NxpAAAoJEEGGEV9WsZDtLYkAnjDa
hdvZazzXkJ2ZUzNZpViY5AcqAJ988JCPvUTngaxYHvMx2V2LLdszYg==
=hhoI
-----END PGP PUBLIC KEY BLOCK-----
```

B Change history

Version 1.2 fixes a bug in the handling of interleaved transforms for the cases of 2 and 3 dimensions.

Version 1.1 fixed a bug in `FFTW_Ada.Wisdom.Export_Wisdom`. The code in earlier releases was incorrect and passed to `fftw_free` an incorrect pointer when freeing the memory used to pass the wisdom string to FFTW_Ada. The wisdom string itself was correctly passed to FFTW_Ada.

Version 1.0A included minor additions and changes. Two new instantiations of the thin binding were added.

Version 1.0 included a major restructuring of the code into a package hierarchy different to that used in earlier releases (0.9x), but no significant changes to the facilities provided. This change was introduced primarily because the previous hierarchical arrangement with nested generics was found to cause problems when compiled with version 3.13p of the Gnat compiler. This prompted a reconsideration of the structuring to try to factor out the generic aspects from the non-generic and thus minimise potential instantiation problems.

C Instructions for building FFTW with Gnat

This section is provided to assist users without experience of mixed Ada 95/C programming. It is based on the Gnat compiler. If you are using Gnat, regardless of platform, this should be of help. For other compilers you will have to refer to the compiler documentation.

1. Alter the file `config.h`. There are various flags to be defined according to the installation. For Gnat/Mingw32 the following should be defined:

   ```
   #define HAVE_MALLOC_H
   ```

   The following two definitions should be modified as shown. (To find the sizes, look at `limits.h`.)

   ```
   #define SIZEOF_INT 32
   #define SIZEOF_LONG_LONG 64
   ```

   Under Windows, comment out the line: `#define HAVE_WIN32_TIMER` to disable use of the high precision timer, if the file `windows.h` causes compiler errors.

2. Compile all the C files in the `fftw` directory. The command for this is:

   ```
   gcc -c -ansi -I. -O3 *.c
   ```

   The -I. indicates that the current directory is to be searched for the header files. Optimisation must be selected. This requires -O3. -ansi selects ANSI C enforcement. To compile the files in the `rfftw` directory the compiler must also search the `fftw` directory, so an additional switch is needed:

   ```
   gcc -c -ansi -I. -I../fftw -O3 *.c
   ```

3. Link the object files into a library. This is done with the `ar` program from the GNU Binutils collection (supplied with Gnat):

   `ar -rcs libfftw.a *.o` in the fftw directory
   `ar -rcs librfftw.a *.o` in the rfftw directory

   Option `c` is for create, `s` to tell `ar` to add an index table (this speeds up linker access).

4. Compile the test programs.

   In the directory `tests` compile and link the files `fftw_test.c` and `test_main.c`:

   ```
   gcc -c -ansi -I../fftw fftw_test.c
   gcc    -ansi -I../fftw test_main.c fftw_test.o ../fftw/libfftw.a
   ```

   In the directory `tests` compile and link the files `rfftw_test.c` and `test_main.c`:

   ```
   gcc -c -ansi -I../fftw -I../rfftw rfftw_test.c
   gcc    -ansi -I../fftw -I../rfftw test_main.c rfftw_test.o
            ../rfftw/librfftw.a ../fftw/libfftw.a
   ```

   Run the executable test programs and verify that all is in order.

5. Place the library files `libfftw.a librfftw.a` in a directory which is searched by the Gnat linker. (Your application directory will do, but if you are likely to use FFTW_Ada in more than one application, it is better to place them somewhere more general.)