The Big Online Book of Linux Ada Programming

i. Preface

've been working with Linux since kernel 0.97 and with Gnat since version 2.00. In the past five years or so, I've been frustrated by the lack of documentation on Linux Ada programming. Gnat is one of the most powerful development environments for Linux, certainly superior to C or C++, and yet most people have never heard of it. Those that have often ignore it because they can't find enough documentation to install Gnat, let alone to evaluate it.

After my article "Gnat: C++ and Java under Fire", published in the October 1998 edition of Linux Journal, I decided to collect my knowledge of Linux Ada programming and set down what I've learned: from installing Gnat to interfacing with the Linux kernel. I wanted to create a book that had everything I needed in one place to write professional Linux applications. After publishers declined to put it in print because Ada developers are a small (though growing) niche in the Linux market, I decided to publish it online so that the facts about Linux Ada programming would be understood.

This document covers basic software development on Linux, a review of the core Ada 95 language, and an introduction to designing programs that work with the Linux kernel and standard C libraries. It also covers some of the Ada bindings that exist for packages like Motif, TCL and GTK+.

This book tries to describe Linux specifics whenever possible. This is not another UNIX book recycled with the word "Linux" substituted in.

Although many Ada basics are covered, this document assumes the reader is familiar with a high-level programming language such as BASIC, C, C++, Java . Borland Delphi programmers will notice similarities between Delphi and Ada.

Because C is the dominant language in the Linux world, the differences between C and Ada are hilighed throughout the text.

The document is designed to be used as a reference after it's been read, with many tables and examples covering common Linux programming problems.

Although this book covers a lot of material, it is not intended as an exhaustive survey of Linux Ada programming.Linux is in a constant state of development. Refer to your Linux documentation for the latest information and newest features. Also, Ada 95 has several application specific and portability features which are not covered since they are not related to general Linux Ada programming.

Because of the fast pace of Linux development, information in this document may be obsolete, or (to paraphrase Douglas Adams) apocryphal or wildly inaccurate. However, most of the facts have been verified against Gnat 3.11 (or a later version), and most of the examples in this document have been compiled under Gnat.

Ken O. Burtch, September 1999

Rewrite & Expand] Ada 95 is arguably the most powerful development language available for Linux, with features comparable to Java and execution speeds similar to, and sometimes exceeding, C. GCC Ada (or Gnat), the main Ada 95 compiler for Linux, is also absolutely free. This makes a combination that's hard for Linux programmers to ignore.

1.1 A Brief History of Linux

The Linux operating system that was created as a hobby by a young student, Linus Torvalds, at the University of Helsinki in Finland. Linus, interested in the UNIX clone operating system Minix, wanted to create an expanded version of Minix with more capabilities. He began his work in 1991 when he released version 0.02 and invited programmers to participate in his project. Version 1.0 was released in 1994. The latest version is 2.4 and development continues.

Linux uses GNU General Public License (GPL) and its source code is freely available to everyone. Linux distributions, CD-ROMs with the Linux kernel and various other software ready for installation, do not have to be free, but the Linux source code must remain available. Making source code available is known as 'open source'.

The word "Linux" is properly pronounced using a Swedish accent, making it difficult to pronounce in North America. It is most often pronounced with a short "i" and with the first syllable stressed, as in LIH-nicks, but it is sometimes pronounced LYE-nicks (the anglicized "Linus' UNIX") or LEE-nucks.

Strickly speaking, Linux refers to the operating system kernel that starts and manages other programs and provides access to system resources. The various open source shells, compilers, standard libraries and commands are a part of another project called GNU. The GNU project was started by the Free Software Foundation (FSF) as an an attempt to create a free version of UNIX. The main Linux C compiler, gcc, is a part of the GNU project.

There is also a GNU kernel project, but this has been largely superseded by the Linux kernel.

X Windows is also not strictly a part of Linux. Xfree86, the free version of X Windows, was adapted to the Linux operating system.

1.2 1995: The Year of Ada and Gnat

In 1974, the US Department of Defense realized it was spending too much on software. They wanted a new computer language that could handle all of their needs, from controlling the hardware in a missile guidance system to doing artificial intelligence. In 1983, they created the language Ada (now known as Ada 83), a heavily modified version of the Pascal language. "Ada", a proper name, refers to Countess Ada Lovelace (1815-1852), considered by some to be the world's first programmer.

The original Ada had several shortcomings in the areas of software engineering: Ada programs tended to be big and awkward to maintain over time. In 1990, ANSI began a project to revise Ada, to include object oriented features, hierarchical program libraries, support for other languages, and add-ons for specialized applications like systems programming, real-time systems, distributed information (client/server) systems and scientific programming. The updated language is known as Ada 95.

GNAT is a GPL Ada compiler, available for Linux, Windows NT, and many other platforms. It was originally created at New York University. GNAT is owned by Ada Core Technologies (ACT, http://www.gnat.com): although gnat is free, companies who want support can purchase it for a fee. The Linux version of GNAT supports the entire Ada 95 standard, including all optional features. It includes many extensions, like cross-compiling and support for the C++ language. ACT also

provides GLADE, a free RPC-based TCP/IP networking implementation Ada 95's distributed systems annex.

The GNAT manual describes their compiler as "an industrial-quality Ada 95 compiler, integrated into the GCC retargetable compiler system. GNAT is a complete compiler, validated on several platforms, that includes support for all the Ada 95 annexes specified in the Ada Reference manual. Because of its integration into the GCC system, GNAT is available on a large number of hardware/operating system platforms, and can be used as a cross-compiler from any of its targets to any other one. Because of the common code-generator technology of GCC, GNAT has excellent support for multi-language programming: Ada, C, C++, Fortran, etc.

GNAT also represents a substantial improvement in Ada compilation technology. It's [sic] open-system philosophy stands in contrast with the opaque approach of older Ada compilers. There are no hidden and complex central libraries whose use requires a totally new set of commands, and no rigid development environments that often force needless recompilations. While preserving all of Ada's safety, GNAT's source-based model provides the flexibility and efficiency typically encountered in C development environments. Furthermore, GNAT's flexibility greatly facilitates its integration within third-party development environments and CASE tools. A number of standard editors, debuggers, profilers, memory analyzers, test coverage or configuration-management tools, etc. can be used with GNAT, which coexists comfortably with familiar programming tools (unlike older Ada compilation systems)."

Fun Fact: When Gnat 3.11p was released, Robert Dewar said that Linux would never be a billion dollar platform and deserved no special consideration by ACT. By the time Gnat 3.12p was released just over a year later, the Red Hat company was worth more than 18 billion dollars, or 40% of the server market. The first platform supported by Gnat 3.12p was Linux.

1.3 Why Use Ada?

C and C++ represent the de facto standard for Linux programming. After all, the kernel itself is written in C. However, C++ is not suitable for all kinds of projects because different computer languages have different strengths and weaknesses. Ada was designed for team development and embedded systems, leading to advantages over C in development time and debugging. An in-depth 1995 study by Stephen F. Zeigler (http://www.adaic.com/docs/reports/cada/cada_art.html) showed that development in Ada costs about half that of C++. It also suggests that Ada produces "almost 90% fewer bugs for the final customer".

GNAT was developed closely with gcc, the native C compiler for Linux. Unlike some compilers that translate a program into C and then feed the C program into gcc, gcc has built-in support for the Ada language. Like g++, the GNU C++ compiler, gnat works with gcc, allowing it to produce fast, quality executables without any intermediate steps.

This integration gives a lot of flexibility to programmers who want or need to support multiple languages. GNAT has an extensive set of features for trading variables and function calls between Ada and C/C++. It can import C/C++ items into Ada, export Ada items to C/C++. You can also link Ada functions indirectly into Java, using Java's ability to import C++ functions.

GNAT comes with over 140 standard libraries. These include numeric and string libraries, file operations, hash tables and sorts. If you would rather work directly with Linux C libraries, a variety of "binding" libraries exist, available for download from the Public Ada Library or The Home of the Brave Ada Programmers. These include bindings for POSIX (that is, the Linux kernel), X Windows, Motif, TCL and WWW CGI applications. The Ada Linux Team prepackage many bindings for use with their version of the Gnat compiler.

More and more Linux libraries feature Ada bindings, including ncurses (a standard text screen drawing library) and GTK (the Gimp Toolkit, a graphics package).

Although gnat is distributed under the GPL license, gnat and its libraries may be used in commercial applications.

The **GtkAda** mailing list is at http://gtkada.eu.org.

The **Gnat** mailing list is at http://www.diax.ch/users/gdm/gnatlist.htm.

The **Gnat Glade** chat mailing list is at glade-chat@act-europe.fr.

1.4 Why Ada and Linux?

Ada provides a number of important features for Linux programmers:

- **Fast Executables** - the GNAT compiler produces executables using the same code generator as gcc.
- **User Friendly** - Ada is easy to learn and use, making it a popular choice for introductory computer science courses. Its source code is much easier to read that C, C++ or Java.
- **Standardized** - Ada compilers adhere to a strict standard making Ada programs reliable and portable. Even Java hasn't been standardized.
- **Flexibility** - Ada has many specialized design features that address issues usually ignored by other languages, such as real-time applications, safety-critical software, and low-level hardware access.
- **Faster, Cheaper Development** - As the previously mentioned Zeigler study shows, Ada programs tend to have fewer errors that C++ programs. This means you can get your work done faster with less time and money spent on debugging.
- **Scalability** - Ada is designed for embedded systems and team projects, making it an ideal choice for large projects. This same scalability, and the object oriented features of the language, make the source code prone to a longer lifespan.
- **Ample Libraries** - The GNAT compiler comes with many general purpose libraries, and bindings exist for most of the key Linux libraries.
- **Open Source Friendly** - Ada's readability and scalability make it an ideal language for open source development. Child packages, for example, make it easy to extend someone else's work without affecting the original source code.

1.5 Linux Ada Resources

There are a variety of resouces on the Internet for Linux Ada development.

One important resource is the **comp.lang.ada** newsgroup, which is frequented by many Ada celebrities, including Robert Dewar of Ada Core Technologies and Tucker Taft, the principle designer of Ada 95. If you have questions about the inner workings of Ada 95, this is the place to go.

Ada Linux Team (ALT) is a group of programmers dedicated to Linux programming specifically using Ada. This site is located at www.gnuada.org/alt.html, running on a Sun/Linux machine hosted by Sam Tardieu. ALT provides the latest versions of software and libraries for Linux, including bug fixes for Gnat, prepackaged and ready for installation.

The **GNU Ada** site, http://gnuada.sourceforge.net (formerly www.gnuada.org), on how to install Gnat.

The **Ada Source Code Treasury** at www.adapower.com provides examples of both Linux and Windows Ada applications. Included are examples of sockets, MD5 encryption and packages to work with Windows servers from a Linux computer.It also has a free, unsupported binding to Motif by Itermetric/Avestar.

If you are looking for general algorithms and source code examples, **PAL** (the Public Ada Library), is a large source code repository located atwww.pegasoft.ca/pal/ada/pal.html . It includes thousdands of source code examples, bindings, compilers and the official Ada 95 documentation.

The **Ada Software Engineering Library** has over 1 Gig of files. It's available at http://unicoi.kennesaw.edu/ase/index.htm.

The **Home for Brave Ada Programmers** at www.adahome.com provides a lot of general reference material and bindings.

The **Ada Information Clearinghouse (AIC)** at www.adaic.com contains statistics, studies and other general information.

AdaCraft - http://www.it.bton.ac.uk/staff/je/adacraft/ and Ada Programming are online books on Ada.

Michael Feldman's Who's Using Ada List

http://www.skinner.demon.co.uk/aidan/programming/libra has a tool called Libra (Library of Reusable Ada Code) for many common data structures such as lists, queues, and Internet sockets such as HTTP and POP3.

http://www.ainslie-software.com has a tool called **AdaJNI** (Java Native Interface) that lets you call java methods from Ada 95.

2 Installating Gnat on Linux

GCC Ada was original called Gnat. Over the years it has undergone many changes, with different download sites, versions and patches. This chapter covers the various version, where to get them and how to install them.

GCC Ada is a part of the GCC (GNU Compiler Collection) project. The "gcc" command itself isn't a compiler: it's a program that determines the appropriate software to comple your source code file. The Ada compiler is called gnat1. The C compiler is cc1. When gcc detects an Ada source file, it runs the gnat1 program to compile it.

Because the gcc system and GCC Ada must work as a team, specific versions of Ada are created for specific versions of gcc. Gnat 3.10p was compiled against the gcc for Linux kernel 2.0.29 (for example, the version of gcc used in the Slackware 3.2 distribution). Gnat 3.11p, 3.12p and 3.13p are compiled against gcc 2.8.1.

 To find out which version of gcc you have, run gcc with the -v switch.

There are public (unsupported) and commerical versions of GCC Ada. **Prior to GCC 3.0**, the public version of Gnat was distributed separately from the GCC project and was available from ftp://cs.nyu.edu/pub/gnat. The GNU Ada Team (http://gnuada.sourceforge.net) release RPMs for Linux and other OS's. of Ada for different versions of Linux. Some people refer to these older versions as GNAT/GPL, to distingiush them from GNAT/GCC.

In addition, GNAT/GPL has a GPL restriction on the Ada Run-time library. The RTL provides much of the basic functionality of Ada and is required for most Ada programs to run--for example, Text_IO is in the RTL and switching off the RTL would prevent put_line("Hello World!") from running. This means that programs written in older versions of GNAT must conform to the GPL license. (I don't know for sure but this is probably an oversight by Ada Core Technologies since they have always insisted that GNAT can be used to develop non-GPL applications.) GNAT/GCC removed the GPL requirement from the Run-Time Library so that programs written in modern versions of GNAT are not restricted to a particular license.

The GNAT compiler source code is distributed with a license called GNAT- Modified GPL (GMGPL). This is an extension of GPL to allow open use of Generics/Templates which are not covered in the regular GPL.

With **GCC 3.0**, the public version of Ada is included in GCC and GCC Ada is available for most Linux distributions. If it is not included in a standard installation, it's usually available as an add-on package called "gcc-gnat", "gcc-ada" or a similar name. To test to see if you have Ada support, create a file called "hello.adb" and try compiling it with "gcc hello.adb". If gcc recognizes the file as an Ada program, then your copy of gcc has Ada support included.

The version included with **GCC 3.0** to **3.3** was Gnat 5 and was considered experimental. That is, it hadn't undergone rigorous testing and changes to the gcc system may introduce new bugs. **GCC 3.4** to **4.x** are considered stable.

Commercial supported versions (GNAT Pro) and academic versions of GCC Ada are available from Ada Core Technologies (ACT). They also have a software buildle called "GNAT/GPL" containing the Gnat compiler, various libraries (XML, CORBA, GTK, GLADE distributed processing) available through their web site.

You can check which version of Gnat you are running using like this:

```
$ gnatmake -v 2>&1 | head
```

```
GNATMAKE 4.0.2 20050901 (prerelease) (SUSE Linux)
Copyright 1995-2004 Free Software Foundation, Inc.
Usage: gnatmake  opts  name  {[-cargs opts] [-bargs opts] [-largs opts] [-margs opts]}
```

There are alternate ports of GCC Ada, such as AVR-Ada, a version for AVR 8-bit microcontrollers.

2.1 Installing GCC Ada on Linux

GCC Ada is included in most modern versions of Linux, although it may not be installed by default. Use your installation tool to search for "Gnat" or "GCC Ada". In some cases, it may be a part of the "extras" archive for your distribution (that is, software not included standard but available for download from the distribution's web site). Some examples:

- For Novell's **OpenSUSE 10**, Ada can be found by installing for "gcc-ada" and "libada" with YAST.
- Early versions of **Red Hat Fedora**, GCC Ada is automatically included. In later versions, it is packaged separately as "gcc-gnat".
- In Debian Linux, try installing "gnat-3.3" (or similar version number).

You can also look for installable versions GCC Ada at http://sourceforge.net/projects/gnuada.

Be aware that some versions of Linux include the old "GNAT" version of Ada (usually version 3.14p or 3.15p). These versions are based on the GCC 2 and are out-of-date. They are missing many modern features, such as support for the new Ada 2005 standard.

2.2 Building GCC Ada from Sources

Occasionally you may want to compile Gnat yourself from its sources. For example,

- you may want to learn more about computer language design
- you may want to enable support for other languages (e.g. C++)
- you may want to make libgnat a shared library
- you may want to upgrade gnat for the newest C libraries
- you may want to change the multithreading model

The source code for GCC Ada, of course, is a part of the GCC source code available at the GCC home page.

GCC Ada is a standard part of the GCC languages. However, GCC Ada requires GCC Ada in order to build a new version. Bootstrapping GCC Ada on a machine without an Ada compiler is beyond the scope of this book.

In simple terms, follow the directions to build GCC included in the GCC source code or from http://gcc.gnu.org/install/. When it is time to specify what languages you would like to build, include "ada" in the language list. (Even "all" will not include Ada--it must be requested explicitly. "all" represents the most common GCC languages.)

Rumour has it that GNAT 3.15p can be used to bootstrap GCC 4.1 or newer. You will need to run configure with "CC=adagcc configure" to do this. Older versions of GCC are not so GNAT 3.15 friendly.

2.3 Installing the old NYU Gnat

Ada versions 0.0 to 3.x are available for download from the NYU web site. The version refers to the version of the Ada compiler, not GCC. These all all built for GCC 2.x.

You may want to install the old public versions of Gnat on very old versions of Linux. The standalone Gnat distribution from ACT comes with its copy of the correct version of gcc and can install Gnat and its gcc in a separate directory. The binary versions from ACT's web site have C++ support removed, so if you want gcc to support C++ and Ada simultaneously, you'll have to recompile gcc and Gnat from their sources.

The ACT releases of GNAT include letter codes:

- "p" - a public release, with minimal support
- "w" - a wavefront release
- "a" - a commerical release, with full support

The version numbers have special meaning:

- Version 3 and under - normal ACT releases
- Version 4 - validated releases that are tested for Ada standards compliance. These are not available publically.
- Version 5 - GCC 3.x version included with most Linux distributions

 It is possible to install one version of gcc overtop of another and to select one version or the other using the gcc -V switch, but gcc must again be recompiled from its (newest) sources to make it aware of the other version.

There are patches available via the Ada Linux Team web site for compiling gnat from the sources for the egcs compiler instead of gcc. egcs (pronounced "eggs") is a variation of gcc designed specifically for Pentium computers. Egcs is based on gcc 2.8.0. Slackware 3.6, for example, used egcs. The egcs optimizations are obsolete with GCC 3.0.

The binaries on the NYU web site (http://cs.nyu.edu/pub/gnat) do not have the extra features available with the ALT RPMs, but they include extra installation information, including how to install Gnat's various add-ons. There are also versions for other operating systems besides Linux.

 Gnat 3.12 and older have an additional install option to overwrite you're existing copy of gcc, provided it is right version. Since it is rare that a distribution has the exact same version of gcc, this option is no longer provided.

ACT will sometimes release several versions of Gnat for different C libraries. When downloading the binaries, make sure that you download the version compiled against the appropriate C library. This is due to the constantly evolving nature of Linux.

 To find out which libc library your distribution uses, examine the /lib/libc.so link to find out which file it points to. For example, if /lib/libc.so points to a libc5 library, then you'll need the libc5 version.

For example, version 3.13p has been compiled for gcc 2.8.1. If you don't have gcc 2.8.1, you can specify a separate directory where gnat will install itself and its own personal copy of gcc 2.8.1. Using this method, you need to perform an additional step. The installation program (doconfig) creates a shell script containing environment variables that you can copy to your shell startup script (under bash, this is usually the .profile file in your home directory). Gcc uses these variables to locate the gnat files.

You will need to include the gnat directory in the front of your PATH variable to prevent gnat from using the gcc that came with your Linux distribution. For example, use the shell command:

```
export PATH="/usr/gnat/bin:$PATH"
```

Only use this command when you want to use Gnat since it effectively hides the copy of gcc that came with your distribution.

If you don't want gnat to be enabled by default, you can write a short shell script that assigns the environment variables, sets the path, and starts a new shell.

2.4 Installing the old ALT Gnat

The Ada Linux Team version of Gnat is available from their web site. Versions exist for the Red Hat, S.u.S.E. and Debian distributions. They may also work on the Mandrake and Caldera distributions.

Recently, the chief ALT maintainer has moved onto other interests and the ALT web site is out-of-date. Due to Gnat's integration into Gcc 3.x, there has been less interested in Gnat RPMS. However, there are still RPMs around hosted by other people (and usually based on the ALT system). Check out other sites such as ftp.ada95.com/pub/gnat-3.14p/RH-7/ or http://prdownloads.sourceforge.net/gnuada/.

The ALT versions include support for ASIS, GLADE and native Linux threads. The package includes gnatgcc, a version of gcc with Gnat and C++ support, and gnatgdb, a version of gdb that supports Ada source code, plus gnatprep and the other Gnat utilities.

 The rpm files are built for Red Hat and S.u.S.E. distributions. If you try installing it on another distribution, use --nodep to ignore any package dependency warnings.

RPMS for Version 3.15p for Red Hat 7.2 are available at http://www.alex.wubn.net/packages/3.15p/RPMS andhttp://www.alex.wubn.net/packages/3.15p/SRPMS.

1. Download and read the readme file.
2. Download the gnat-3.xxp-runtime* rpm file (where xx is the current version of Gnat and * is the rest of the filename). For older RPMs, this is gnat-3.xxp*.
3. Download the gnat-3.xxp rpm file. For older RPMs, this is gnat-3.xxp-devel*.
4. rpm -i gnat-3.xxp-runtime*
5. rpm -i gnat-3.xxp*
6. Download and install any of the additional Gnat packages you need

The rpm files on the ALT site are configured to work with the ALT version of gnat. To install them, simply download them and run rpm with the -i switch.

The ALT GNAT build system is available for those wanting to know more about how the RPMs are constructed. Using CVS, you can check our the source code.

```
export CVSROOT=":pserver:anoncvs@hornet.rus.uni-stuttgart.de:/var/cvs"
cd $HOME
cvs login  # (use empty password)
cvs -z9 co -d ALT gnuada/alt-build
```

2.5 Compiling Older Versions of Gnat from Sources

In order to recompile Gnat 3.x, you'll need the following:

1. A copy of the gcc sources in order to build a copy of gcc that's compatible with Gnat. The required version is listed in the Gnat documentation.
2. A copy of the gnat sources. The sources are available for download from the gnat download site and its mirrors.

[I should compile gnat and make notes and flush out the details more--KB]

First, you need to recompile the gcc compiler. Make sure you follow gcc's instructions for activating Ada support.

```
make CFLAGS="-g -fsigned-char" LANGUAGES="c c++ ada"
make stage1
make CC="stage1/xgcc -Bstage1/" CC="-g -O2 -fsigned-char" STAGE_PREFIX="stage1/" LANGUAGES="c c++
ada"
<build tools and lib with CC="./xgcc -B./">
```

There are two problems that can arise:

1. The standard C library may have changed.
2. The souce code for the gcc compiler itself may have changed.

Changes to the C library rare unless the library is several generations out of date. Even so, by consulting the man pages you can usually find out the new parameters the various C functions expect.

Upgrading gcc to a new version of gcc, however, can be difficult. Gnat's gcc patches are designed for a specific version of gcc. It is usually a good idea to get a copy of the source code for the version of gcc Gnat was designed for and compile a second gcc compiler just for use with Gnat. For gnat 3.13, you'll need the gcc 2.8.1 source code. You should be able to compile an older version of gcc to work with newer C libraries, provided the compiler is only a few months out of date.

Now follow the directions to compile Gnat. Make sure libgnat.a is accessible to the linker. If it isn't, copy it to /usr/lib and run ldconfig to update Linux's shared library tables.

2.6 Case Study: Installing Gnat 3.11 on over an old Linux Distribution

We installed Gnat 3.11p on a Pentium running a Slackware distribution with egcs and lib6. We wanted to replace egcs with gcc 2.8.1 and install the Gnat binaries (compiled for 2.8.1) over top.

We first went to the Sunsite mirror which provides Linux compiled binaries of gcc, ready to be unpacked and installed Unfortunately, the readme file reported they had trouble compiling gcc and supplied egcs instead. egcs is based on gcc 2.8.0 which meant that we couldn't use it with gnat 3.11. Instead we downloaded the gcc 2.8.1 source code from a GNU FTP mirror site and prepared to build the compiler from scratch.

1. We ran Gnat doconfig program and select option 1. The gcc path that it's expecting is displayed as i686-pc-linux-gnu. This was going to be our configuration host setting for gcc.
2. We followed the instructions in the gcc INSTALL file. configure --with-gnu-as --with-gnu-ld --enable-threads=posix --host i686-pc-linux-gnu
3. We checked the gcc makefile to make sure i686-pc-linux-gnu was reasonable. It required lib6 and lib6 was installed. The Makefile also showed that the i686 setting is compatible with our Pentium (i586).
4. Before running make, we changed the Makefile's OLDCC variable from cc to /usr/bin/gcc. There was a cpp syntax error while buildinglibgcc1.a, probably the error the Slackware people encountered. We tracked down the line causing the problem in the Makefile and discovered they were calling cc to do the compiling, which doesn't handle

the C preprocessor (cpp) properly. Typing in the line at the shell prompt showed that /usr/bin/gcc worked fine while /usr/bin/cc would not. The note in the Makefile said we shouldn't use gcc to avoid circular references in some of the functions (that is, that it might inadvertantly call the 2.8.1 compiler instead of the old 2.7.2.3 compiler), so we made sure we included the full path.

5. make LANGUAGES="c c++"
6. mkdir stage1; make stage1
   make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O2 -fsigned-char"
7. mkdir stage2; make stage2
   make CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O2 -fsigned-char"
8. make compare reported no errors.
9. make install CC="stage2/xgcc" -Bstage2/" CFLAGS="-g -O2" LANGUAGES="c c++"
10. gcc -v and gcc -dumpversion reported the correct version. We deleted the old /usr/lib/gcc-lib/i486-linux directory to save some space.
11. We installed gnat by running doinstall

If we were doing C++ programming, we would need to install the standard C++ library, libstdc++, as well. In this case, we enabled C++ support to avoid recompiling gcc for C++ in the future.

2.7 Gnat and Windows

Although this is a Linux book, here's some tips for Windows users:

Most Gcc uses use the Cygwin system which has a POSIX-compliant run-time system. This makes it easier to port and run programs from UNIX-like systems, including Linux. However, Gnat is based on Mingw and it uses the msvcrt runtime. (The reason for Mingw is, apparently, a problem with the Cygwin licensing agreement.) Unfortunately, because they have different run-time systems, Cygwin and Mingw are not compatible with one another.

If you want to build it on windows, you'll need the Mingw compiler.
Try http://www.mingw.org/download.shtml" and look for the "experimental/test packages" section. If you have Cygwin and want it to use Gnat at the same time, you'll modify the registry entries for Gnat and replace GCC for GNAT_GCC in key names/values. If you're willing to use the Mingwin version of Gcc for your C programs, put the Gnat/Mingwin path ahead of the Cygwin path in your environment PATH variable.

For those simply looking for a compiler to run at the DOS prompt, DJGPP (the Gcc for MS-DOS project) supports recent versions of Gnat.

ACT sells commerical support for Gnat on VxWorks, the Windows emulator for Linux.

Ada for .Net

A# is a product to generate .Net byte code for Ada. It supports Visual Studio 5. It's released under a GPL license.

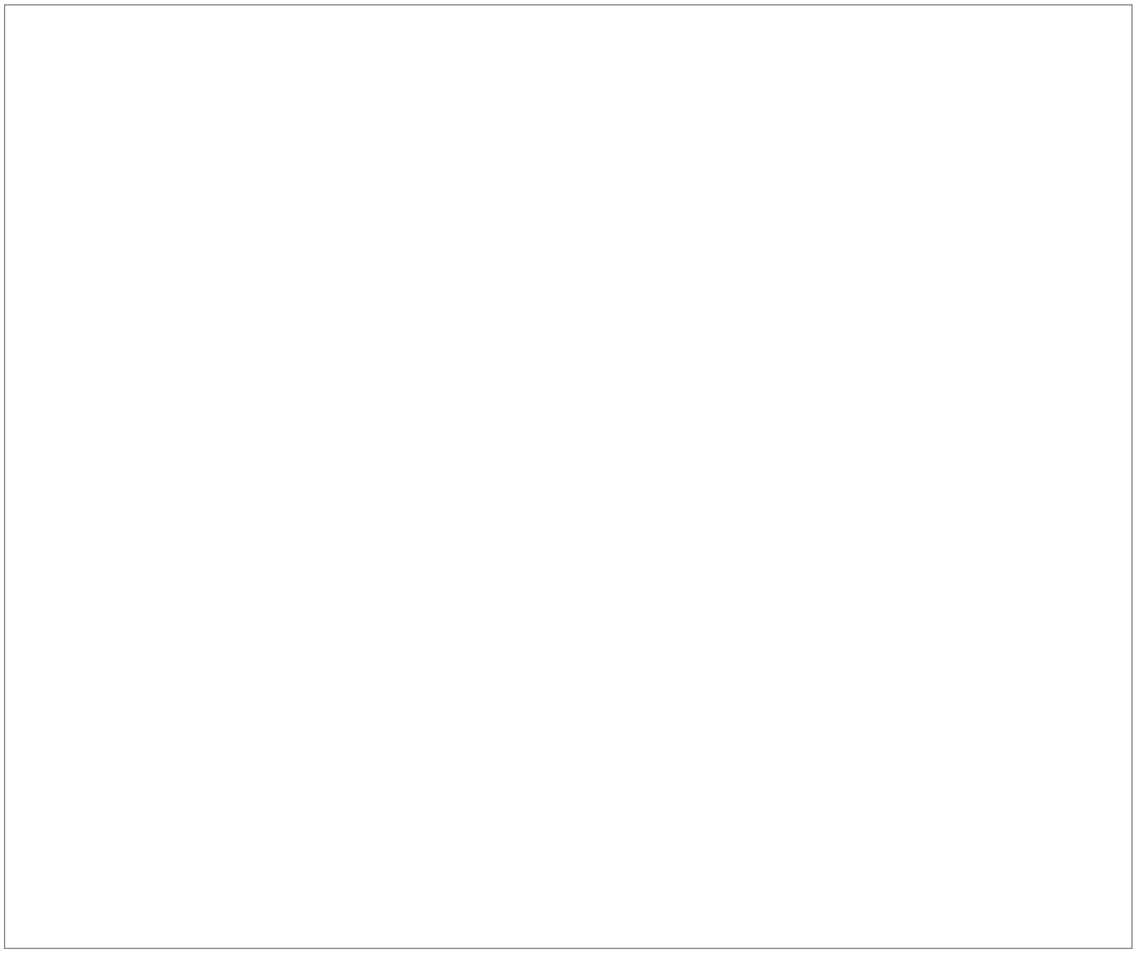
3 Introduction to the IDE's

There are two IDE's, or Integrated Development Environments, available for gnat and Linux. PegaSoft's TIA (Tiny IDE for Ada) is a text-based IDE, while GRASP is an X-Windows IDE. Both have similar basic features.

On the other hand, if you are looking just for text editors with Ada syntax hilighting, many exist for Linux including elvis, emacs/xemacs and nedit.

3.1 TIA: The Console IDE

TIA, Tiny IDE for Ada, is a console IDE for Gnat. Besides being my own program, it was written using Gnat runs using the GPL texttools packages described later in this document. The screen layout is similar to pico's, with the menu options displayed along the bottom of the screen. If you are running on the Linux console or a xterm window, you can choose the menu items with alt key combinations or using your mouse.



This IDE is designed for rapid Ada development.To meet this goal, it uses a number of interesting features:

- a **ddd** style debugger
- **automatic saving** - whenever you open a new source file, tia saves your old file
- **quiet updates** - each time a file is saved, TIA will attempt to recompile the file, to reduce the project rebuilding time. TIA will only update one file at a time to avoid slowing down your machine.
- **automatic spelling correction** - When you press return/enter while editing your source file, TIA automatically corrects common spelling mistakes for any of the following words or

phrases: procedure, function, package, exception, subtype, "end if;", "end loop;" , "end record;".

- **error hilighting** - you can move between compiler errors with a single keypress, and the cursor is automatically positioned at the exact location of the error and the message displayed at the bottom of the window.
- **quick open** - you can open recently opened files with a single keypress
- **tight integration with gnat** - for example, you can load a package spec and create a body using gnatstub by simply selecting Stub in the File menu.
- support for **keyboard macros**

If you are interested in an X Windows IDE, you should read the next section on GRASP.

3.1.1 Quick Start

Before you compile a program, you have to set up the project parameters under **Proj**. For simple, single file programs, put the name of the program in the main line and select your CPU type. TIA will save this information when you quit in a ".adp" file (Ada Project).

You can check the current file with **Check**. If there are errors, use Err to move to the place where an error occurred and the error message will appear on the bottom of the screen. Repeatedly use Err to fix all errors. Note that as you edit the program, such as adding or deleting lines, Err may not take you to the exact line because the lines have moved.

Build your project with **Build** and you're ready to run your program.

3.1.2 TIA Keyboard Legend:

These are the key functions in TIA. If you are running TIA under X Windows, the X window manager may use some of these key combinations for its own purposes.

Control Keys

Control-6 - Mark/Unmark
Control-A - Execute macro (follow with the key for the macro)
Control-B - Copy (single line or to the mark)
Control-E - End
Control-L - Redraw screen
Control-N - Page Down
Control-P - Page Up
Control-T - Backtab, back one item on screen
Control-V - Paste
Control-X - Cut (single line or to the mark)
Control-Y - Home

Navigating The TIA Screen

End - last line of text
Home - first line of text
Page Up - Up One Page
Page Down - Down One Page
Esc/F1 - TextTools' Accessories Menu
Tab - Next Item On Screen
Backtab - Previous Item On Screen (note: Linux console doesn't support the back tab key--use control-t)
Alt-*Char* - Jump to the item with hot key *Char* (Linux Console/xterm)

Scroll Bars Keys

Down Arrow - 10% Forward in Document End - Bottom of Document
Home - Top of Document
Left Arrow - Back one line
Right Arrow- Forward one line
Up Arrow - 10% Back in Document

In TIA, the width of the text is limited to size of the edit area. Any lines that are longer that the edit area are denoted with an ellipsis at the end. The edit area does not scroll left or right as it does in pico.

3.1.3 The File Menu

New Source

Start a new source file.

Open Source

Open a new source code window, or an existing one. Type in the name of the source file and choose open to open it. (.adb is assumed if you don't specify an ending.) Choose browse to walk through the directories using a open dialog box. Or you can chose one of the recently opened files that appear at the bottom. On the Linux console, use alt-# to open these files. Chose new and TIA will create an empty package body for you to fill in--just delete what you don't want.

Save

Saves the file. TIA automatically saves whenever you check or build.

Save As

Save As. Save the file under a different name.

Revert

Reloads the current file, discarding any changes that haven't been saved by you or TIA. (TIA automatically saves a file when a new one is loaded.)

Diff

Displays the differences between the current file and when it was last saved using
the diff command.

Print

Pipes the file, with a header, to the lpr command, printing it on the default line printer.

Stats

Display information about the current file and memory usage.

Stub

Creates an empty package body for the current file. The current file must be a package spec.

Check

Checks the current file for syntax errors.

Xref

Displays a crossreference of all identifiers in the current file.

Quit

Stops the program.

3.1.4 The Edit Menu

Cut

Deletes the selected text and puts it in the clipboard. Same as ctrl-x.

Copy

Copies the selected text to the clipboard without deleting it. Same as ctrl-b.

Paste

Inserts the text on the clipboard. Same as ctrl-v.

Append

Moves the cursor to the right end of the current line. This is useful for adding comments at the ends of lines.

3.1.5 The Find Menu

Find/Replace

Find brings up the find dialog to search for text. Fill in the top line and select find to find the next occurrence of the text in your document. Select backwards to search towards the top of the document instead of towards the bottom. Fill in the replace line and select replace to replace the text you are searching for with new text. Select cancel to erase the find text.

Next

Next finds (or finds and replaces) the next occurrence of the text in the source code.

If the text is not found, TIA beeps.

Next Err

Moves the cursor to the location of the next error and displays the error message at the bottom of the screen.

Goto

Moves the cursor to a specific line.

3.1.6 The Misc Menu

Edit Macros

Brings up the macro edit screen. Macros are keyboard short cuts you define. Each macro must fit on a line.The first character on the line is the trigger, and the remaining characters are the keyboard keys the trigger represents. For example, a line "pprocedure" defines a macro "p" that represents the keystrokes "p","r","o","c","e","d","u","r" and "e".

To use a macro in TIA, press control-A and then the letter of the macro.

Options

Opens the options window. The first option is to allow background updates. Turn this option off on slow machines. The second option sets the background colour to blue or black on colour displays.

Debugger

Runs TIA's ddd-style debugger.

[Expand--KB]

GDB

This item runs the gdb debugger.

3.1.7 The Project Menu

Project Params

The project parameters window. Choose the debugging level, CPU type and optimization level and TIA will pass the information to gnat accordingly. You can specify additional gnatmake options (like -n for no main program in Ada when you want to call Ada subprograms from another language), linking options (such as Linux libraries you need to link to), and the name of the main program. Static binding turns static binding on and off.

CPU Options: 486, Pentium, Pentium II, Other

Optimize Options: None, Basic, Size, Speed

Debugging Options:

- Prelease (assert/debug pragma's on, basic and elaboration checks on)
- Alpha/Beta (assert/debug pragma's on, basic checks on, no elaboration checks)
- Release (assert/debug pragma's off, all non-essential checks off)

Project Type:

- Program (compile and link project as an executable program
- Package (compile, but don't link project since there's no main program)
- Static Library ((unfinished) compile and generate a static library file named lib<project>.a)
- Shared Library ((unfinished) compile and link a shared library file named lib<project>.so.a)

Builder:

Specify the name of the program to build the project with, usually gnatmake:

- Gnatmake (Gnat's project builder)
- Make (Linux's standard project builder)
- Cook (an enhanced project builder based on make).

Static Linking: select this to link in all the libraries used into a self-contained executable

Egcs: select this to run egcs instead of gcc [untested]

ALT: select this to compile on a system using the ALT version of gnat

Build

TIA attempts to build the project and create a working executable file.

People

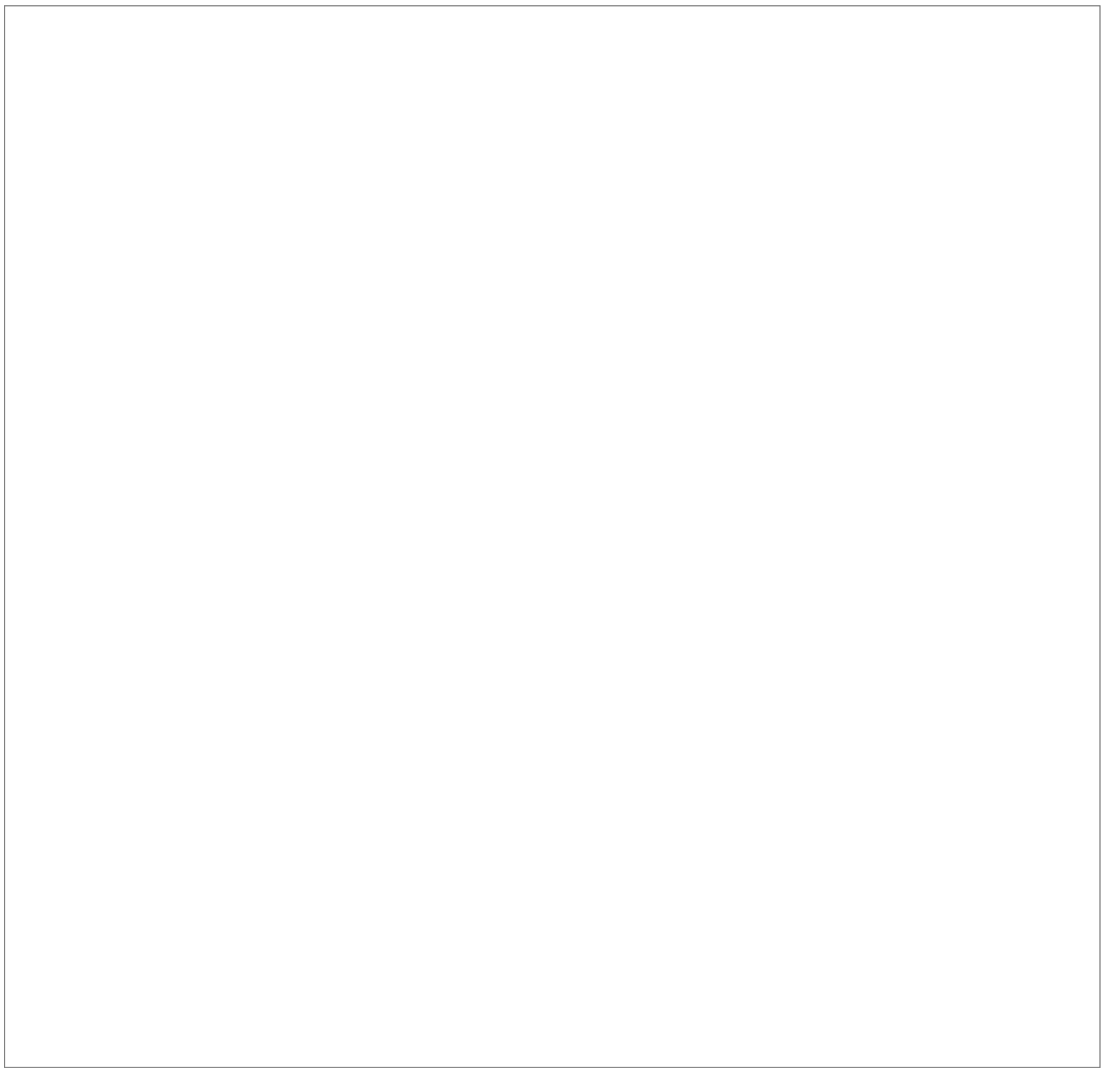
To be finished

3.1.8 The ? Menu

This is the About window. It shows information about the current version of TIA, including the version and copyright notice.

3.2 GRASP: The X Windows IDE

GRASP is a free X-Windows IDE that supports Ada 95. It's based on Motif and provides similar basic features to TIA. The main difference is that GRASP is a multi-language IDE and that it supports source code analysis, annotating your source code with Control Structure Diagrams (CSD's) and showing code complexity with graphs called CGP's.



Besides Ada 95, GRASP supports C, C++, Java and VHML source files. It supports operating systems other than Linux and can also work with Ada compilers other than gnat.

GRASP is available for download from that GRASP home page at http://www.eng.auburn.edu/grasp.

3.2.1 Installation

1. Download a version of GRASP from the GRASP web site. GRASP uses the Motif widget library. The static version has a copy of Motif included with it: download this version of you don't have a Motif compatible library (such as LessTif).

   If you have a Motif compatible library, download the dynamic version to save disk space. If you are using LessTif, make sure that the libXm and related files are properly linked in /lib and run ldconfig to ensure Linux sees the changes.

2. Move the tar archive to the location you want to install GRASP in. For example, "/usr/local" would be a good choice.

3. Unpack the grasp archive with "tar xfvz".
4. GRASP requires an environment variable called "GRASP_HOME" to be set so GRASP knows where it was installed. To define GRASP_HOME every time for any user, add the following line to the end of your /etc/profile file:

   ```
   export GRASP_HOME=graspdir/graspada
   export PATH="$PATH:$GRASP_HOME/bin"
   ```

   where graspdir is the directory where you installed grasp (eg./usr/local").

Login in again to make the changes take effect and type "grasp&" to start GRASP.

GRASP provides online help. Extensive documentation is provided in HTML format, but it's not lynx browser friendly. You'll have to use a GUI browser like Netscape.

3.2.2 Quick Start

1. Open a new source window using "File / Ada 95...".
2. Type in your program.
3. Save the source file as "test1.adb".
4. Add "test1.adb" to your project by chosing "File / Add to Project"
5. If you are using ALT gnat, change the compiler configuration under "Compiler / Command Setup":
   1. Select Compiler / Command Setup. You'll see the list of commands Grasp uses to invoke gnat.
   2. Change the Compile and Check commands from "gcc" to "gnatgcc", the name used by ALT.
6. Compile the program by chosing "Compiler / Compile and Link". If there were no errors, the message window will appear with the message "Grasp: operation completed" and no error messages above it.

3.2.3 The Project Window

The first window that appears is the GRASP project window. It contains the word "GRASP", but as you add source files to your project, they will be added to this window.

The File Menu

Ada 95 et al. - opens a new source file window. Choose a language: Ada 95, C, C++, Java or VHDL.

Save all files - saves all open files.

Exit GRASP - quits GRASP, closes all open windows.

The Project Menu

New - start a new project

Open - open an exisiting project

Close - close current project

Save - save current project

Create / Save as - save current project under a new name, creating a new project.

Open Selected File- opens hilighted source file in the project window

Remove Selected Files - deletes these files from the project window

Add Files to Project - add source files to the project window

The Search Menu - search for files to add to the project

The Preferences Menu

Colour/Font - change the appearance of the source code window

Tab Size - change the number of spaces to indent when the tab key is pressed

Generate .gml files - NQS--KB

The Window Menu

Message Window - opens the GRASP error message window.

Search Window - opens the file searching window

3.2.4 Source File Window

This is the window where you type up your Ada programs and packages.

The File Menu

Clear - erase the entire source file

Open - open a new source file

Save - save the source file

Save as - save the source file under a new name

Print - prints the source file to a postscript printer, or to a file to be printed with ghostscript

Language - change the source file language. This doesn't translate the file to a new language, but tells GRASP how the file should be hilighted.

Exit- closes the source file window

The Edit Menu

Undo - undoes last edit change

Cut, Copy, Paste - standard cut, copy and pasting text

Paste Primary - NQS--KB

Search / Replace - standard search /replace

Comment - turns hilighted text into a comment

Uncomment - removes comment marks from hilighted text

Convert Keywords to Upper/Lowercase - changes the case of all keywords in the document.

Goto Line - goto a particular line

Insert File - insert a source file into this one.

The View Menu

Show Unit Symbols - toggles module symbols in CSD diagram

Show Data Symbols - toggles data symbols in CSD diagram

Show Boxes - toggles boxes around multiline statements in CSD diagram

Intrastatement Align - If on, for statements longer than one line, the second line and onward will be indented to the position of the first open paranthesis.

Force Newlines - If on, divides up statements so they will be on separate lines.

Auto Line Numbers - toggles line number display in CSD diagram

Auto Indent - toggles indentation in CSD diagram

Line Numbers - adds line number to CSD diagram corresponding to lines in source code

Generate CSD - creates a Control Structure Diagram for source code

Remove CSD - removes CSD diagram from source code

Show Controls - toggles button bar at top of edit area

Show message Bar - toggles message bar at bottom of window

The Templates Menu - insert source code for typical Ada multiline statements. The source code must be edited to fit the current program.

The Window Menu - opens various grasp windows

The Compiler Menu

Make - builds the project using the make command defined in command setup (typically make)

Compile and Link - builds the project using gnatmake

Compile - compiles the source file without building the project

Semantic Check - checks the source code for errors

Flag Setup - configures the switches for compiling, linking, etc.

Command Setup - configures the commands to compile, link, etc.

The Run Menu

Run - runs the program using popen

Run Previous - runs the last file that was run

Run File - runs a particular file

Cleanup Session - kills any hung processes.Doesn't affect daemons.

The CPG Menu

Generate CPG - create a Compilexity Profile Graph

Weights - configure the CPG weights

3.2.5 The Button Bar

Auto - automatically rebuild the CSD whenever a major change occurs, such as inserting a template or loading a new file.

Generate CSD - same as choosing View/ Generate CSD

Font Size - changes the font size

3.3 Other Tools and IDEs

3.3.1 VAD-Visual Ada Developer

VAD is an Ada code generator written in the TCL/TK graphics scripting language. Besides gnat, there are about 10 additional packages you must install before VAD will run. You type in a TCL/TK description of a VAD widget in a text file and VAD will produce all the necessary source code to

use the widget. Most graphics formats are supported. VAD is available from http://websamba.com/guibuilder.

3.3.2 Jessie

Jessie is an X Windows IDE, primarily written in Java, for building large projects and designing multiple executables at once. It's an open source project of Silicon Graphics (http://www.sgi.com) and works with multiple languages. ACT has announced that Gnat will provide Jessie support in the future. Jessie is downloadable from http://oss.sgi.com/projects/jessie.

3.3.3 Rapid

RAPID is an X Windows GUI Builder that works with TASH (the Ada TCL/TK package). You can draw TCL/TK windows containing Labels, Text Buttons, Radio Buttons, Check Boxes, and other widgets. When you select "Compile", RAPID saves the Ada source code necessary to display using TASH the window you drew. RAPID is available from ALT.

3.3.4 VIDE

VIDE is a C/C++/Java IDE, but it will work with Gnat if configured correctly. However, it doesn't support Ada keyword hilighting.

3.3.5 GLIDE

Not complete

3.3.6 AdaGIDE

AdaGIDE (the Ada GUI Integrated Development Environment) is an Ada IDE for Microsoft Windows. It includes debugger support, a text editor and a code formatter. It supports Ada .Net.

For information on AdaGIDE, visit http://adagide.martincarlisle.com/index.html.

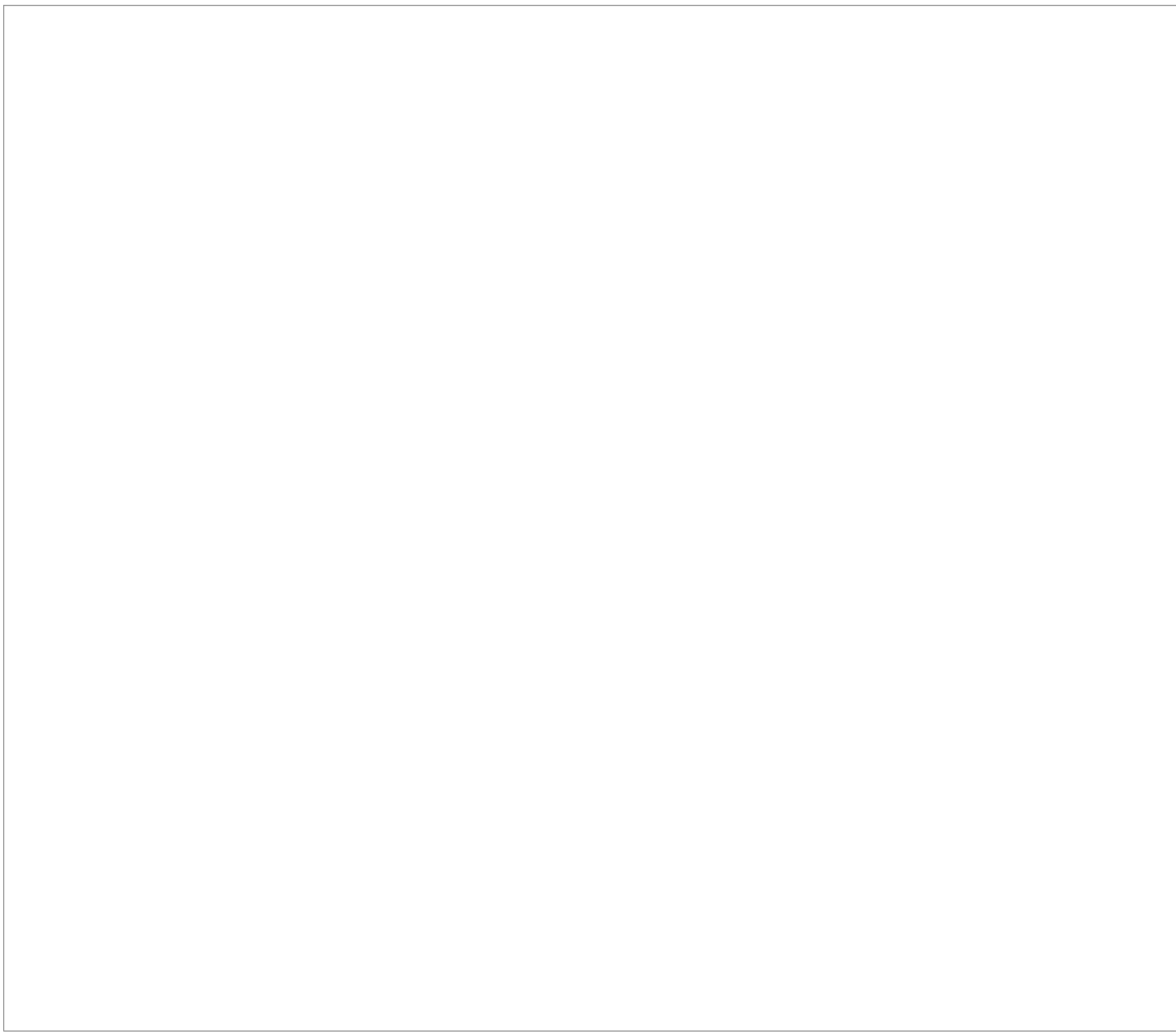
3.3.7 AdaBrowse

AdaBrowse is a HTML documentation generator for Ada 95 library unit specs, similar to what javadoc does for Java. It's based on ASIS. For information, seehttp://home.tiscalinet.ch/t_wolf/tw/ada95/adabrowse/.

3.3.8 KDevelop - KDE's IDE (and it supports Ada)

Kdevelop, standard with most versions of Linux, supports Ada.

When you start a new project, select Ada and the simple Hello World project for a demonstration.



3.3.9 Ada on Eclipse IDE

ACT sells a commerical product called GNATbench for the Eclipse IDE.

Ada for Netbeans is a IDE based on Java's NetBeans IDE.

3.4 BUSH (AdaScript Business Shell)

BUSH is PegaSoft's *Business Shell*. BUSH is a powerful shell and scripting language designed around the POSIX standard and a scripting extension of Ada called AdaScript. BUSH contains many built-in subprograms including database access and network sockets and is very compatible with Gnat. Combined with Gnat, BUSH is part of the ABEE scalable enterprise initiative providing an end-to-end Ada solution for business.

The BUSH home page is located at http://www.pegasoft.ca/bush.html

4 From Source Code to Executable

This section is an overview of creating new programs on Linux.

4.1 Gnat Filename Conventions

Unlike Microsoft Windows, Linux filenames do not require a suffix to indicate the filetype. Nevertheless, Linux files often have suffixes to make it easier to identify the type of files by their names. gnat makes extensive use of suffixes. Here are some filename conventions:

- .ads - Ada package specification
- .adb - Ada package body or program
- .adc - Gnat configuration file (for dead code elimination)
- .adt - Gnat tree file (for dead code elimination)
- .adj - Defaults for [NQS-KB]
- .adp - TIA project file or Gnat gnatxref/gnatfind project file
- .cfg - GLADE distributed program configuration file
- .ali - debugging and linking information produced by gnat
- .xrb - cross-reference file generated by gnatf
- GRASP_defaults... - GRASP defaults file, holds your preferences
- .gpj - a grasp project file
- .gui - a VAD TCL/TK widget description

For example, demo.adb would be an Ada program named demo.

 You can change the colour used by `ls` to display these filenames by changing the `/etc/DIR_COLORS` file. Directions on how to do this are included in the file.

4.2 Writing Your First Ada Program

4.2.1 Writing a program with an IDE:

Start a new project. For example, with TIA type

```
tia hello
```

to create a new project called "hello.adp". Type in the following Ada program.

```
with text_io;
use text_io;
procedure hello is
begin
    put_line( "Hello World!" );
end hello;
```

This Ada program will print a short message on the screen. Save the file and build the project.

If there is a problem with your program, TIA will show you the line number, the position on the line, and an error message describing the problem. If you typed in the program correctly, you should now have an executable file called hello in your directory. TIA will ask you if you want to run your program.

Your program should display the message

```
Hello World!
```

before TIA's window reappears.

4.2.2 Writing a Program without an IDE

With a standard UNIX editor such as pico or vi, create the following file called "hello.adb". For example,

```
pico hello.adb
```

Type in the following Ada program.

```
with text_io;
use text_io;
procedure hello is
begin
   put_line( "Hello World!" );
end hello;
```

This Ada program will print a short message on the screen. When the file is saved, the gnatmake command to build the project:

```
gnatmake hello.adb
```

If there is a problem with your program, gnatmake will show you the line number, the position on the line, and an error message describing the problem. If you typed in the program correctly, you should now have an executable file called hello. Run the file by typing

```
hello
```

(or ./hello on some distributions) and Linux will respond by displaying the message

```
Hello World!
```

4.2.3 After Building

After building your project, there should be several files in your directory:

- hello.adb - this is the Ada program you typed in. The is called a *source file*.
- hello.o - this is the binary code created by the compiler. This is called an *object file*.
- hello.ali - this is additional information about the program created by gnat.
- hello - this is the executable program

If you want to clean up your directory, the hello.o and hello.ali are information files and can be safely erased. However, on large project with multiple files, leaving them will speed up the building process.

4.3 The Three Step Process

When you build a project using gnatmake, or when you use an IDE to run gnatmake for you, gnatmake performs three operations:

1. **Compiling**: gnatmake checks your file for errors. If there are no errors, it creates an object file containing the binary version of your program. If there are any errors in your file, gnatmake stops.
2. **Binding**: gnatmake verifies that all the files in the project are up to date. If there are files that need to be compiled, gnatmake will compile them as well.

3. **Linking**: gnatmake combines all the object files to create an executable program.

On simple projects, these steps can all be done automatically. However, on some projects with particular requirements, you may need to take special actions during one of these steps. You can perform these separate steps yourself. For example, using the hello.adb program:

1. Compile the program with: gcc -c hello.adb
2. Bind the program with: gnatbind hello.ali
3. Link the program with: gnatlink hello.ali

Once again, you have an executable program called "hello".

4.4 Gnat Compiling Options

The version of gcc for gnat has all of the normally document gcc switches, plus some new switches for gnat. You can run gcc by itself, or have gnatmake run gcc for you. Unless otherwise noted, these switches can be applied to both gcc and gnatmake.

- **-b** - For crosscompiling, compile for a target machine
- **-Bdir** Multiple gnats, load the gnat compiler from directory dir instead of the default one
- **-c** - gcc only, tells gcc to compile only and not to try to link with the C linker
- **-g** - create an executable that can be used with the gdb debugger
- **-Idir** - Beside the directory with the first file, check directory dir for more source files
- **-I-** - Do not look for source files in the directory where the first file resides
- **-On** -On Optimize, from 0 (none) to 3 (maximum, automatic internal inlining). See below.
- **-s** - gcc only, create an assembly language source file instead of an object file
- **-Wuninitialized** - warnings on uninitialized variables
- **-v** - show what steps the gcc compiler is performing
- **-gnata** - turn on debugging pragmas. See below.
- **-gnatb** - keep messages brief
- **-gnatc** - check the program, but don't compile it
- **-gnatd*x*** - activate ACT internal debugging switch 'x', where x is a character
- **-gnatD** - with -gnatG, save debugging info to files ending in .dx
- **-gnate** - display error message immediately instead of waiting until end of compile (versions prior to 3.14)
- **-gnatE** - turn on dynamic elaboration checks
- **-gnatf** - give more information about errors
- **-gnatg** - turn on gnat style checks
- **-gnatG** - show pseudo-code of how Gnat interprets your source code (for internal use by GNAT)
- **-gnati*c*** - use character set c
- **-gnatk*n***- constrain file names to n characters
- **-gnatl** - include source code with error messages
- **-gnatL** - C++ exception handling (setjmp/longjmp)
- **-gnatm*n*** - show no more than n errors
- **-gnatn** - allow inline subprograms across source code in different files
- **-gnatN** - inline as much as possible, even subprograms not marked for inlining
- **-gnato** - turn on checks normally turned off (such as numeric overflow checking)
- **-gnatp** - turn all checks off
- **-gnatq** - don't quit because of errors--compile entire source file
- **-gnatr** - check for reference manual source code layout

- **-gnatR** - listing with alignment info
- **-gnats** - synatx check only
- **-gnatt** - create tree output file
- **-gnatu** - list units being compiled
- **-gnatV*x*** - change level of validity checks to n (none), default (d), full (f).
- **-gnatw*m*** - warning mode(s) m. These include
    - **-gnatwa** - show all optional warnings
    - **-gnatwA** - show no optional warnings
    - **-gnatwc** - warnings for always true/false expressions in statements
    - **-gnatwe** - treat warnings as errors
    - **-gnatws** - suppress warnings
    - **-gnatwl** - warnings on elaboration errors
    - **-gnatwu** - warnings on unused variables, uninitialized parameters, unused packages
    - **-gnatyk** - check indentifier case
- **-gnatx** - suppress cross-reference information
- **-gnaty** - Impose line length limit, etc. [? - KB]
- **-gnatzm** - generate distribution stubs for m
- **-gnatZ** - zero-cost exceptions (default)
- **-gnat83** - enforce old Ada 83 conventions
- **-gnat95** - enforce Ada 95 conventions (default)
- **-mno-486** - create an executable that can run on a Intel 386 or newer
- **-m486** - create an executable that can run on a Intel 486 or newer
- **-mcpu=*model*** - compile an executable for the given cpu model
- **-fstack-check** - required for checks for stack overflows. If you are experiencing core dumps on large arrays, try turning on this switch to raise a STORAGE_ERROR exception instead

The -gnat switches can be combined together, such as -gnatbcs for -gnatb, -gnatc, and -gnats.

If your operating system environment has a variable called **ADA_OBJECTS_PATH**, use this like to list directories containing Ada files built into operating system libraries (like .a or .so files). Use **ADA_INCLUDE_PATH** to specify directories containing .ads files for these system libraries. These can be combined with -L or -I switches: the variables and switches have the same function.

Many of the GCC switches listed in 4.5 can be used as well.

To use -gnatN or -gnatn:

- -O1 or better is required
- The subprogram that will be inlined must be small
- The subprogram must not have nested subprograms (gcc restriction)
- The subprogram must appear in a package body

4.4.1 Run-time Error Checking

Ada has extensive checking for run-time errors. By default, gnat turns off some of these checks to improve the speed of the programs. To turn on all error checking, you need to use **-gnato -gnatE** switches. To turn off all error checking, you need to use **-gnatp**.

IDE: TIA sets these switches for you based on your choices in the project parameters window.

4.4.2 Checking without Compiling

In gnat 3.10, if you want to check a source file without actually compiling it, use the **gnatf** utility. In gnat 3.11 or later, you can use **gcc with the -gnatc** option to check a source file.

IDE: TIA uses -gnatc when you chose File/Check.

4.4.3 When you have Too Many Errors

When you have so many compiling errors that they run off the top of the screen, you can redirect the errors to a file and list them with the less command by adding the following to the end of your compiling command: "2> temp.out; less temp.out".

4.5 Gnat Binding Options

gnatbind checks the integrity of a project before the linking phase. You can run gnatbind by itself, or have gnatmake run it for you.

- **-A** - (default) generate binder program in Ada. See -C, -x.
- **-aIdir** - besides the directory of the file, search for source files in directory dir
- **-aOdir** - besides the directory of the file, search for .ali files in directory dir
- **-b** - brief messages
- **-C** - generate binder program (in C, not Ada). See -A, -x.
- **-c** - check only
- **-e** - list elaboration dependancies
- **-E** - exception stack traceback (when compiling with -funwind-tables)
- **-f** - use full reference manual semantics in an attempt to find a legal elaboration order
- **-h** - help
- **-Idir** -combination of -aI and -aO
- **-I-** - don't look for source and .ali files in regular places
- **-l** - list the chosen elaboration order
- **-mn** - show no more than m binding errors
- **-M*n*** - main program to be called n, not the default name
- **-n** - no main program, for when the main program is written in another language
- **-nostdinc** - (no standard includes) ignore default directory when looking for sources
- **-nostdlib** - (no standard libraries) ignore default directory when looking for libraries
- **-o file** - output to a file name file
- **-O** - list objects
- **-p** - pessimistic - try worst case elaboration order
- **-r** - renames the main program from main to gnat_main
- **-s** - require all source files to be present
- **-shared** - link in Gnat run-time library as a shared library (if available--for example, ALT version)
- **-static** - link in Gnat run-time library statically
- **-t** - ignore time stamp errors
- **-T*n*** - tasking time slices are n milliseconds long. n=0 means no time slicing, as per Annex D.
- **-we** - treat warnings as errors
- **-ws** - suppress all binder warnings
- **-v** - verbose messages
- **-x** - check only, ignore source files. Don't generate a binder program. See -s.
- **-z** - same as -n [?-KB]

4.6 Gnat Linking Options

gnatlink combines object files together to form a finished executable program. You can run it by itself, or gnatmake can run it for you.

- **-o *file*** - name of executable file to create
- **-v** - verbose messages
- **-gnatlink n** instead of gcc, use linker named n
- **-l *lib*** in the specified library

 When linking in libraries, the order of the libraries is important. When libraries depend on each other, libraries must be listed before the libraries that they use.

4.7 Gnatmake Options

To compile any Ada program, use the **gnatmake** command. gnatmake checks all the packages a program relies upon and automatically compiles any packages that need compiling. For example,

```
gnatmake main.adb
```

will compile the file main.adb, automatically compiling all Ada files referenced by main.adb, if necessary. This is unlike other building tools like make and cook because the dependancy of source files is listed in every Ada file by the **with** statement. make and cook are designed to work with C which has no equivalent statement and requires the programmer to list the dependencies in a separate file.

When gnatmake is finished compiling, it will automatically bind and link the program, producing an executable file called main.

There are times when you want gnatmake to compile the project, but not to bind or link it. You can tell gnatmake not to link by using the -n option:

```
gnatmake -n main.adb
```

Here is a summary of the gnatmake switches:

- **-a** - consider all files, even read-only source files and standard system files like ada.text_io
- **-c** - compile only
- **-f** - recompile entire project
- **-jn** - on multiprocessor machiens, compile using n processes at once
- **-k** - ignore errors, and compile as much as possible
- **-M** - create a list of dependences suitable for make's Makefile
- **-i** - instead of the current directory, keep intermediate files in directories where their sources are found
- **-m** - update dependancies without compiling
- **-n** - check dependencies, but don't do anything
- **-o name** - save the executable as name
- **-q** - quiet - no status messages
- **-s** - switch change - recompile if the switches have changed
- **-u** - compile only indicated file. Don't remake whole project.
- **-v** - verbose - explain why files are compiled
- **-aIdir** - besides the directory of the first file, search directory dir for source files
- **-aOdir** - besides the directory of the first file, search directory dir for object and .ali files
- **-Adir** - same as -aIdir and -aLdir

- **-Idir** - same as -aodir -aIdir
- **-I-** - don't look for source files in the directory of the first file
- **-Ldir** - besides directory of the first file, look for libraries in directory dir
- **-cargs s** - pass switches s to compiler
- **-bargs s** - pass switches s to binder
- **-largs s** - span>pass switches s to linker (e.g. -largs somefile.o to link in the somefile object file)

4.7.1 So you changed the comments...

Use **gnatmake** with the **-m** option, which updates gnat's files without producing a new object code file. Use this to avoid pointless recompilations when all you changed were the comments in a source file.

4.7.2 Gnatbl: Bind and Link

In certain cases, such as mixing different languages with Ada, you may need to compile the source files with different compilers but the binding and linking can be done in one step. **gnatbl** is a shortcut tool that runs gnatbind and gnatlink.

4.8 Project Management

In many cases, Gnat needs no "makefile" to build itself. The Ada source code includes all the information about how the packages of a project are related to one another. Recent versions of Gnat support project files (with the gnatmake -P switch). Project files contain additional information about how a gnatmake should build a project outside of the information contained in the basic Ada source code.

The project files are laid out in an Ada style, using a "project" block:

project p [extends *p2*] is
...

Some clauses the appear are:

- for Main use *unit*;
- for Exec_Dir use *path*;
- for Source_Dirs use *path-array*;
- for Object_Dirs use *path-array*;

4.9 Linker Pragmas

In addition to the project file mentioned in 4.8, there are pragmas that embed linking options into a source file so that they don't have to be listed on the command line.

for Default_Switches(") embeds command line switches into your source code. For example, *for Default_Switches("c") use ("-mmcpu=pentium");*. The use portion is a list of comma-separated strings. Likewise, use the langauage "ada" to add Ada switches: *for Default_Switches("ada") use ("-v"); --verbose*.If the switch has double quotes, embed the quotes in the string use two quotes or concatenate an ASCII.Quotation character.

[More to be written]

4.10 AdaControl Source Code Checker

AdaControl is a free, GNAT-based tool that detects the use of various kinds of constructs in Ada programs. Its first goal is to control proper usage of style or programming rules, but it can also be used as a powerful tool to search for use (or non-use) of various forms of programming styles or design patterns. Searched elements range from very simple, like the occurrence of certaine entities, declarations, or statements, to very sophisticated, like verifying that certain programming patterns are being obeyed.

AdaControl can be downloaded from http://www.adalog.fr/adacontrol2.htm.

5 Building Large Projects

5.1 Make: The Traditional Project Builder

IDE: TIA supports make. To use make, select it in the project parameters window.

Gnatmake is the best tool for building small projects. However, if you have a lot of C functions, you may want to use Linux's traditional project building command, **make.**

The make command interprets a series of rules saved in a file called "Makefile". These rules describe which files are dependent on which other files. Each rule is followed by the command needed to update the files, such as the command to compile them.

For example, if you had a Ada program called dbase and it relied on the source files common.adb, scanner.adb and parser.adb, a Makefile might include the rule:

dbase: common.o scanner.o parser.o

    gnatlink —o dbase

This rule says that the dbase executable file depends on the object files for the 3 Ada source files, and to update dbase make has to link the object files with the gnatlink command.

If you are writing an Ada program with C source files, the basic strategy for using make with gnat is to make rules than ensure the C files are compiled properly, and then to finish the project using Gnatmake.

Makefiles can have comments and variables and rules that refer to parameters to the make command as opposed to files. The many options can't be covered here.

5.1.1 A Simple Ada Makefile

The following Makefile will compile an Ada program called main.adb, plus any packages used by main. This should work for most small projects. Edit the OBJS variable to include the object files for every package used by your program.

To use this make file, type "make" to build your Ada project, or "make clean" to remove any intermediate files produced by the compiler.

 The ALT version of Gnat uses uses the name gnatgcc, not gcc, for the GCC compiler.

```
# Sample Ada makefile
#
# Assumes main program is named main.adb
#
# by Ken O. Burtch
OBJS = main.o somepackage.o
# How to compile Ada files
.adb.o:
    gcc -c $<
.SUFFIXES: .adb .o
# How to link the main program
main: $(OBJS)
```

```
    gnatbind -xf main.ali; gnatlink main.ali
clean:
    rm *.o *.ali core
```

5.2 Cook: A Parallel Make

IDE: TIA supports cook. To use cook, select it in the project parameters window.

cook is a program for building projects. Unlike make, cook has additional features such as the ability to define true variables and functions, and the ability to build a project using multiple machines in parallel. This can be a useful tool for large Ada projects.

cook also comes with a tool to convert Makefiles to cook Howto.cook files.

If you need to install cook, there are four basic steps are:

- configure # run GNU configure
- make # build cook
- make sure # verify cook by running tests
- make install # install cook on your computer

When you type "cook", cook looks for a file called Howto.cook. This file, called a "cookbook", contains rules, or "recipes", for building a project.

Each rule has three parts:

- **targets** - the files built with this rule (e.g. the object file names)
- **ingredients** - the files need to do the building (e.g. source code fields)
- **body** - the commands to do the work. these can include other rules.

Here is an example Howto.cook file with one rule:

```
main: main.adb
{
    gnatmake main.adb;
}
```

The target is "main". To create main, cook must examine main.adb. If main.adb is newer than main, cook creates a new main by running the Gnatmake command.

Cook comes with some predefined rules for compiling certain kinds of files. These predefined cookbooks can be attached to your Howto.cook file by using "#include". For example,

```
#include "c"
```

includes basic rules describing the relationships of C files to each other.

5.2.1 Cooking in Parallel

Cook can use the rsh command to build several parts of a program at once, either on a computer with multiple CPU's or over a network. If rsh hasn't been configured, you'll need to do this before you can run cook. For example, all the computers should have the source directory mounted via NFS under the same mount directory. Also, the clocks on all the computers should be set identically or cook may be confused by the age of the files.

To cook in parallel, run cook with the -par switch. This option indicates the number of computers cook can use, the default being 4. You can indicate fewer computers. For example, -par=2 will run cook using 2 computers.

To indicate which machines to use, assign the hosts to the parallel_hosts variable.

parallel_hosts = first_computer second_computer third_computer

Simple Howto.cook files will compile in parallel without any modification.

5.2.2 A Simple Ada Cookbook

The following Howto.cook file will compile an Ada program called main.adb, plus any packages used by main. This should work for most small projects. Edit the OBJS variable to include all the object files from the packages you are using.

To use this make file, type "cook" to build your Ada project, or "cook clean" to remove any intermediate files produced by the compiler.

 The ALT version of Gnat uses uses the name gnatgcc, not gcc, for the GCC compiler.

```
/* ---------------------------------------- */
/* This is a simple Ada Howto.cook cookbook */
/*                                          */
/* it assumes the main program is named     */
/* main.adb                                 */
/*                                          */
/* by Ken O. Burtch                         */
/* ---------------------------------------- */

OBJS = main.o somepackage.o;

/* How to compile individual Ada files */

%.o: %.adb {
    gcc -c %.adb;
}

/* How to compile individual C files */

/* ( Just in case we want to mix C and Ada ) */

%.o: %.c {
    gcc -c %.c;
}

/* How to bind and link the main program */

main: [OBJS] {
    gnatbind -xf main.ali;
    gnatlink main.ali;
}

/* How to clean up intermediate files */

clean: {
    rm *.o *.ali core;
```

}

There are many other features in cook not covered here. More information about cook can be found in the Cook User Manual and the Cook Reference Manual.

5.3 Automake and Autoconf: UNIX Portability

If you want to create projects that run on a variety of UNIX platforms, not just Linux, you'll want to look at GNU autoconf and automake. The GNU tools use these programs extensively.

Included with Linux, **autoconf** creates a shell script called "configure". Customized for your project, when this script is executed, it scans the features of the particular UNIX that it is running on and tailors all Makefiles accordingly. It optionally produces a C file called "config.h" which contains information about the features it found.

automake, the other half of autoconf, creates a Makefile.in using templates called "Makefile.am". Once automake is finished, all you have to do is run "configure" to make your final makefile and type "make" to build the project on any version of UNIX.

It is possible to use autoconf and automake on Ada Makefiles, but this topic is beyond the scope of this book. More information on automake and autoconf can be found using "info autoconf" and "info automake".

The following is an example of what happens when you run an autoconf configure script, as run for the FreeAmp program.

checking host system type... i686-pc-linux

checking for a BSD compatible install... (cached) /usr/bin/install -c
checking whether build environment is sane... yes
checking whether make sets ${MAKE}... (cached) yes
checking for working aclocal... found
checking for working autoconf... found
checking for working automake... found
checking for working autoheader... found
checking for working makeinfo... found
checking whether make sets ${MAKE}... yes
checking for gcc... gcc
checking whether the C compiler (gcc) works... yes
checking whether the C compiler (gcc) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking for c++... c++
checking whether the C++ compiler (c++) works... yes
checking whether the C++ compiler (c++) is a cross-compiler... no
checking whether we are using GNU C++... yes
checking whether c++ accepts -g... yes
checking for POSIXized ISC... no
checking for ranlib... ranlib
...
checking for libc5... no
checking for dlopen in -ldl... yes
checking for MIT PThreads... no
checking for Base LinuxThreads... yes
checking for LinuxThreads w/ErrorCheck Mutex... yes

checking for sys/asoundlib.h... no
creating ./config.status
creating Makefile
...
creating config/config.h

5.4 PRCS: Project-wide Source Control

PRCS, the Project Revision Control System, is a source control system designed for large projects. It is easier to use and requires less administration than CVS, especially when a large number of files is involved. For example, PRCS (almost) always performs all actions across the entire project while CVS commands act only on your current position: if you are in a subdirectory, CVS only updates the subdirectory while PRCS will back up to the root directory for the project before performing any work. PRCS also handles file permissions while CVS does not.

Like CVS, PRCS uses a directory called a *repository* to store project files. The repository is directory is specified in a PRCS_REPOSITIORY environment variable.

PRCS uses project files, *.prj*, to manage sets of files at once. Use the project file to control PRCS's behaviour, such as listing file names or extensions that you don't want to add to a project.

Basic command include

- prcs checkout *project* - check out a project
- prcs populate - search for new files and add them to the project
- prcs checkin - commit the changes to the repository to share them with the team
- prcs depopulate *project files* - remove files added to a project
- prcs admin - perform administration functions such as setting file permissions

You can find out more about, and get the latest version
here: http://www.xcf.berkeley.edu/~jmacd/prcs.html

6 Development Utilities

6.1 Saving Time with Gnatstub

Starting with gnat 3.11p, gnat provides a prototyping tool called **Gnatstub**.Gnatstub takes an Ada package specification and creates a corresponding body, ready to have the details outlined in the spec filled in. These empty subprograms are sometimes called "stubs".

This is especially useful on a large project where programmers write a series of package specs to test their design. Once the package design is set, Gnatstub can create a basic body and save the programmers the work of copying and modifying the specification by hand.

IDE: TIA will run gnatstub on the current file using Stub in the File menu.

For example, suppose you have the following package specification in a file called tiny.ads:

```
package tiny is

   procedure simple_procedure;
   function simple_function return boolean;

end tiny;
```

You can create a stub body for this package using

```
gnatstub tiny.ads
```

Gnatstub produces the following tiny.adb file:

```
package body tiny is

----------------------
-- simple_function --
----------------------

function simple_function return boolean is
begin
   return simple_function;
end simple_function;

-----------------------
-- simple_procedure --
-----------------------

procedure simple_procedure is
begin
   null;
end simple_procedure;

end tiny;
```

This package body is in proper Ada format, ready to be compiled. Of course, it doesn't actually do anything useful. It's up to the programmer fill in the implementation details.

6.2 Cross-referencing with Gnatxref

Gnatxref (or gnatf in gnat 3.10) is a utility that produces an index of every occurrence of an identifier in a program, including all identifiers used by packages that the program depends upon.

The -v option produces a listing in the format for a vi editor tags file.

IDE: TIA will run gnatxref on the current file by chosing Xref in the File menu.

For our hello.adb program, Gnatxref produces the following:

```
Text_IO U a-textio.ads:51:13 {} {hello.adb:1:10 4:7 }
Put_Line U a-textio.ads:260:14 {} {hello.adb:4:15 }
Ada U ada.ads:18:9 {} {hello.adb:1:6 4:3 }
hello U hello.adb:2:11 {} {}
```

Each line begins with the identifier being indexed. The "U" means [not sure-KB]. The next segment is the file that defines the identifier, and the position in the file. The {} means [not sure-KB]. The final bracketed section lists all occurrences of the identifier in the program.

In this example, identifier text_io appears in the first line (the with) and the fourth line (the put_line).

6.3 Eliminating Dead Code with Gnatelim

gnatelim is a utility that searches for unused parts of your program in the object files and removes them from the final executable. It works by creating a list of subprograms that the compiler shouldn't compile. If you save this list as gnat.adc, gnatmake will automatically read this file and will skip these subprograms when compiling.

To use gnatelim, you need to generate tree files using the -gnatt switch. The Gnat manual recommends these steps when using gnatelim (assuming that your main program is main.adb):

1. gnatmake -c main
2. gnatbind main
3. gnatmake -f -c -gnatc -gnatt main
4. gnatelim main > gnat.adc
5. gnatmake -f main

These commands will generate a complete set of tree files for your project, strip out all unused subprograms, and will then recompile the project as a finished executable.

gnatelim is is based on ASIS.

[gnatelim doesn't work under gnat 3.11.--KB]

6.4 Execution Stack & Memory Leak Detection

Gnat 3.11 does not display the execution stack in the event of of an exception. Gnat 3.12 provides additional information about the source of an exception. You can get additional information about the execution stack using the gnat.traceback package (12.15).

The **gnatmem** utility monitors a running programming using the Gnat gdb debugger. When the program is finished running, gnatmem displays a summary of dynamically allocated memory. You can use this information to find "memory leaks", places in your program where allocated memory was not deallocated. Because gnatmem uses gdb, the program should be compiled with gdb support turned on (the -g switch).

Note: gnatmem doesn't work with Gnat 3.11.

To run gnatmem, type

```
gnatmem program
```

The gnatmem switches are:

-q - quiet - hides statistics and shows only potential memory leaks
n - a number between 1 and 10 indicating the depth of the backtrace information
-o *file* - save the gdb output to the indicated file. The gdb script is saved as gnatmem.tmp
-i *file* - processing using the file previously saved with **-o**. Use this to test a program that crashed while gnatmem was running.

6.5 Conditional Compiling with Gnatprep

Although the Ada 95 design team decided against including preprocessing compiler directives like C does, gnat provides a preprocessor so you can use conditional compiling directives in your Ada programs.

Gnatprep, the Gnat PREProcessor, takes a source file with conditional directives, a file with variable assignments for the conditional directives, and produces a source file with all statements not satisfying the conditional directives removed.

C: The conditional directives do not allow expressions. There must only be a variable, and that variable must be true or false.
IDE: No IDE's currently support gnatprep.

Suppose you create a file called "prepvalues" with the following Gnatprep definitions:

```
ALPHAVERSION := true
BETAVERSION := false
RELEASEVERSION := false
TRANSLATION := English
```

Suppose also that you had a short program with Gnatprep statements in it:

```
with text_io;
use text_io;

procedure preptest is

   -- only include the relevant parts for this version
   #if ALPHAVERSION
   s_version : string := "alpha";
   #elsif BETAVERSION
   s_version : string := "beta";
   #elsif RELEASEVERSION
   s_version : string := "release";
   #else
   s_version : string := "unknown";
   #end if;

   -- string is the value of the gnatprep variable named translation
   s_translation : string := "$TRANSLATION";

begin
   Put_Line( "This is the " & s_version & " edition" );
   Put_Line( "This is the " & s_translation & " translation" );
end preptest;
```

Running gnatprep on the above program with the prepvalues file gives you the following program:

```
with text_io;
use text_io;

procedure preptest is

   -- only include the relevant parts for this version

   s_version : string := "beta";
   -- string is the value of the gnatprep variable named translation
   s_translation : string := "English";

begin

   Put_Line( "This is the " & s_version & " edition" );
   Put_Line( "This is the " & s_translation & " translation" );

end preptest;
```

The Gnatprep command switches are:

-Dsymbol=value - define values on the command line instead of a prep values file, same as -D in C. For example, -DMacintosh=FALSE
-b - replace gnatprep commands with blank lines (instead of -c)
-c - comment out gnatprep commands (instead of -b)
-r - generate a Source_Reference pragma
-s - print a sorted list of symbols and values
-u - treat undefined symbols as if they were FALSE

C: There is no gnatprep equivalent of __FILE__ (name of current source file) or __LINE__ (number of current line).

6.6 Profiling with Gprof

Gprof in the GNU profiler. It shows which subprograms in a program are being executed the most. You can use this information to find the parts of a program with the greatest need for CPU efficiency and hand optimize those parts accordingly.

To use Gprof, you must rebuild your project using the **-pg** switch at both the compiling and linking stages. With Gnatmake, you must include -pg with both -cargs and -largs switches.

IDE: TIA will profile your project with gprof if you select Profile in the Project menu. It automatically rebuilds your project with the necessary gprof switches, starts your main program, and then displays the gprof results.

For example, we can use Gprof on the following factorial program:

```
package fact is
   function factorial( param : integer ) return integer;
end fact;

package body fact is

   function factorial( param : integer ) return integer is
   begin
      if param < 2 then
           return 1;
      end if;
```

```
        return param * factorial( param - 1 );
    end factorial;

end fact;

with fact;
use fact;
procedure bench2 is

    maxFactorials : constant integer := 1000;

    type factorialArray is
    array( 1..maxFactorials ) of integer;

    list : factorialArray;

begin

    for i in 1..maxFactorials loop
        list( i ) := factorial( i );
    end loop;

end bench2;
```

After compiling and linking the program with the -pg switch, run the program. The program produces a gmon.out file containing profile information about the program. Now we can use Gprof to get an analysis of the program.

Running gprof -c bench2 returns the following information. Note that Ada subprograms are labeled with the package name, a double underscore, and the subprogram name.

[Need to clean this up--KB]

Flat profile:

Each sample counts as 0.01 seconds.

%cumulative self self total

time seconds seconds callsus/callus/callname

62.500.050.05__mcount_internal

25.000.070.02mcount

12.500.080.01100010.0010.00fact__factorial

0.000.080.0010.00
10000.00_ada_bench2

0.000.080.0010.000.00fact___elabb

%the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted

seconds for by this function and those listed above it.

self the number of seconds accounted for by this

seconds function alone.This is the major sort for this listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/callfunction and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 100.00% of 0.01 seconds

index % time self children called name

499500fact__factorial [1]

0.010.001000/1000_ada_bench2
[3]

[1]100.00.010.001000+499500fact__factorial
[1]

0.000.000/0mcount (177)

499500fact__factorial [1]

0.000.000/0_start [473]

[2]100.00.000.01main [2]

0.000.011/1_ada_bench2 [3]

0.000.000/0__gnat_initialize
[399]

0.000.000/0adainit [61]

0.000.000/0__gnat_break_start
[395]

0.000.000/0adafinal [60]

0.000.000/0__gnat_finalize
[397]

0.000.000/0exit [95]

0.000.011/1main [2]

[3]100.00.000.011_ada_bench2
[3]

0.010.00 1000/1000fact__factorial
[1]

0.000.000/0mcount (177)

0.000.001/1adainit [61]

[6]0.00.000.001fact___elabb
[6]

0.000.000/0mcount (177)

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines.The line with the
index number at the left hand margin lists the current function.

The lines above it list the functions that called this function,
and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children.Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a

cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the function into this parent.

children This is the amount of time that was propagated from
the function's children into this parent.

called This is the number of times this parent called the
function `/' the total number of times the function
was called. Recursive calls to the function are not
included in the number after the `/'.

name This is the name of the parent.The parent's index
number is printed after it.If the parent is a
member of a cycle, the cycle number is printed between
the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the child into the function.

children This is the amount of time that was propagated from the
child's children to the function.

called This is the number of times the function called
this child `/' the total number of times the child
was called. Recursive calls by the child are not
listed in the number after the `/'.

name This is the name of the child.The child's index
number is printed after it.If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)

The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

(418) __mcount_internal[1]
fact__factorial[5] <cycle 2>

[3] _ada_bench2(177) mcount

[6] fact___elabb[4] <cycle
1>

If factorial is a function internal to program bench2, the function name won't show up in Gprof. For example:

```
procedure bench2 is

   function factorial( param : integer ) return integer is
   begin
      if param > 1 then
         return param * factorial( param - 1 );
      end if;
      return 1;
   end factorial;

   maxFactorials : constant integer := 100;
   type factorialArray is
      array( 1..maxFactorials ) of integer;

   list : factorialArray;

begin

   for i in 1..maxFactorials loop
      list( i ) := factorial( i );
   end loop;

end bench2;
```

gprof -c bench2 returns the following information:

Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated

%cumulativeselfselftotal

time seconds seconds callsTs/callTs/callname

0.000.000.00990.000.00main

0.000.000.0010.000.00_ada_bench2

etc.

Note: gnatmem doesn't work with Gnat 3.11.

6.7 Static Libraries and Shared Libraries (GnatDLL)

To create static libraries (.a suffix under Linux), create the archive file by combining all the .o object files into an .a archive file using ar. Then, put the .ads and .ali files in a appropriate directories. Change the .ali files to read-only. Gnat will then search for the object files in a library. When building your project, use the appropriate switches (-I, -L, -l, etc.) so that gcc and find your library and the include files.

To create shared libraries, Gnat has a tool called **gnatdll** to automate the process in a operating system independent way.

[To be completed]

6.8 Source as Web Pages Using GnatHTML

This utility converts an Ada source file into a series of indexed, coloured HTML web pages. By default, the web pages are stored under a subdirectory called HTML. By posting your source code on a network or the Internet, developers can examine your work with a web browser.

The gnathtml switches are:

- **-83** - look for Ada 83 keywords only
- **-cc** *color* - comment colour
- **-d** - convert files which depend on this file as well
- **-D** - like -d, but also convert standard library files
- **-f** - crossreference local entries
- **-l***n* - display line numbers every *n* lines
- **-I***dir* - file search path
- **-o***dir* - specify the output directory (default html/)
- **-p***file* - use this Gnat project file
- **-sc** *color* - symbol colour
- **-T***file* - read the files to convert from this file

Example using gnathtml with the following program:

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure htmltest is
  -- a demonstration of gnathtml

    function factorial( n : natural ) return natural is
  -- compute the factorial of n
  begin
    if n < 2 then
      return 1;
    else
      return n * factorial( n-1 );
    end if;
  end factorial;

-- main program

begin
  put_line( "Browse this source!" );
  new_line;
  put_line( "The factorial of 5 is" & natural'image( factorial( 5 ) ) );
end htmltest;
```

This creates an index file like this:

Files

- htmltest.adb

Functions/Procedures

- factorial
- htmltest

The first link would show you your entire file:

File : htmltest.adb

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure htmltest is
   -- a demonstration of gnathtml

   function factorial( n : natural ) return natural is
   -- compute the factorial of n
   begin
      if n &lt; 2 then
         return 1;
      else
         return n * factorial( n-1 );
      end if;
   end factorial;

-- main program

begin
   put_line( "Browse this source!" );
   new_line;
   put_line( "The factorial of 5 is" & natural'image( factorial( 5 ) ) );
end htmltest;
```

The links use the .htm (not .html) extension for portability.

6.9 GnatFind

[To be written—KB]

7 Optimizing Your Project

Optimization is the customization of a program to run as small and/or as fast as possible on a particular type of computer.
If you program is running slower than you expected, or is using more memory or disk space than you expected, you should first examine the approach you used in your Ada source code. Can you use better data structures, or implement faster algorithms? For example, a bubble sort is an easy way to sort relatively small amounts of data, but a quick sort is faster on thousands or millions of pieces of data.

In large programs, the subprogram causing the biggest bottlenecks may not be obvious. Experimenting with different test data and timing the results can often narrow down the problem areas. You could also try the gprof profiling tool, which will give you statistics on your program performance and will show that you are on the right track. Why spend hours or days improving a section of your program that isn't causing the problem? This is especially important in a business environment: focus your time on the sections that will give the greatest improvements.

Some optimizations can be done automatically by the Gnat compiler. There are both compiler switches and language pragmas for fine tuning your programs.

7.1 Compiler Optimization Options

There are several compiling switches used to optimize programs.

The -O switch tells the compiler how much time it should spend optimizing the program:

- **No -O** - fastest compiling, but gives you gnat warnings for optimization pragmas. Use only when fast compiling is essential.
- **-O/-O1** - slower compiling, cleaner executable, and no pragma optimize warnings from gnat. You should normally include this.
- **-O2** - more optimization. Use this for the smallest executable.
- **-O3** - full optimization, automatic code inlining for small subprograms and loop unraveling. Use this for the fastest executable.

When using floating point numbers, you may experience rounding errors if you don't use the -ffloat-store switch as discussed in 8.5.

Inlining is also affected by two other switches:

- **-gnatn** - allow inlining between packages where pragma inline is used in a package specifications.
- **-gnatN** - allow automatic inlining between packages (this is very memory intensive)
- **No -gnatn/N** - no inlining between packages, even if a pragma inline is used in a package specification.

These switches both require a -O switch for inlining to take effect.

The **-gnatp** switch turns off all non-essential error checking such as constraint and range checks. This is the same as using pragma Suppress(All_Checks) on every file in the entire program, making the program smaller and faster.

There are some other gcc optimization switches which can sometimes be used:

-ffast-math - gcc will ignore certain ANSI & IEEE math rules. For example, it will not check for negative numbers before invoking the sqrt function. This improves math performance but can cause side- effects for libraries expecting the ANSI/IEEE rules to be honoured.

-mfpmath=sse - turn on SSE extensions (for Intel chips). This improves the speed of array operations.

-fomit-frame-pointer - gcc will free the register usually dedicated to hold the stack frame pointer. This improves performance, but makes debugging difficult--many debugging utilities require the frame pointer.

-unroll-loops - gcc will make some looping statments (while, for) run faster.

IDE: TIA sets the proper switches for you based on your selections in the project parameters window.

7.2 Gnat Source Optimization Options

Ada Package	*Description*	*C Equivalent*
`pragma Pack( Aggregate );`	Use minimum space for the aggregate.	-
`pragma Optimize( Space / Time / Off );`	How you want your statements optimized.	-
`pragma Inline( Subprogram );`	Inline the subprogram	`inline`
`pragma Inline_Always( Subprogram );`	Inline the subprogram	–
`pragma Discard_Names( type );`	Don't include ASCII identifiers in executable.	-
`pragma Suppress( All_Checks );`	Turns off all checking	-
`pragma Convention( Fortran, array );`	Store arrays in inverse order	-

There are seven pragmas available to change the size and execution speed of your program.

Pragma **Pack** compresses an array, record or tagged record so that it uses the minimum space possible. For example, a packed boolean array takes up one bit for each boolean. Pack only packs the aggregate, not any aggregate items that might make up the aggregate: if you have an array of records, you'll need to both pack the array and the records to use the minimum space possible. Packing aggregates usually slows down the execution of your program.

```
type CustomerProfile is record
   Preferred : boolean;
   PreordersAllowed : boolean;
   SalesToDate : float;
end record;
pragma Pack( CustomerProfile );
```

Gnat can perform close packing, that is, packing right down to individual bits, for array elements or records of 64 bits or smaller.

Pragma **Optimize** specifies how you want your statements to be optimized: to run as fast as possible (time), to be as small as possible (space), or no optimization at all. Optimize does not affect data structures.

```
pragma Optimize ( space );
package body AccountsPayable is
```

Pragma **Inline** makes Ada inline the subprogram whenever possible. That is, it physically inserts the subprogram whenever it's named instead of calling it in order the make your program run faster. This uses up a lot of space and is only practical for small procedures and functions.

```
procedure Increment( x : integer ) is
```

```
begin
   x := x + 1;
end Increment;
pragma Inline( Increment );
```

Compiling switch -O3 must be used or pragma inline is ignored. -O3 will also automatically inline short subprograms for you.

Pragma **Inline_Always** forces inlining between packages (like -gnatn) regardless of whether or not -gnatn or -gnatN has been used.

Pragma **Discard_Names** frees up space by discarding the ASCII images (names) of identifiers. For example, if you have a big enumerated type, Ada normally maintains strings for the names of each of the enumerated items in case you want to use the 'img attribute. You can discard these names if you never intend to use 'img.

```
type aDogBreed is (Unknown, Boxer, Shepherd, MixedBreed );
pragma Discard_Names( aDogBreed );
```

When you discard names, the 'img is still available. Instead of returning the enumerated value's image, 'img returns the position of the enumerated type (for example, 0, 1, 2 and so forth).

Fun Fact: The ASCII images of your variable names are stored as C strings at the end of your executable file. You can view them using the less (or strings) shell command.

Pragma **Suppress** turns off all checking which improves performance but will also make debugging issues more difficult. Unlike -gnatp, the pragma can be applied over sections of the source code.

Fortran stores arrays in the reverse order of most other languages. It as been reported that, for some arrays, this improves hits in the operating system cache. Luckily, Ada can store an array in Fortran order using pragma **Convention**.

Other optimization tips:

- Avoid using unconstrained types
- Use the newest version of GCC Ada possible
- Try compiling with -S and examine the assembly code
- Create your own libraries. Many of the standard Ada libraries are designed for portability and features, not speed, and may not be optimized for your machine. By creating a custom library, compiled with settings for your CPU, basic jobs like I/O can greatly improve performance.

7.3 CPU Optimization Options

There are two main CPU optimization switches in GCC 2.x, as listed in the GCC manual:

-mno-486 - optimize for 80386.
-m486 - optimize for 80486. These programs will still run on a 80386.

 Future versions of Gnat built for GCC 3.x or later will probably support:

- **-mpentium** - optimize for Pentium / Intel 586
- **-mcpu=i686** - optimize for Pentium II/ Intel 686
- **-mcpu=k6** - optimize for AMD K6

There are currently no switches newer CPUs such as Pentiums. Under GCC 2.8.1 (and Gnat), the GCC FAQ recommends the following switches for reasonable Pentium performance: "-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2 -fno-strength-reduce".

There are other switches that may or may not be helpful, depending on your program: read the gcc FAQ for full details.

IDE: TIA sets the proper switches for you based on your selections in the project parameters window.

Let's put all these flags together. Suppose you are trying to develop a program for the Intel Pentium CPU with an emphasis on speed. During development, the Gnatmake switches would be "-O1" since this setting suppresses pragma optimize warnings. For the final release, the Gnatmake switches should be "-m486 -O3 -malign-loops=2 -malign-jumps=2 -malign-functions=2 -fno-strength-reduce -gnatp" for maximum performance on a Pentium processor.

7.4 What Difference Does Optimization Make?

In the previous sections, we saw GCC compiler switches and and Ada pragmas that affect the speed and size of your finished application. But how much of a difference does optimization make? And are there any problems caused by optimization?

The optimization switches and pragmas affect different applications differently. Some will give better results to certain kinds of applications, while others may actually have a negative effect. The following table summarizes the results of optimizing on the Hartstone Ada benchmark program. Hartstone is a multithreading mathematics test available freely on the Internet http://ftp.sunet.se/pub4/benchmark/hartstone/.

Table: Hartstone 1.1 Benchmark Summary

Gnat switches	Ada pragmas	CPU Time	File Size	Task Set Util
-gnatE -gnato -g	-	0.13s	294265	0.41%
-gnatE -gnato	-	0.13s	147433	0.41%
-gnatE	-	0.10s	138679	0.32%
no switches	-	0.10s	138679	0.29%
-O	-	0.07s	113076	0.22%
-O2	-	0.07s	113324	0.22%
-O3	-	0.07s	118790	0.20%
-O3 -gnatp	-	0.08s	104290	0.37%
-O3 -gnatp Pent	-	0.08s	105042	0.15%
Max	-	0.05s	105714	0.15%
Max	Optimize(Space)	0.05s	105714	0.15%
Max	Optimize(Time)	0.05s	105714	0.15%
Max	Pack arrays	0.11s	105712	0.15%

Pent - GCC Pentium optimization switches
Max - Pent + -ffast-math + -fomit-frame-pointer
This test was conducted with a Pentium II 350, 64 Megs RAM and ALT Gnat 3.12p-9. As they say, your milage many vary (and probably will).

By optimizing the application, Hartstone can be reduced to half its size and run about 2/3 faster than using no optimization. However, if we pack the arrays in Hartstone, we save two bytes but lose all the improvements in speed. Sometimes smaller programs are not faster.

Let's try optimizing a convoluted program that uses integers, arrays, functions and mathematics and see what effect the optimization techniques have.

```
procedure bench is

   --Simple benchmark program to test optimization

   pragma optimize( time );

   type bench_integer is new long_integer range long_integer'range;
   type small_integer is new long_integer range 0..9;

   function p( param : bench_integer ) return bench_integer is
      divideby : constant bench_integer := 4;
   begin
      return param / divideby;
   end p;
   pragma inline( p );

   j : bench_integer := bench_integer'last;

   -- deliberate error in main program for j * 2

   type atype is array(0..9) of small_integer;
   --pragma pack( atype);

   a : atype;

begin

   for i in 1..100_000_000 loop

      j := abs( p( bench_integer( i ) ) - (j * 2) );
         a( integer( j mod 10 ) ) := small_integer( j mod
         bench_integer( small_integer'last) );

   end loop;

end bench;
```

Notice that j is assigned the largest bench_integer possible. This will force an overflow error the first time around the for loop, when j is multiplied by two. The following chart shows the effect of the different switches and pragmas, and indicates when gnat caught the overflow error. The test was conducted on a Pentium II 350 with 64 Megs of RAM using the gnat 3.11 NYU binaries and was timed with the **time** command.

Gnatmake Switches	Pragmas	CPU Time	Size	Error Caught?
gnatmake -gnato -gnatE	-	-	118162	YES
gnatmake -gnato	-	-	118162	YES
gnatmake -gnatE	-	40.3 s	118162	No
gnatmake	-	40.3 s	117634	No
gnatmake -O	-	10.8 s	117426	No

gnatmake -O2	-	10.8 s	117426	No
gnatmake -O3	-	10.8 s	117426	No
gnatmake -O3 -gnatp	-	9.6 s	117410	No
gnatmake -O3 -gnatp Pent	-	9.6 s	117410	No
gnatmake -O3 -gnatp Pent	Optimize(Space)	9.6 s	117410	No
gnatmake -O3 -gnatp Pent	Optimize(Time)	9.6 s	117410	No
gnatmake -O3 -gnatp Pent	Pack atype	4.4 s	117326	No

Although the proper optimization can make this program run faster, but with overflow checking was turned on with -gnato, the overflow error is caught. The lesson here is that error checking only works when it's turned on.

We can compare the results to the equivalent C program:

```
int p( int param ) {
    return param / 4;
}

int i;
int j = 2147483647;
int a[10];
int main() {
    for (i=1; i<=100000000; i++) {
        j = abs( p(i)-(j*2));
        a[ j%10 ] = j%10;
    }

    return 0;
}
```

GCC Switches	Pragmas	CPU Time	Size	Error Caught?
gcc -Wall	-	12.8 s	24541	No
gcc -O3 Pent	-	8.6 s	24541	No

In this case, notice that C never detected the overflow error. Secondly, notice that the Ada program ran **twice as fast** as the C program.

In theory, an Ada compiler can take advantage of the typing information and the optimization hints provided by the pragmas. The C compiler has less information and this can hinders the optimization process. (I've never investigated whether or not Gnat does this or how much of an effect it has.)

The optimization techniques will affect different programs differently. You need to chose the best approach for your particular project.

7.5 Working with the Assembly Source

Assembly language is the low-level programming language for working with the hardware of a particular computer. Using assembly language, you can access the processor registers, use unusual features of the processor, and dictate exactly which operations the processor performs. Assembly language programs are usually several times smaller and faster than programs written in high-level languages, but they are also several times harder to build, maintain and debug.

The Linux assembler is called **gas** (the GNU assembler). Like GNAT and C++, gas works through gcc. To assemble an assembly language source file, simply run gcc. The compiler will recognize the assembly language file and will assemble it using gas.

If you want to view the assembly source code of your Ada program, use the "-c -S -fverbose-asm" options when compiling. GNAT will create a file with a ".s" suffix containing the assembly source. You can view it, or even edit it and assemble afterwards. Improving the instructions produced by the compiler and then assembling afterwards is known as **hand optimizing**. This technique is typically used for high performance applications such as games, where the programmer needs to get the maximum performance from the hardware.

The following is the stderr.s file for the stderr.adb program described elsewhere in this document.

```
.file"stderr.adb"

.version"01.01"
/ GNU Ada version 2.8.1 (i686-pc-linux-gnu) compiled by GNU C version 2.8.1.
/ options passed:-I../texttools/ -mcpu=i486 -march=i486 -gnatp -gnatf -O3
/ -m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2
/ -fno-strength-reduce -fverbose-asm
/ options enabled:-fdefer-pop -fcse-follow-jumps -fcse-skip-blocks
/ -fexpensive-optimizations -fthread-jumps -fpeephole -fforce-mem
/ -ffunction-cse -finline-functions -finline -fkeep-static-consts
/ -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop
/ -fschedule-insns2 -fcommon -fverbose-asm -fgnu-linker -m80387
/ -mhard-float -mno-soft-float -mieee-fp -mfp-ret-in-387
/ -mschedule-prologue -mcpu=i486 -march=i486 -malign-loops=2
/ -malign-jumps=2 -malign-functions=2
gcc2_compiled.:
.section.rodata
.LC0:
.string"This is an example of writing error messages to stderr"
.align 4
.LC1:
.long 1
.long 54
.LC2:
.string"This message is on standard error"
.align 4
.LC3:
.long 1
.long 33
.LC4:
.string"This message is on standard output"
.align 4
.LC5:
.long 1
.long 34
.LC6:
.string"This is also on standard error"
.align 4
.LC7:
.long 1
.long 30
.LC8:
.string"But this is on standard output"
.text
.align 4
.globl _ada_stderr
.type_ada_stderr,@function
_ada_stderr:
pushl %ebp
movl %esp,%ebp
movl $.LC0,%eax
movl $.LC1,%edx
pushl %edx
pushl %eax
call ada__text_io__put_line__2
```

```
pushl $1
call ada__text_io__new_line__2
movl $.LC2,%eax
movl $.LC3,%edx
pushl %edx
pushl %eax
call ada__text_io__standard_error
pushl %eax
call ada__text_io__put_line
movl $.LC4,%eax
movl $.LC5,%edx
pushl %edx
pushl %eax
call ada__text_io__put_line__2
addl $32,%esp
pushl $1
call ada__text_io__new_line__2
call ada__text_io__standard_error
pushl %eax
call ada__text_io__set_output
movl $.LC6,%eax
movl $.LC7,%edx
pushl %edx
pushl %eax
call ada__text_io__put_line__2
call ada__text_io__standard_output
pushl %eax
call ada__text_io__set_output
movl $.LC8,%eax
movl $.LC7,%edx
pushl %edx
pushl %eax
call ada__text_io__put_line__2
movl %ebp,%esp
popl %ebp
ret
.Lfe1:
.size_ada_stderr,.Lfe1-_ada_stderr
.ident"GCC: (GNU) 2.8.1"
```

See Chapter 19 for a discussion of embedding assembly language into an Ada program.

8 Debugging Your Project

8.1 Limit and the Heap Size

The default storage pool (or the "heap") is kept on the user's stack. Unusually large variables in subprograms (including the main program) can cause out of memory errors. Variables in packages are not affected by the stack size. You can increase the stack space using Linux's limit command (although using dynamic allocation is usually a better solution).

limit stacksize 1024 kbytes # 1 Megabyte user stack

To constrain the size of your stack, as far as Gnat is concerned, use the GNAT_STACK_LIMIT environment variable to indicate the number of kilobytes of stack space. In individual Ada tasks, the stack size can be set by pragma Storage_Size.

Stack size checking is normally disabled by Gnat. Section 4.4 discusses this.

8.2 The Debugging Pragmas

Ada Feature	*Description*	*C Equivalent*
`pragma Assert( condition);`	Assert a condition.	`assert`
`pragma Debug( Procedure );`	Debugging procedure call	-
`pragma Suppress_Debug_Info;`	Disable pragma debug	`#ifdef var...#end if`
`pragma No_Return( subprogram );`	Indicate a subprogram that never completes	-
`pragma Initialize_Scalars;`	Initialize scalars to illegal values	-

GNAT provides two useful pragma for debugging programs. To use these pragmas, you need to use the -gnata option during compiling. Removing -gnata causes GNAT to ignore these pragmas, making it easy to compile a version of your program for public release without having to delete the debugging statements from your source code.**Pragma Assert** lets you test to make sure a certain condition is true. If it isn't, then an exception is raised. Use Assert to check for conditions which you assume are true during program development. This is especially useful when several programmers are working on a project and you don't know if a condition will change in the future.

```
pragma Assert( ScreenHeight = 24 );
```

In this example, if the variable ScreenSize is not 24, an ASSERT_ERROR exception is raised.

Pragma Debug lets you call a procedure for debugging purposes. For example, you can use this to print information about the program while it is running. Because this is a pragma, you can place it almost anywhere, even in the middle of variable declarations.

```
x := 5;
pragma Debug( PrintToLogFile( "X is now" & x'img ) );
```

If PrintToLogFile is a procedure that saves messages to a log file, this example saves the message "X is now 5" to the log file.

Pragma debug can be disabled with **pragma Suppress_Debug_Info**.

Pragma No_Return can be used to indicate subprograms that never complete. This suppresses the related compiler warnings.

Pragma Initialize_Scalars initializes anything not an array, record or tagged record to illegal values wherever possible. This pragma helps expose variables used before they are initialized. Use this at the start of a program or package.

In more recent versions of Gnat, a new pragma **Initialize_Scalars** should be used due to some confusion about what the older Normalize_Scalars affects. Initialize_Scalars is very similar to Normalize_Scalars but is easier to use (no need to have the pragma used for the full partition). This e.g. means you do not have to recompile the run-time. Also, the initial values to initialize the scalars can be decided at bind time. With Normalize_Scalars, there is no control over the values to initialize the scalars.

Suppose you have a integer variable with a range between 1 and 100. Normally, Ada won't assign an initial value (unless you specify one). With Initialize_Scalars, your variable will be initialized to some value out of range, perhaps -1. If you attempt to use this variable, you'll probably raise a CONSTRAINT_ERROR exception.

Initialize_Scalars works better if you use the **-gnatVf** switch.

8.3 Identifying Files

[Rewrite and Expand]From Usenet:

> If you are using 2.2.x, you can use the /proc to find a process which is> using a directory or file. If is the case, try this: ls -lad `find> /proc` | grep home> The number after the /proc should be the process ID.>> Any way, it seems a little bit delicate umount your /home after the> boot... Couldn't you use a rescue disk and format your /home without> mount it in a boot?>> Are you sure your /home isn't only a part of your / (root directory)?> When you type "df", does it report a different device for the /home?> Sorry, if I'm asking a very basic question.>try fuser or lsof.

8.4 Compiler Info with -gnatG

The -gnatG compiler switch shows Gnat's interpretation of your source code after its initial analysis. If you specify -gnatD, Gnat will write this information to a file ending in .dg (for "debug").

The following is a listing of the "pointers" program used later in this book:

```
with Ada.Text_IO, System.Address_To_Access_Conversions;
use Ada.Text_IO;

procedure pointers is

   package IntPtrs is new System.Address_To_Access_Conversions( integer );
   -- Instantiate a package to convert access types to/from addresses.
   -- This creates an integer access type called Object_Pointer.

   five       : aliased integer := 5;
   -- Five is aliased because we will be using access types on it

   int_pointer : IntPtrs.Object_Pointer;
   -- This is an Ada access all type

   int_address : System.Address;
```

```
    -- This is an address in memory, a C pointer

begin

    int_pointer := five'unchecked_access;
    -- Unchecked_access needed because five is local to main program.
    -- If it was global, we could use 'access.

    int_address := five'address;
    -- Addresses can be found with the 'address attribute.
    -- This is the equivalent of a C pointer.

    int_pointer := IntPtrs.To_Pointer( int_address );
    int_address := IntPtrs.To_Address( int_pointer );
    -- Convert between Ada and C pointer types.

end pointers;
```

The -gnatG shows the compiler's analysis of your program. In this case, it displays the results of the instantiation of the generic package:

```
with system;
with ada;
with ada.text_io;
with system.address_to_access_conversions;
use ada.text_io;
with system;
with system;
with unchecked_conversion;

procedure pointers is

      package intptrs is
            subtype object is integer;
            package address_to_access_conversions renames intptrs;
            null;
            type object_pointer is access all object;
            for object_pointer'size use 32;
            function to_pointer (value : address) return object_pointer;
            function to_address (value : object_pointer) return address;
            pragma convention (intrinsic, to_pointer);
            pragma convention (intrinsic, to_address);
            freeze object_pointer []
            freeze to_pointer []
            freeze to_address []
      end intptrs;

      package body intptrs is

            function to_address (value : object_pointer) return address is
            begin
                  if value = null then
                        return null_address;
                  else
                        return value.all'address;
                  end if;
            end to_address;
```

```
        function to_pointer (value : address) return object_pointer is

            package a_to_pGP3183 is
                subtype source is address;
                subtype target is object_pointer;
                function a_to_pR (s : source) return target;
            end a_to_pGP3183;
            function a_to_p is new unchecked_conversion (address,
               object_pointer);
        begin
            return target!(source(value));
        end to_pointer;
    end intptrs;

    package intptrs is new system.address_to_access_conversions (integer);
    five : aliased integer := 5;
    int_pointer : intptrs.object_pointer := null;
    int_address : system.address;
    freeze intptrs []
begin
    int_pointer := five'unchecked_access;
    int_address := five'address;
    int_pointer := intptrs.to_pointer (int_address);
    int_address := intptrs.to_address (int_pointer);
    return;
end pointers;
```

8.5 Floating Point Numbers

The GCC FAQ reports that floating point rounding problems can occur with -O2 and -O3 unless you use -ffloat-store (keeps floating-point numbers out of CPU registers). Using this switch will slow your program.

8.6 Gdb: the GNU debugger

gnat 3.11 and later with a version of **Gdb**, the GNU command line debugger, that fully supports Ada data structures.You shouldn't have to use Gdb very often as most problems are solvable with a few well-placed put_line's. However, if the program produces are core file or is behaving unpredictably because of an obscure coding mistake or a compiler bug, Gdb is the best way to find out what is going wrong and where.

In order to use Gdb, you must compile your program with debugging support enabled (with the -g option in gnat, or -ggdb in C).

To start Gdb on a program called dbase, use

gdb dbase

(or gnatgdb with the ALT version) and Gdb responds with its command line prompt, "(gdb)". Type **run** to start the program normally to the point of the crash. You can print out the value of variables using the **print** command:

(gdb) print ch

This will print the character in the variable named ch at the time when the program was stopped.

To look at a core file produced by a segmentation fault, use

gdb dbase core

You can examine the variables at the time the program crashed.

Gdb contains many other commands. You can get online help at any time with the help command.

GNAT has a hidden **-gnatdg** flag. If you compile your program using this flag, you'll get extra information for Gdb, such as making all temporary variables used by gnat visible to the debugger.

8.7 Code Restrictions

There are several pragmas for disabling certain language features. These **restriction** pragmas can be used to enforce a certain policy and warn a programmer when the policy is violated. For example, if you are writing a real-time program, you may want to disable Ada features that do not have a known response time so that your program will not have random delays.

The restriction pragmas include:

- **Ada_83** - do not allow Ada 95 language features
- **Ada_95** - (default) allow Ada 95 language features
- **Controlled** - turn off garbage collection for a type. This has no effect with Gnat since it does not implement garbage collection.
- **Ravanscar** - enforce Ravanscar run-time restrictions
- **Restricted_Run_Time** - similar to Ravanscar
- **Restrictions** - disable particular language features

no_run_time also enforces restrictions because the Ada run-time library is not available.

More information on the usage of these pragmas is available in the Gnat documentation.

9 Team Development

9.1 Change Logs

The Change Log

Creating a change log is the easiest method to document changes you've made in a project. A change log is a text file (usually called "CHANGES") containing an explaination of all recent changes in a project. For example, if you are working on an open source project and you add support for encrypted passwords, you might document this by writing

```
Nov 1 - added password support to file passwords.adb
```

If anyone was going to add password support to your project, they can quickly see that you've already done it. Or if somebody was adding additional features to the passwords.c file, they will know that they may have to change their work since you've already changed the same file.

The change log is popular on open source projects involving few programmers because there's little chance of two programmers modifying the same source file simultaneously.

The Formal Change Log

Most businesses or professional institutions have a much more formal structure for their change logs. In an environment involving important data, a program problem means someone else will have to retrace your activities in order to find and correct a problem.

Formal logs are kept in a binder since there's always the possibility that a major failure will make it impossible to sign onto the computer. Each change is documented with

- the date
- person who made the software change
- what programs were affected
- a description of the change

Some companies have internal audits done to ensure that changes were properly made. The auditors will pick random pages from the change log and ask the programmer to verify the changes. In these cases, a formal change log may also include information such as the size, ownership and permissions of files affected. This serves both to quickly check a file as well as to force a programmer to verify the security of the files he or she installs. The most common security loopholes in UNIX are caused by people not checking the ownership and permissions of installed files.

9.2 RCS: Revision Control System

RCS (Revision Control System) is a tool that shares a document or program source code between multiple people. It also automatically numbers the file with a version number (eg. 1.1, 1.2) with each revision, and maintains a change log. CVS, an extension to RCS, is described below.Once you initialize RCS for a project, people in your project "check out" (with the **co** command) a copy of a file, and when they're done making changes they "check in" the file (with the **ci**command).

RCS is also a good tool for maintaining documentation.

Read the **rcsintro** man page for more information on getting started.

The following is a transcript of a session using RCS. Assume that you have a source file called "f.c". To add the source file to RCS, you'd use ci. You are prompted for a general description of the

file and RCS assigns version number 1.1 to the file. The file is deleted from your directory and moved into RCS's care.

```
armitage:/home/ken/ada/rcs# ci f.c
RCS/f.c,v <--f.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>>
Curreny Definitions
>> .
initial revision: 1.1
done
armitage:/home/ken/ada/rcs# ls
RCS
```

To check out the file, read-only, use co. The file reappears.

```
armitage:/home/ken/ada/rcs# co f.c
RCS/f.c,v -->f.c
revision 1.1
done
armitage:/home/ken/ada/rcs# ls
RCS f.c
```

To check out a file to change it, use **co -l** (lock out others):

```
armitage:/home/ken/ada/rcs# co -l f.c
RCS/f.c,v --> f.c
revision 1.1 (locked)
done
```

Suppose you add the line "--test line" to the file. Rcsdiff will report any changes you've made to the file since checking it out:

```
armitage:/home/ken/ada/rcs# rcsdiff f.c
===================================================================
RCS file: RCS/f.c,v
retrieving revision 1.1
diff -r1.1 f.c
0a1
> --test line
```

Finally, you can check the file back in. RCS increments the version number and prompts you for a message for the change log.

```
armitage:/home/ken/ada/rcs# ci f.c
RCS/f.c,v <-- f.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Added comments
>> .
done
```

9.3 CVS: Concurrent Versions System

CVS (Concurrent Versions System) is a front end to RCS designed to work with groups of files in multiple directories. CVS can work with individual files, whole directories or you can organize large projects into groups of files (called a *modules*) that you want to work with. Like RCS, it timestamps files, maintains version numbers, and identifies possible problems when two programmers update the same section of a program simultaneously.

CVS is very popular for open source development. CVS can be configured to allow programmers all over the world to work on your project without having to be logged into your computer.

In order to use CVS, the project leader needs to create a directory for CVS to work in (called the *repository*) and a subdirectory called CVSROOT. Then you define an environment variable called CVSROOT so CVS knows where to find the CVS directory. For example, to make "/home/our-project-cvs" the repository for your team, set up the CVSROOT under bash as

export CVSROOT=/home/our-project-cvs

The repository holds copies of all the files, change logs, and other shared resources for your project.

To add a new project to the CVS repository, use the **import** command. Import will take the files in the current directory and put them in the repository under the name you specify. Import also requires a short string to identify who is adding the project, and a string to describe the state of the project. This strings are comments and can be anything: your login and "init-rel" for initial release may be good choices.

By convention, CVS begins numbering your project with "1.1"

```
[root@redbase cvs]# cvs import project kburtch init-rel
(CVS starts your default editor, typically vi)
CVS: ----------------------------------------------------------------------
CVS: Enter Log. Lines beginning with `CVS: ' are removed automatically
CVS:
CVS: ----------------------------------------------------------------------
(make notes and exit vi)
N project/currency.adb
No conflicts created by this import
```

The "N project/currency.adb" line indicates that CVS created a new project called "project" and added the Ada file currency.adb to it. currency.adb is now stored in the CVS repository, ready to be shared amongst the team members.

To work with a project, you use **co** (or **checkout**). This CVS command will save a copy of the project in your directory. It will also create a CVS directory to save private data files used by CVS. To use co, move to your home directory and type:

```
[root@redbase cvs]# cvs checkout currency.adb
cvs checkout: Updating .
U project/currency.adb
```

The subdirectory project will contain your own, personal copies of project files to work on. CVS maintains the original copy of currency.adb. Another programmer can also checkout currency.adb while you are working on your copy.

If you do a checkout right after an import, you may have to remove the original files: CVS will not overwrite any existing files.

To add a file to CVS, use the **add** command. To add a file called currency.adb, use

[root@redbase cvs]# cvs add currency.adb


Single files, directories or even CVS modules can also be added to your project using "add".

As you work on your source code, you can check your work against the project using the **update** command.

```
[root@redbase cvs]# cvs update
cvs update: Updating .
```

When updating, CVS checks the files in your copy of the project against its copies. If another team member made changes to one of the project Ada files, CVS will copy the new file to your directory.

If another team member made changes to one of the Ada files you've been working on, CVS will attempt to update your copy without destroying your work.

Sometimes the changes involve the same part of the Ada file and CVS won't be able to combine the changes automatically. CVS calls this a **conflict**. For example, suppose your Ada file contained a function

```
function ConvertCurrency( amount : integer ) return float;
```

If you changed this function to use a float amount, and another team member has changed amount to a string, CVS will report a conflict. You will have to talk to the team member who made the change and make an agreement what amount should be.

If there are no other problems after an update, you can continue working on your source code.

To delete a file, remove it using "rm" and then "update". CVS will see your file is no longer in the project.

 The CVS command **release** will permanently remove a file from a project (including the copy in CVSROOT), but it also prevents you from recovering the file from the CVSROOT directory in an emergency. Unless storage space is limited, consider using the rm/update method of removing files.

While working on your source code, your changes are not distributed to the rest of your team until you are ready. When your source code is tested and ready to be made available, use **ci** (or **commit**). Before commiting your changes, delete non-essential files (such as .ali, .o or executable files) to save space in the repository.

The **log** command gives information about a group of files:

```
[root@redbase cvs]# cvs log -l project
cvs log: Logging project
RCS file: /usr/cvs/project/currency.adb,v
Working file: project/currency.adb
head: 1.1
branch: 1.1.1
locks: strict
access list:
symbolic names:
p1: 1.1.1.1
keyword substitution: kv
total revisions: 2; selected revisions: 2
description:
----------------------------
revision 1.1
date: 1999/01/13 17:27:33; author: kburtch; state: Exp;
branches: 1.1.1;
Initial revision
----------------------------
```

```
revision 1.1.1.1
date: 1999/01/13 17:27:33; author: kburtch; state: Exp; lines: +0 -0
Project started
=============================================================================
```

Status gives you an overview of a group of files:

```
[root@redbase cvs]# cvs status project
cvs status: Examining project
===================================================================
File: currency.adb Status: Up-to-date
Working revision: 1.1.1.1 Wed Jan 13 17:27:33 1999
Repository revision: 1.1.1.1 /usr/cvs/project/currency.adb,v
Sticky Tag: (none)
Sticky Date: (none)
Sticky Options: (none)
```

9.4 Creating Transcripts with Script

So you did something wrong. How to you show what you did to your fellow programmers?
The **script** command creates a file called "typescript" in the current directory. The typescript file is a text file that records a list of everything that appears on the screen. You can stop the recording process with the **exit** command.

9.5 Timing Execution with Time

The shell command **time** will tell you how long a program took to run, and reports general statistics such as how many page faults occurred.

```
armitage:/home/ken/ada/sm# time myprog
3.09user 0.95system 0:05.84elapsed 69%CPU(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs(4786major+4235minor)pagefaults 0swaps
```

10 An Introduction to Ada

Ada is a full-featured language with many capabilities, rules, and nuances. Although the fundamentals are easy to learn (Ada somewhat resembles BASIC), it is several times larger that C, and to truly master the language requires considerable practice. To make understanding easier, the discussion is broken up into two chapters. This chapter outlines the basics of the language, and the next chapter discusses features for team development, large projects, and other specialized tasks.

This in no way covers everything there is to know about Ada. I've chosen to cover those features that have been the most use to me over the years in my projects. For example, array slicing alone could take up several pages of discussion, but I've never had a need for it in recent years. Of course, you may be involved in a project in which array slicing is crutial. In these cases, I recommend you get a good Ada 95 reference such as Barnes' Programming in Ada 95.

Likewise this is not a complete introduction to computer programming. Some background knowledge is assumed.

Ada also has many specialized features for specific tasks such as scientific computing and real-time systems. Where I deliberately skip a subject, I usually make a note that I have done so. I've also hilighted useful information for C programmers who are learning Ada.

You may also want to consider the Lovelace Tutorial which is aimed more at beginners.

Now, on to main programs.

10.1 Your Main Program

A **main program** in Ada is a procedure with no parameters that starts your program running. This is the set of instructions that the computer begins to follow when your program is first executed.

The following program will print a message on the screen when you run it.

```
with Ada.Text_IO;
procedure firstProgram is
-- my first Ada program
begin
   Ada.Text_IO.Put_Line( "This is my first Ada program." );
end firstProgram;
```

C: Ada is not case sensitive. "WITH" or "With" is the same as "with".

An Ada program consists of sets of words and punctuation symbols. The words fall into two categories. First, the words in bold in bold are called **keywords**. These are words that have special meaning to Ada. Second, the words that aren't in bold are identifiers. These are the names of variables, procedures, packages and other items with names or titles in the language.

IDE: Ada IDE's will hilight keywords in bold for you. Some editors such as emacs, elvis and nedit will also hilight keywords. This is a good way to check for spelling mistakes.

In this program, begin is a keyword because Ada uses the word "begin" to denote where the program is to begin executing instructions. On the other hand, "firstProgram" is an identifier because it is the name of our program.

All keywords in Ada are also **reserved words**: this means that the keywords cannot be used as identifiers.

The main program can have any name, as long as the name matches the filename. In gnat, the source code for a main program ends in .adb (Ada body). This program should be saved as firstprogram.adb.

C: The main program doesn't have to be "main" unless you save the program as "main.adb".

 If you call a program "test.adb", remember that test is a built-in shell command. To run a program named test, you'll have to type "./test" instead of "test" to avoid running the shell command by mistake.

Comments are denoted by two minus signs (--). This is a note to the reader; Ada will ignore it. Everything you type to the right of the symbol is treated as a remark to the reader.

C: Ada has no equivalent to the block comment /* and */.

10.2 Text_IO

Ada	*Description*	*C Equivalent*
`put( s );`	Display a string	`printf( "%s", s );`
`put( n'img );`	Display a number	`printf( "%d", n );`
`put_line( s );`	Display a line of text and start a new line	`printf( "%s\n", s );`
`new_line;`	Start a new line	`printf( "\n" );`
`get( c );`	Read a character from the keyboard	`c = getc();`
`get_line( s, len );`	Read a line of text from the keyboard	`gets(&s);`
etc.		

Like many modern computer languages, Ada doesn't have any built-in methods of reading the keyboard or writing messages on the screen. It doesn't assume you're writing a program for a PC (you could be doing embedded programming, for example)--but in general, you need to interpret what people type and display the results to the screen. You have to add this functionality specifically.

The standard input/output package for Ada is **Text_IO**. This package prints characters and strings to the screen and reads characters and strings from the keyboard. It can also read and write simple sequential text files. (Packages will be discussed in detail starting at 11.1 in the next chapter.)

Text_IO is only useful for simple programs. It doesn't have the ability to draw buttons, windows or menus. For X Windows programming, you'll require other packages/libraries to perform input and output.

C: In C, `printf` and `company` can use an arbitrary number of parameters, where the parameters can be of different types. Text_IO's puts have one parameter, and the parameter must be a string or a character. Upcoming sections demonstrate how to print other types.

The most commonly used operations are:

- **Put** - write to the screen, but don't start a new line
- **Put_Line** - write to the screen and start a new line
- **New_Line** - start a new line
- **Get** - read a character from the keyboard
- **Get_Line** - read a string from the keyboard

The following program is an example of Text_IO.

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure basicio is
-- this program demonstrates basic input/output

   c : character; -- this is a letter

begin

   Put_Line( "This program displays information on the screen" );
   Put_Line( "and reads information from the keyboard");
   New_Line;

   Put_Line( "Put_Line displays a line of text and advances to" );
   Put_Line( "the next line." );

   Put( "Put " );
   Put_Line( "displays text, but it doesn't start a new line" );

   Put_Line( "New_Line displays a blank line");
   New_Line;

   Put_Line( "Get waits for a character to be typed.");

   Put_Line( "Type a key and the Enter key to continue." );
   Get( c );
   Put_Line( "The character you typed was ''' & c & ''' );

end basicio;
```

```
This program displays information on the screen
and reads information from the keyboard

Put_Line displays a line of text and advances to
the next line.
Put displays text, but it doesn't start a new line
New_Line displays a blank line

Get waits for a character to be typed.
Type a key and the Enter key to continue.
c
The character you typed was 'c'
```

Besides letters and numbers, there are special characters called *control characters* which, instead of displaying a character, change the Linux display. To print controls characters, you need to use one of Ada's built-in character sets. For example, ASCII is a predefined list of all the ASCII characters. To send an explicit form feed character, use

```
Put( ASCII.FF );
```

Some common control characters are:

- **ASCII.NUL** - C end of string character
- **ASCII.CR** - carriage return - move to beginning of line
- **ASCII.HT** - horizontal tab
- **ASCII.LF** - line feed - start a new line

- **ASCII.FF** - form feed - start a new page on a printer

C: Put doesn't recognize C string escape codes like "\n" or "\r".

Besides ASCII, Ada has a number of other character sets defined in the Ada.Characters packages.

 The ASCII set is officially made obsolete in Ada 95 by Ada.Characters.Latin_1, but it's still often used because it's easier to type.

10.3 Fundamental Data Types

Ada Type	*Description*	*C Equivalent*
`Character`	A single character	`char`
`Integer`	An integer (32-bit) number	`int`
`Natural`	Zero or positive integer	-
`Positive`	Positive integer	-
`Long_Integer`	A big integer (same as long in Gcc)	`long (same as int in Gcc)`
`Long_Long_Integer`	A really big (64-bit) integer	`long long`
`Short_Integer`	A small (16-bit) integer	`short`
`Short_Short_Integer`	A really small (8-bit) integer	`char`
`Float`	A real number	`float`
`Long_Float`	A big real number	`double`
`Long_Long_Float`	A really big real number	`long double`
`Short_Float`	A smaller real number	?
`Fixed`	A fixed-point real number	-
`String`	An Ada fixed-length string	`char` array

C: There are no built-in equivalents of unsigned types. Natural and Positive are integer values that aren't allowed to be negative, effectively requiring the sign bit to be zero.

Characters cannot be used for small integer values--characters variables can only represent character values.

Generally speaking, programs take data, process it in different ways, and create new information. Data is categorized into different data types.

Data that is typed into a program is known as **literals**. Ada has several kinds of literals:

- 'c' is a character. Character literals are enclosed in single quotes.
- -5 is an integer
- 45.5 is a float or a fixed with one decimal place
- "This is a string" is a fixed string. Strings literals are enclosed in double quotes.

C: Ada doesn't have long numerical literals, like "45L". Numeric literals are a special type called universal integer and adapt to fit the requirements of an expression.

C: Ada strings do not end with an ASCII 0 character: they end with the upper bound of the array that encloses them. To change an Ada string into a C string, concatenate a null character like this:

```
"This is my string" & ASCII.NUL;
```

There are three kinds of real numbers. A **fixed**, or fixed point, number is a number that has a fixed number of decimal points. For example, U.S. dollars are often represented using fixed numbers because there are two decimal places. A **float**, or floating-point, number is a number that doesn't have a fixed number of decimal places. **Decimal** numbers are a variation of fixed numbers commonly used for currency.

Some Ada programmers recommend that floats are used whenever possible because float calculations are usually faster than fixed calculations. This is because most computers today have floating point support in their hardware.

Fixed point numbers are declared with a **delta** part (and optional range limit) which the compiler uses to determine the minimum number of bits needed to store the number and the fractional part. If you need a specific reason (for example, if you intend to use a for loop to step through the range of the type), chose it using a **for** clause.

```
type aFixedNumber is delta 0.1 range 0.0 .. 1.0;
-- represented as multiples of 0.0625 (1/128)
type aFixedNumber2 is delta 0.1 range 0.0 .. 1.0;
for aFixedNumber2'Small use 0.1;
-- represented as multiples of 0.1
```

On Linux, decimal numbers use floating-point arithmetic.

Floating point numbers are very important for business and scientific applications. When floating point numbers are converted to integers, do the numbers round to the nearest integer or is the decimal part simply discarded? In C, this is system dependent: System V-based UNIX's usually round to the nearest integer, while some other systems discard the decimal part. (Others, like HP-UX, the number rounds towards the nearest even integer providing the floating point number is exactly half way between two integers.)

On Linux, C truncates the decimal part.

In Ada, the way numbers round are strictly defined by the language: you can be sure that, no matter what operating system you are using, floating point numbers converted to integers will always round to the nearest integer. If the floating point number is half way between two integers, it will round "up".

The following program demonstrates floating point rounding:

```
with ada.text_io, ada.float_text_io;
use ada.text_io, ada.float_text_io;

procedure rounding is
   -- rounding example

   procedure ShowRounding( f : float ) is
      -- show the floating point value, and show the value
      -- after it's converted to an integer
      int_value : integer;
   begin
      Put( "  Float number " );
```

```
        Put( f, fore => 5, aft => 3 );
        int_value := integer( f );
        Put_Line( " rounds to " & int_value'img );
    end ShowRounding;

begin
    Put_Line( "This is a demonstration of how Ada 95 rounds" );
    New_Line;

    ShowRounding( 253.0 );
    ShowRounding( 253.2 );
    ShowRounding( 253.5 );
    ShowRounding( 253.8 );
    ShowRounding( -253.8 );

end rounding;
```

```
This is a demonstration of how Ada 95 rounds

    Float number     2.530E+02 rounds to  253
    Float number     2.532E+02 rounds to  253
    Float number     2.535E+02 rounds to  254
    Float number     2.538E+02 rounds to  254
    Float number    -2.538E+02 rounds to -254
```

You can compare the results with the following C program:

```
#include <stdio.h>

static void show_rounding( float f ) {
      int i;

      i = f;
      printf( "  Float number %g", f );
      printf( " rounds to %d\n", i );

} /* show rounding */

int main () {

    show_rounding( 253.0 );
    show_rounding( 253.2 );
    show_rounding( 253.5 );
    show_rounding( 253.8 );
    return 0;

}
```

```
    Float number 253 rounds to 253
    Float number 253.2 rounds to 253
    Float number 253.5 rounds to 253
    Float number 253.8 rounds to 253
```

Rounding to integers is a common way in C business applications to round money to the nearest dollar or cent. This is accomplished by multiplying the floating point value by 100.0, adding .5, and then taking the integer value and converting it once more into a floating point value. In Ada, there's a built-in type attribute to round floating point numbers: this makes conversion to an integer unnecessary.

C: The fundamental integer types don't "wrap around" the way C data types do. Values that grow too large produce overflow errors. However, gnat turns off integer overflow exceptions by default to improve performance. Ada provides properly behaved C types and conversion functions in the Interfaces.C package. Interfaces.C includes the following types:

```
type int is new Integer;
type short is new Short_Integer;
type long is range -(2 ** lbits1) .. +(2 ** lbits1) - 1;
type signed_char is range SCHAR_MIN .. SCHAR_MAX;
  for signed_char'Size use CHAR_BIT;
type unsigned is mod 2 ** int'Size;
type unsigned_short is mod 2 ** short'Size;
type unsigned_long is mod 2 ** long'Size;
type unsigned_char is mod (UCHAR_MAX + 1);
for unsigned_char'Size use CHAR_BIT;
```

GNAT has a second package, Interfaces.C.Extensions, that includes additional types, such as unsigned_long_long.

As it's name suggests, the Text_IO package only performs I/O with text, not numbers or other types of information. If you want to print, say, and integer value using Text_IO, you must first convert the integer to a string using the **'img** attribute (or **'image**). (Attributes are discussed in the next section.)

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure basicio2 is
-- this program demonstrates more advanced input/output

   i : integer := 5; -- this variable contains an integer number
                                  -- i initially has the value of 5
   s : string(1..20); -- this variable contains a 20 character string
   len : natural;

begin

   Put_Line( "This program displays information on the screen" );
   Put_Line( "and reads information from the keyboard");
   New_Line;

   Put_Line( "'img returns the string representation of a variable's" );
   Put_Line( "value. The value i is" & i'img);
   New_Line;

   s := "....................."; -- set s to 20 periods

   Put_Line( "The variable s is " & s);
   Put_Line( "Get_Line reads a string from the keyboard" );
   Put_Line( "Type in a message up to 20 characters and press Enter:" );
   Get_Line( s, len );

   Put_Line( "After Get_Line copies your message to s, s is now '" & s & "'" );
   Put_Line( "The message is" & len'img & " characters long." );
```

```
    Put_Line( "The characters after your message remain unchanged." );

end basicio2;
```

```
This program displays information on the screen
and reads information from the keyboard

'img returns the string representation of a variable's
value. The value i is 5

The variable s is .....

Get_Line reads a string from the keyboard
Type in a message up to 20 characters and press Enter:
jingle bells

After Get_Line copies your message to s, s is now 'jingle bells.....'
The message is 12 characters long.

The characters after your message remain unchanged.
```

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure basicio3 is
    -- this program demonstrates more even advanced input/output

    i : integer := 5; -- this variable contains an integer number
                                        -- i initially has the value of 5
    s : string(1..5); -- this variable contains a 5 character string

    len : natural;    -- length of string

begin

    Put_Line( "This program displays information on the screen" );
    Put_Line( "and reads information from the keyboard");
    New_Line;

    Put_Line( "The value i is" & i'img);
    New_Line;

    Put_Line( "integer'value changes a string into an integer value" );
    Put_Line( "Type in a 4 character integer characters with a leading" );
    Put_Line( "space or negative sign and press Enter:");
    Get_Line( s, len );

    i := integer'value( s );
    Put_Line( "The value of i is " & i'img);
    New_Line;

end basicio3;
```

```
This program displays information on the screen
and reads information from the keyboard

The value i is 5

integer'value changes a string into an integer value

Type in a 4 character integer characters with a leading
space or negative sign and press Enter:
2345

The value of i is 2345
```

Besides Text_IO, Ada provides additional "Text_IO" packages for the basic Ada data types. Using these packages, you don't need to use **'img** to convert the variable to a string. For example, the package Ada.Integer_Text_IO can put and get integers, and Ada.Float_Text_IO can put and get floating point numbers. You can use these packages simultaneously with Text_IO.

These additional packages do not have Put_Line or Get_Line because these are specifically for strings. The Put command has two additional capabilities: to space information to fit into specified field widths, and to display numbers in formats other than base 10.

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure basicio4 is
-- this program demonstrates integer input/output
   i : integer := 5; -- this variable contains an integer number
   -- i initially has the value of 5
begin
   Put_Line( "This program displays information on the screen" );
   Put_Line( "and reads information from the keyboard" );
   New_Line;
   Put( "The value i is" ); Put( i ); New_Line;
   New_Line;
   Put_Line( "Type in an integer number." );
   Get( i );
   New_Line;
   Put( "The value i is" ); Put( i ); New_Line;
   New_Line;
   Put_Line( "'width =>' specifies the amount of room to display the number in." );
   Put_Line( "This can be used to display columns of numbers." );
   Put( "Using a width of 5, the value i is '" );
   Put( i, width => 5 );
   Put_Line( "'" );
   New_Line;
   Put_Line( "'base =>' specifies a number system besides the normal base 10" );
   Put( "Using binary notation, the value i is " ); Put( i, base => 2 ); New_Line;
   New_Line;
   Put_Line( "Set the variable Default_Width or Default_Base to avoid using" );
   Put_Line( "'width =>' and 'base =>'." );
   Put( "The Default_Width was " ); Put( Default_Width ); New_Line;
```

```
    Default_Width := 20;
    Put( "The Default_Width is now " ); Put( Default_Width ); New_Line;
    Put( "The value i is '" ); Put( i ); Put_Line( "'" );
    New_Line;
end basicio4;
```

```
This program displays information on the screen
and reads information from the keyboard
The value i is 5
Type in an integer number.
432
The value i is         432
'width =>' specifies the amount of room to display the number in.
This can be used to display columns of numbers.
Using a width of 5, the value i is '   32'
'base =>' specifies a number system besides the normal base 10
Using binary notation, the value i is 2#110110000#
Set the variable Default_Width or Default_Base to avoid using
'width =>' and 'base =>'.
The Default_Width was 11
The Default_Width is now 20
The value i is '                   432'
```

Table: Predefined Text_IO packages for Numeric Types

Type	Text_IO Package
Short_Short_Integer	Ada.Short_Short_Integer_Text_IO
Short_Short Integer (wide)	Ada.Short_Short_Integer_Wide_Text_IO
Short_Float	Ada.Short_Float_Text_IO
Short_Float (wide text)	Ada.Short_Float_Wide_Text_IO
Short_Integer	Ada.Short_Integer_Text_IO
Short_Integer (wide text)	Ada.Short_Integer_Wide_Text_IO
Integer	Ada.Integer_Text_IO
Integer (wide text)	Ada.Integer_Wide_Text_IO
Float	Ada.Float_Text_IO
Float (wide text)	Ada.Float_Wide_Text_IO
Long_Float	Ada.Long_Float_Text_IO
Long_Float (wide text)	Ada.Long_Float_Wide_Text_IO
Long_Integer	Ada.Long_Integer_Text_IO
Long_Integer (wide text)	Ada.Long_Integer_Wide_Text_IO
Long_Long_Float	Ada.Long_Long_Float_Text_IO

Long_Long_Float (wide)	Ada.Long_Long_Float_Wide_Text_IO
Long_Long_Integer	Ada.Long_Long_Integer_Text_IO
Long_Long_Integer (wide)	Ada.Long_Long_Integer_Wide_Text_IO
Unbounded (String)	Ada.Unbounded_IO
Wide_Unbounded (String)	Ada.Wide_Unbounded_IO
Calender.Time	Gnat.Time_IO

10.4 Type Attributes

Ada has a selection of *attributes*, or built-in functions, that can be applied to types and variables. Attributes are attached to the end of a type or variable name using a single quote.

The most useful attribute, the **'img** attribute, returns the ASCII image of what it's attached to, which is handy for printing values on the screen using only Ada's Text_IO package. (5)'img, for example, is the string "5". false'image, the image of the boolean value false, is the string "FALSE" in capital letters. One quirk of 'img is that if the value is a positive number, 'img adds a leading blank.

'img is a GNAT shorthand for the Ada attribute **'image**. 'image requires you to specify the type of the parameter. integer'image(5) is the string "5". This attribute is useful on complicated expressions where 'img won't work because of the lack of parentheses.

Here's a list of Ada 95 attributes: [To Be Completed]

Access - access value for an identifier

Address - address value for an identifier
Adjacent - the floating point value adjacent to the given value
Aft - for fixed types, number of decimal digits after decimal to accommodate a subtype
Alignment - storage value of an identifier
Base - unconstrained subtype of a type
Bit_Order - whether or not bits are high order first
Body_Version [NQS]
Callable - true if task can be called
Ceiling - round up a floating point value
Class - classwide type of an identifier
Component_Size - size of array components in bits
Compose
Constrained
Copy_Sign
Count
Definite
Delta
Denorm
Digits
Exponent
External_Tag
First - first index in an array
First_Bit
Floor - round down a floating-point value
Fore
Fraction
Identity
Image

Input - convert value to a string
Last - last index in an array
Last_Bit
Leading_Part
Length - number of elements in an array
Machine
Machine_Emax - maximum real type exponent on your hardware
Machine_Emin - minimum real type exponent on your hardware
Machine_Mantissa - size of mantissa on your hardware in bits
Machine_Overflows - true of your machine overflows real types [NQS]
Machine_Radix
Machine_Rounds
Max - maximum value
Max_Size_In_Storage_Elements
Min - minimum value
Model
Modulus
Output
Partition_ID - for distributed processing
Pos - position in a discrete type (such as an enumerated) — opposite of val
Position
Pred - previous value in a discrete type
Range - range of values for a type
Read
Remainder
Round
Rounding
Safe_First
Safe_Last
Scale
Scaling
Signed_Zeros
Size - size of storage in bytes
Small
Stoarge_Pool - used to set the storage pool for a pointer
Storage_Size
Succ - next value in a discrete type
Tag - tag for a tagged record
Terminated - true if task has terminated
Truncation
Unbiased_Rounding
Unchecked_Access - return access type, but ignore scope checks
Val - value of a discrete type at a certain position — opposite of pos
Valid - determine if the expression evaluates to a legal result
Value - convert string to a value — opposite of image
Version
Wide_Image - same as image, but for a 16-bit string
Wide_Value - same as value, but for a 16-bit string
Wide_Width
Write
Here's a list of additional gnat-specific attributes:
[should fold these in above]

Abort_Signal - task abort exception
Address_Size - number of bits in an address
Bit - offset to first bit in object
Default_Bit_Order - whether or not CPU uses high order first
Elab_Body - the procedure that elaborates a package body
Elab_Spec - the procedure that elaborates a package spec
Enum_Rep - the numerical value of an enumerated identifier
Fixed_Value - unchecked conversion of integer to a fixed type
Img - shorthand for ‘image
Integer_Value - the reverse of Fixed_Value
Machine_Bits - for compatibility with other Ada compilers
Max_Interrupt_Priority - the maximum interrupt priority
Max_Priority - the maximum task priority
Maximum_Alignment - determine the bit alignment of an external object
Mechanism_Code - how a parameter is passed to a subprogram
Null_Parameter - for passing null pointer for a composite object
Object_Size - for fixed and discrete types, default allocation size
Passed_By_Reference - true if type is normally passed by reference
Range_Length - number of values in a discrete type
Storage_Unit - same as System.Storage_Unit
Tick - same as System.Tick
Type_Class - return type basic class of an identifier (such an enumerated or array)
Universal_Literal_String - return a string literal for a number
Unrestricted_Access - like access, but has no accessibility or aliased view checks
Value_Size - number of bits to represent a value of a given subtype
Word_Size - same as System.Word_Size
The following program demonstrates some of the basic Ada attributes.

```
with text_io;
procedure attrib is
   type enum is ( dog, mica, megabyte );
begin
   Text_IO.Put_Line( "Some Basic Ada Attributes:" );
   Text_IO.New_Line;
   Text_IO.Put_Line( "Boolean bits is  " & boolean'size'img );
   Text_IO.Put_Line( "Short short integer bits is" &
      short_short_integer'size'img );
   Text_IO.Put_Line( "Short integer bits is " & short_integer'size'img );
   Text_IO.Put_Line( "Integer bits is " & integer'size'img );
   Text_IO.Put_Line( "Long integer bits is " & long_integer'size'img );
   Text_IO.Put_Line( "Long long integer bits is " &
      long_long_integer'size'img );
   Text_IO.Put_Line( "Natural bits is " & natural'size'img );
   Text_IO.Put_Line( "Positive bits is " & positive'size'img );
   Text_IO.Put_Line( "Short float bits is " & short_float'size'img );
   Text_IO.Put_Line( "Float bits is " & float'size'img );
   Text_IO.Put_Line( "Long float bits is " & long_float'size'img );
   Text_IO.Put_Line( "Long long float bits is " &
      long_long_float'size'img );
   Text_IO.Put_Line( "Our 3 item enumerated bits is " &
      enum'size'img );
   Text_IO.New_Line;
```

```
    Text_IO.Put_Line( "First integer is " & integer'first'img );
    Text_IO.Put_Line( "Last integer is " & integer'last'img );
    Text_IO.New_Line;
    Text_IO.Put_Line( "First enumerated is " & enum'first'img );
    Text_IO.Put_Line( "Last enumerated is " & enum'last'img );
    Text_IO.Put_Line( "Mica is in position" & enum'pos( mica )'img );
    Text_IO.Put_Line( "The third enumerated is " & enum'val(2)'img );
    Text_IO.New_Line;
    Text_IO.Put_Line( "The smallest float is" & float'small'img );
    Text_IO.Put_Line( "The largest float is" & float'large'img );
    Text_IO.Put_Line( "The number of digits in float is" &
       integer'image(float'digits) );
    Text_IO.Put_Line( "The size of the mantissa in bits is" &
       float'mantissa'img );
    Text_IO.Put_Line( "However, the CPU's mantissa is" &
       float'machine_mantissa'img );
end attrib;
```

Here are the results of the program on a Pentium II with gnat 3.11:

```
Some Basic Ada Attributes:
Boolean bits is 1
Short short integer bits is 8
Short integer bits is 16
Integer bits is 32
Long integer bits is 32
Long long integer bits is 64
Natural bits is 31
Positive bits is 31
Short float bits is 32
Float bits is 32
Long float bits is 64
Long long float bits is 9
Our 3 item enumerated bits is 2
First integer is -2147483648
Last integer is 2147483647
First enumerated is DOG
Last enumerated is MEGABYTE
Mica is in position 1
The third enumerated is MEGABYTE
The smallest float is 1.17549435082228751E-38
The largest float is 1.93428038904620299E+25
The number of digits in float is 6
The size of the mantissa in bits is 21
However, the CPU's mantissa is 24
```

10.5 Operations and Expressions

Ada Operator	*Description*	*C Equivalent*
and	Boolean and	&&
or	Boolean or	||
xor	Boolean xor	?
not	Boolean not	~
=	Equals	==
/=	Not equals	!=
abs	Absolute Value	?
mod	Integer modulus	%
rem	Float remainder	-
and then	Short circuited and	-
or else	Short circuited else	-
in	Value in range	-
not in	Short for not(...in...)	-

Boolean operations: **and**, **or**, **not, xor**
Comparisons: >, >=, <, <=, =, /=
Unary operations: +, -, **abs**
Binary Operations: +, -, *, /, **mod**, **rem**, &, **

C: C boolean operators always short circuit. In Ada, there are both short circuiting operations and operations that do not short circuit.


Short circuiting operations are not considered true operators, and as such, can't be overloaded (see below).


Membership Tests (**in** and **not in)** are not considered true operators and can't be overloaded.

```
if dog in aDogBreed then
    Put_Line( "The dog is a breed in the enumerated aDogBreed" );
end if;
if i in 1..10 then
    Put_Line( "I is between 1 and 10" );
end if;
```

1..10 is called a *range*. The range attribute returns the range of values for a type.

```
if salary not in MiddleManagementSalary'range then
    Put_Line( "The salary is not in the middle management type's range" );
end if;
```

C: Assignment is considered a statement, not an operator.

10.6 Variable Declarations

You define a variable as the variable name, colon, the type of information the variable will hold, and a semicolon.

```
totalSales : float;
```

This creates a new variable called totalSales that contains a real number. Some variations:

```
runningTotal : integer := 0;

-- this variable starts out at 0
companyName : constant string := "Bob's Widgets Inc.";
--companyName is set to Bob's Widgets Inc. and it can't be changed while
--the program is running
char1, char2 : character;
```

Complex variables references can be assigned a shorthand with a rename declaration.

```
sb : float renames EmployeeList( CurrentEmployee ).SalaryInfo.Bonus;

...
sb := 5.0; -- same as EmployeeList( CurrentEmployee ).SalaryInfo.Bonus := 5.0
```

C: There are no self-referential operators, such as C's +=.

10.7 New Types

Ada Statement	*Description*	*C Equivalent*
type	Create a new type.	typedef
subtype	Create a variation of an existing type	-

New types are defined with the **type** statement.

```
type aSalary is new float;
type aSmallSalary is new Salary range 0.0 .. 35_000.0;
```

When you create a new type, the type is considered to be incompatible with the type it is derived from. If you want to add a small salary to a salary, you'll have to use type casting, even though they are both floats.

```
totalSalary, BigSalary : aSalary;
smallSalary : aSmallSalary;
totalSalary := bigSalary + aSalary( smallSalary );
```

To type cast one type into another, use the type name and the value to convert in parantheses after it.

C: "(type) value" style of type casting doesn't work in Ada.
C: "Ada has stronger restrictions on typecasting. In C, for example, you can cast a character pointer as an integer pointer. Although this is considered a bad programming practice, it is allowed. Ada will not allow this kind of type casting.

Use **subtype** to create a type aSmallSalary that is compatible with aSalary.

```
subtype aSmallSalary is aSalary range 0..35_000.0;
```

Subtype can also be used to rename types.

```
subtype sb is aSalaryBonusForEmployeesNamedBobStevens;
```

In this example, sb is a short form for aSalaryBonusForEmployeesNamedBobStevens. "sb" is techncially called a "subtype mark", a term which sometimes appears in Gnat error messages. One common error, "subtype mark required in this context", indicates that there are several different types that could be used and you have to indicate to the compiler which should used.

10.7.1 Modular Types

Normally, if a calculation produces an answer too large for the variable type being assigned to, it creates an error. This is called a numeric overflow. For example, if int is an integer,

```
int := integer'max +1; -- check (KB)
```

will result in a constraint error because the answer is bigger than the biggest number an integer variable can contain.

Ada provides another type of integer called a modular type. The word "modular" comes from the mathematical modulus operation. Modular types never overflow. Instead, if a number becomes bigger than the largest possible number the variable can contain, the value of the modular type "wraps around" to the lowest value and continues to grow from there. If the int variable in the above example was a modular called int modular, then

```
int := intmodular'max + 1;
```

would result in int being assigned intmodular'min.

C: C integer types are all modular because C doesn't catch overflow errors.

There are no built-in modular types. All modular types are new types created by a type statement.

```
type mod10 is mod 10; -- value ranges from 0 to 9
```

10.7.2 Text_IO and New Types

To perform Text_IO input and output on new types, you have to create your own version of Text_IO for the new type. This process is called "instantiation", and is covered later in the section on generics. The format is

```
package MyNewTextIOPackage isnew PredefinedGenericIOPackage( mytype );
```

For example, to create a Text_IO package for the aSalary type,

```
package aSalary_Text_IO is new Ada.Text_IO.Float_IO( aSalary );
```

Ada creates a new package called aSalaray_Text_IO customized for the aSalary type. You can use your package just like one of the standard Ada numeric Text_IO packages.

```
Salary := 50_000.00

aSalary_Text_IO.Put( Salary );
```

The following table lists all the Text_IO packages that can be instantiated for a particular type.

Table : Predefined generic Text_IO packages for performing Input/Output

Base Type	Package
Complex Numbers	Ada.Text_IO.Complex_IO
Complex Numbers (wide text)	Ada.Wide_Text_IO.Complex_IO
Decimals (NQS)	Ada.Text_IO.Decimal_IO
Decimal Numbers (wide text)	Ada.Wide_Text_IO.Decimal_IO
Enumerateds	Ada.Text_IO.Enumeration_IO
Enumerateds (wide text)	Ada.Wide_Text_IO.Enumeration_IO
Fixed Points	Ada.Text_IO.Fixed_IO
Fixed Points (wide text)	Ada.Wide_Text_IO.Fixed_IO
Floating Points	Ada.Text_IO.Float_IO
Floating Points (wide text)	Ada.Wide_Text_IO.Float_IO
Integers	Ada.Text_IO.Integer_IO
Integers (wide text)	Ada.Wide_Text_IO.Integer_IO
Modulars	Ada.Text_IO.Modular_IO
Modulars (wide text)	Ada.Wide_Text_IO.Modular_IO

10.8 Aggregate Types

Arrays are tables of values with specific bounds. For example, to declare a table of 10 people

```
type peopleHeightList is array( 1..10 ) of integer;
```

The bounds can be specified as a type.

```
type peopleHeight is new integer range 1..10;

type peopleHeightList is array( peopleHeight ) of integer;
```

This creates an array from 1 to 10, the range of possible values for the type peopleHeight. This is the same as using array(peopleHeight'range).
You can create a multidimensional array by using more than one index to the table.

```
type peopleStats is array( peopleHeight, peopleAge ) of integer;
```

You can assign default values to an array using :=, the assignment operator. The list of values is enclosed in brackets. You can specify a specific value using **=>**, or specify a default with **others =>**.

```
PeopleHeights1 : peopleHeightList := (others => 0);

-- looks strange, but assigns 0 to all the heights in the entire list
peopleHeights2 : peopleheightList := ( 10, others => 0 );
-- first height is 10, others are 0
peopleHeights3 : peopleHeightList := (5 => 15, others => 0 );
-- fifth height is 15, others are 0
```

Arrays are accessed by specifying values for the indices. To get the height for the fifth element in the PeopleHeights1 array, you'd type:

```
Put_Line( PeopleHeights1( 5 )'img );
```

Records are collections of related information. Each subsection is referred to as a *field*.

```
type employeeProfile is record
    name : string( 1..80 );
    salary : aSalary;
    age : anAge;
end record;
```

You can assign default values to the fields in a record using :=, the assignment operator.

```
type employeeProfile isrecord
    name: string( 1..80 ) := (others => ' ');
    salary : aSalary := 30_000.0;
    age: anAge := 30;
end record;
```

Default values for whole records can be specified when record variables are declared.

```
Bob : employeeProfile := ("Bob Smith", 35_000.0, 37 );

Denise : employeeProfile := ( name => "Denise Jones", salary => 39_000.0,
    age => 42 );
```

In the above examples, we are creating a temporary record and then assigning that record to the variable. You can use this in the executable part of your program, not just in declarations. Ada will require a "subtype mark", an indication of what type of record you are making.

```
NewRec := employeeProfile'("Bob Smith", 35_000.0, 37 );
```

employeeProfile' indicates that the record we've built should be treated as an employeeProfile record.

Although this looks almost exactly the same as type casting, it isn't type casting. Consider the following:

```
J := long_integer'( 5 );

J := long_integer( 5 );
```

The first statement clarifies that 5 is a long_integer: this is a hint to the compiler that 5 should be treated as an long_integer. The second converts 5 from an integer to a long_integer.

Record fields are accessed using a period and the field name.

```
Bob.age := 37;
```

A variant record is a record that contains different sets of mutually exclusive information.

```
type employeeProfile( sex : aSex ) is record
    name : string( 1..80 );
    salary : aSalary;
    age : anAge;
    case sex is
    when male =>
    BeardLength : integer;
    when female => null;
end record;
```

(check syntax)

In this example, a male employee has an additional field called BeardLength.

(when you create a variant record, you must specify the descriminant).

C: In Ada 2005, to create a C-style union, declare the variant record with pragma unchecked_union.

10.9 Enumerated Types

Enumerated types (lists of identifiers) are created using a type statement.

```
type aDogBreed is ( Unknown, Boxer, Retriever, Shepherd, MixedBreed );
```

Different enumerated types may have the same values, but they are considered different from each other. For example,

```
type aCatBreed is ( Unknown, Siamese, MixedBreed );
```

shares two values with aDogBreed, but an unknown cat breed is considered different from an unknown dog breed. If Ada is confused by the ambiguity, you can clarify values with a subtype mark, e.g. aCatBreed'(MixedBreed).

Many of the common attributes work with enumerated types, including 'first, 'last, and 'range. Especially useful are 'pred (get a previous enumerated identifier) and 'succ (get the next enumerated identifer). Specific values can be assigned (using a for clause) so the enumerated type can reflect an external integer value (such as error codes).

```
with ada.text_io, unchecked_conversion;
use ada.text_io;

procedure enumeration_fun is
   -- a demonstration of Ada 95 enumerated types

   type vowels is ( 'a', 'e', 'i', 'o', 'u', none );
   -- characters may be used as well as identifiers.  The standard
   -- character sets are implemented this way.

   type aDogBreed is ( Jack_Russel, Labrador, German_Shepherd, Other );
   type aCanadianRegion is ( West_Coast, Arctic, Labrador, Other );
   subtype coldPlaces is aCanadianRegion range Arctic..Other;
   -- names may overlap between enumerated types

   type anErrorCode is ( None, IOerror );
   for anErrorCode use ( None => 0, IOerror => 7 );
   -- specific values may be assigned to enumerated identifiers

   function toInteger is new unchecked_conversion( anErrorCode, integer );

begin

   -- Basic enumerated type operations

   put( "The vowel 'u' has a position of" );
   put( integer'image( vowels'pos( 'u' ) ) );
   put_line( " in the list." );
   put( "The vowel after 'a' is" );
   put( vowels'image( vowels'succ( 'a' ) ) );
   put_line( "." );

   -- Using a regular enumerated.  Where an identifer belongs to
   -- two enumerated types, we have to apply a type qualifier when
```

```
    -- ambiguity comes up.

    put( "The item before German_Shepherd is " );
    put( aDogBreed'image( aDogBreed'pred( German_Shepherd ) ) );
    put_line( "." );
    put( "The item after Labrador (the region) is " );
    put( aCanadianRegion'image( aCanadianRegion'succ( Labrador ) ) );
    put_line( "." );
    put_line( "Listing of cold regions between 'Labrador' and 'Other':" );

    for cr in coldPlaces'(Labrador)..other loop
            put_line( aCanadianRegion'image( cr ) );
    end loop;

    -- Using an enumerated with assigned numbers.  To get the number
    -- we assigned, we need unchecked_conversion.

    put( "Error code " & anErrorCode'image( IOerror ) );
    put( " is in position" & integer'image(anErrorCode'pos( IOerror ) ) );

    put_line( "." );
    put( "IOerror has a value of" & integer'image( toInteger( IOerror ) ) );
    put_line( "." );
    put( "The code before IOerror is " );
    put( anErrorCode'image( anErrorCode'pred( IOerror ) ) );
    put_line( "." );
    put( "The code after NONE is " );
    put( anErrorCode'image( anErrorCode'succ( None ) ) );
    put_line( "." );
end enumeration_fun;
```

```
The vowel 'u' has a position of 4 in the list.
The vowel after 'a' is'e'.
The item before German_Shepherd is LABRADOR.
The item after Labrador (the region) is OTHER.
Listing of cold regions between 'Labrador' and 'Other':
LABRADOR
OTHER
Error code IOERROR is in position 1.
IOerror has a value of 7.
The code before IOerror is NONE.
The code after NONE is IOERROR.
```

The boolean type is implemented as an enumerated with two values, *true* and *false*. False is always the predecessor of true.

C: Ada enumerated types have more features than C's. You can use 'pred and 'succ to move through the list without casting the enumerated as an integer and using arithmetic. The position of an enumerated identifer is independent of its assigned value.

10.10 Procedures and Functions

Ada Statement	Description	C Equivalent

procedure	A subprogram that returns no value for expressions.	void f(...);
function	A subprogram that returns a value for expressions..	sometype f(...);
declare/begin	A nested block	{...}

There are several ways to break up an Ada program. First, a **procedure**, such as the main program, is a subprogram which returns no value.

```
procedure print_test is

begin
    Text_IO.Put_Line( "This is a test" );
end print_test;
```

C: a procedure is a void function.

The second is a **function**. A function is a procedure that can be used in a expressions because it returns a value. The value is returned with a return statement.

```
function AddOne( X : integer ) return integer is
begin
    return X+1;
end AddOne;
```

AddOne adds one to whatever is in the brackets. To add one to a variable called subtotal, you'd use it like this:

```
    Total := AddOne( SubTotal );
```

The value in the brackets is the parameter to the function. Parameters have modes: **in**, **out** or **in out**. In, which is the default if you specify a mode, means that the variable is treated as a constant. Out means the value is returned when the subprogram is finished. In out means the value goes into the subprogram, is changed, and is returned again when the subprogram is finished.

In Ada, functions can only have in parameters, but procedures can have all three.

```
procedure AddOne( x : in out integer ) is

begin
    X := x + 1
end AddOne;
...
AddOne( Subtotal );
```

C: There is no equivalent of pass-by-copy or pass-by-reference. See the section on interfacing C and Ada.

There is also a special **access** mode, which means that the parameter must be an access variable. This is especially useful with tagged records. You can also get around the in out restriction on functions with the access mode. It is discussed in 11.10.2.

C: An access variable is basically a pointer. These are described later.

An example of multiple parameters:

```
procedure DisplayCurrency( c : aCurrency;
```

```
    fieldWidth : integer; useDollarSign : boolean := true ) is
```
useDollarSign, the third parameter, has a default value of true. Here's now you can call this procedure:

```
DisplayCurrency( 1.97, 8 );

DisplayCurrency( 1.97, 8, false );
DisplayCurrency( 1.97, 8, useDollarSign => false );
DisplayCurrency( c => 1.97, fieldWidth => 8, useDollarSign => false );
```
If you really wanted to, you can also change the order of the parameters using the => convention.

```
    DisplayCurrency( fieldWidth => 8, useDollarSign => false, c => 1.97 );
```

You can also declare arbitrary blocks in Ada. These let you declare variables in the middle of a procedure or function or set apart the designated source code in it's own block. The form of a block is an optional **declare** section, and the block denoted by a **begin** and **end**.

```
procedure nested is
    Date : integer;
begin
    Date := 0;
    declare
        DaysInYear : constant integer := 365;
    begin
        Date := Date + DaysInYear;
    end;
end nested;
```
Here, DaysInYear only exists for the assignment statement that follows it.

The main reason for blocks is to add an exception hander to a particular line without having to write a one line procedure.

```
    begin

        Total := Total / Average;
    exception when numeric_error =>
        Text_IO.Put_Line( "Division by zero!" );
        Total := 0;
    end;
```
Here, if there's a numeric error during the division statement, it's caught and handled.

Operators are in-fix functions, ones that take parameters on their left and right. For example, "+" is an operator. Ada lets you redefine most of the standard operators so they work with types of your choosing. You enclose the operators' symbol in double quotes.

```
function "+"( e : employeeRecord, s : aSalary ) returns aSalary is

begin
    return e.salary + s;
end function;
```
The above function will let you add a salary to an employee record, which assumes you are referring to the salary field in the employee record.

Ada subprograms can have the same name as long as their parameters or return values are different. This is called *overloading*. If Ada can't determine which subprogram you are referring to, you'll receive an error when compiling. In the above example, "+" is overloaded since there's integer

addition, floating addition, and the other built-in meanings for "+", and our special salary addition we just defined.

C: Assignment is not an operator in Ada. Assignment overloading can be simulated with controlled tagged records and the Adjust procedure.

10.11 Control of Flow

Ada Statement	Description	C Equivalent
if	Conditional execution	if
for	Interative loop	for
while	Pretest loop	while
loop	Indefinite loop	-
exit	Loop exit	break
case	Multiple case conditional execution	switch
goto	Unconditional jump	goto

The **if** statement is, well, a standard if statement which you can find in many languages. Here's an example:

```
if x > 0 then
    Text_IO.Put_Line( "X is positive" );
elsif x < 0 then
    Text_IO_Put_Line( X is negative );
else
    Text_IO.Put_Line( "X is zero" );
end if;
```

Ada provides two expression short-circuiting operators: "or else" and "and then". Short-circuiting means that the expression will not be evaluated if the left side doesn't satisfy the condition.

```
    if x > 0 or else y > 0 then
```

In this case, y > 0 is only checked if x is not greater than zero.

There is a general purpose loop statement, **loop**. Loops are exited with an **exit** statement. Ada provides a shorthand, **exit when**, to exit on a condition.

```
loop
    X := X / 2.5;
    exit when X < 4.0;
    X := X + 1.0;
end loop;
```

C: There is no general purpose loop in C.

C: There's no equivalent of the C continue statement.

There is a pretest loop, **while**, which determines whether or not the loop should be entered or reentered based on an expression at the top of the loop;

```
while X >= 4.0 loop
    x := ( x / 2.5 ) + 1.0;
end loop;
```

C: This is the equivalent of a C while loop. There is no post-test loop, like C's do loop.

There is the standard **for** loop as well, to loop through a range of numbers. For loops in Ada may only loop by discrete numbers one unit at a time: no real numbers and no arbitrary stepping values. To go backwards through a range, use the word **reverse**.

```
for I in 1..10 loop
    Total := Total + 1;
end loop;
```

To loop through an entire range of a type, use the 'range attribute. To loop through all the dogs in an enumerated type called aDogBreed,

```
for dog in aDogBreed'range loop
```

Also note that dog is implicitly defined. You don't have to declare it. Ada understands the type from the loop and the loop variable exists for the duration of the loop.

C: For is much more structured than C's for.

Any loop can be exited with exit (or exit when). Ada allows you to label loops in order to exit out of several loops at once. In the following example, exit will exit the current loop and all loops up to and including OuterLoop. In this case, that's both loops.

```
OuterLoop: while y > 0 loop
    while x > 0 loop
        x := x - y;
        if x = 37 then
            exit OuterLoop;
        end if;
    end loop;
    y := y * 2;
    end loop;
```

There is a **case** statement as well, for testing a lot of different individual values. Ada requires a when others case to make sure that all possible cases are handed.

```
case DogBreed is

when Unknown =>
    Text_IO.Put_Line( "I don't know the breed" );
when Shepherd =>
    Text_IO.Put_Line( "It's a shepherd" );
when others =>
    Text_IO.Put_Line( "It's something else" );
end case;
```

Ada also as a **null** statement, which is a placeholder to use when a statement is expected but none is needed. For example, you can't have an empty if statement--there must be at least a "null;". Multiple cases can be included with the vertical bar, or a range can be specified with an ellipsis.

```
case TaxType is

when local_tax => Tax := Tax + LocalTax;
when federal_tax | govt_taxable => Tax := Tax + FederalTax;
when others => null; -- perhaps a warning would be better here instead
end case;
```

C: case is like switch, but the cases don't fall through.

Cases can also use ranges, such as 1..10 or TaxSubtype'range.

I really like goto's, and through a stroke of good luck, Ada includes a **goto**.
Goto labels are denoted with double angle brackets (unlike loop labels that use a colon).

```
for I in 0..10 loop
    -- some computations here
    if emergency then
        goto Help;
    end if;
end loop;
-- stuff that must not be executed in an emergency
<<Help>> Text_IO.Put_Line( "We are now down here" );
```

11 Advanced Ada Programming

11.1 Packages

Ada	*Description*	*C Equivalent*
`package`	Define a package	-

C:In C++, classes serve two purposes. First, they hide declarations. Second, they implement objects. In Ada, declaration hiding is done separately with packages: you do not need to use tagged records to use packages.

Large Ada programs are divided up into packages. These are collections of procedures, functions, variables and other declarations that serve a similar purpose. You can have packages to draw text on the screen, or packages to do accounts payable, or package to work with complex numbers.

To make it easier for different people to work on different packages, and to make packages self-documenting, Ada packages are divided into two parts. The**package specification** (or package spec) contains everything that other programs need to use your package. It contains all variables they can access, all the subprograms they can call, etc. For subprograms, you only use the subprogram header, the procedure or function line, to let the programmers know the name of the subprogram and what the parameters are.

C: Subprograms declared in package specs are similar to C (non-static) function prototypes in header files. Unlike prototypes, which are optional, declarations in the package specs are required for subprograms to be exported outside of the package.

```
package currency is

-- This package defines US and Canadian dollars and converts money
-- between these two countries.

subtype BaseCurrency is float;

-- All currencies have a common base type of anUnspecificCurrency.
-- Because the base type is separate, we can change the representation
-- of all currencies by changing anUnspecificCurrency.

type USDollars is new BaseCurrency;
type CanadianDollars is new BaseCurrency;

-- US and Canadian dollars are incompatible types.  Using the wrong
-- currency will cause an error.

procedure SetExchangeRates( USToCanadian : USDollars; CanadianToUS : CanadianDollars );
pragma import( stubbed, SetExchangeRates );
-- change the exchange rates

function ToCanada( money : USDOllars ) return CanadianDollars;
pragma import( stubbed, ToCanada );
-- convert US dollars to Canadian dollars

function ToUS( money : CanadianDollars ) return USDollars;
pragma import( stubbed, ToUS );
-- convert US dollars to Canadian dollars
-- convert Canadian dollars to US dollars

end Currency;
```

In GNAT, you must save this package under the filename "currency.ads" (.ads for Ada Spec). Here we create different money values two functions to convert between the different rates and a procedure to set the exchange rates to use in the functions. Notice there is no main program.

pragma import(stubbed, ...) indicates that the actual subprograms are incomplete. However, the package spec can still be compiled to check for errors, and other programmers can refer to your spec when compiling their programs or packages. Any attempt to execute a stubbed subprogram will raise a PROGRAM_ERROR exception.

When all the specs for the project are checked, remove the pragma import's from the headers when you begin writing the actual subprograms.

```
package currency is

-- This package defines US and Canadian dollars and converts money
-- between these two countries.

subtype BaseCurrency is float;

-- All currencies have a common base type of BaseCurrency.
-- Because the base type is separate, we can change the representation
-- of all currencies by changing BaseCurrency.

type USDollars is new BaseCurrency;
type CanadianDollars is new BaseCurrency;

-- US and Canadian dollars are incompatible types.  Using the wrong
-- currency will cause an error.

-- Set the exchange rate between US and Canada and two functions to
-- convert between the currencies.

procedure SetExchangeRates( USToCanadian : USDollars; CanadianToUS : CanadianDollars );
-- change the exchange rates

function ToCanada( money : USDOllars ) return CanadianDollars;
-- convert US dollars to Canadian dollars

function ToUS( money : CanadianDollars ) return USDollars;
-- convert Canadian dollars to US dollars

end Currency;
```

To complete the package, we create a package body with the rest of the details, including the completed subprograms. With the implementation details hidden in the package body, other programmers don't have to worry about how currency is actually handled.

```
package body currency is

-- This package defines US and Canadian dollars and converts money
-- between these two countries.

USTOCanadaExchangeRate : USDollars;
-- USDollars because it will always be multiplied with a US amount.
-- Multiplying with a different currency will cause an error.

CanadaToUSExchangeRate : CanadianDollars;
-- CanadianDollars because it will always be multiplied with a Canadian amount

procedure SetExchangeRates( USToCanadian : USDollars; CanadianToUS : CanadianDollars ) is
begin
    USToCanadaExchangeRate := USToCanadian;
```

```
    CanadaToUSExchangeRate := CanadianToUS;
end SetExchangeRates;

function ToCanada( money : USDOllars ) return CanadianDollars is
begin
    return CanadianDollars( money * USToCanadaExchangeRate );
end ToCanada;

function ToUS( money : CanadianDollars ) return USDollars is
begin
    return USDollars( money * CanadaToUSExchangeRate );
end ToUS;

end Currency;
```

Notice we have access to everything we defined in the package spec--we don't need to repeat anything in the body.

Because the two exchange rate variables are defined inside the package body, they are invisible to other programmers.

Save this package body as "currency.adb" (.adb for AdaBody). Make sure all pragma import(Stubbed,'s are removed for the finished subprograms. Compile both and you have a working package.

To use your package in a program, use the **with** statement.

```
with Ada.Text_IO, currency;
procedure currencyTest is
begin
    Currency.SetExchangeRates( 1.5, 0.7 );
    Text_IO.Put( "1 Canadian dollar is" );
    Text_IO.Put( currency.ToUS( 1.0 )'img );
    Text_IO.Put_Line( " US DOllars." );
end currencyTest;
```

Running the program, we get the following result.

```
1 Canadian dollar is 7.00000E-01 US DOllars.
```

To have Ada check which package a subprogram belongs to, and avoid typing the package name constantly, use the **use** statement.

```
with currency;
use currency;
...
SetExchangeRates( 1.5, 0.7 );
```

If the use statement creates an ambiguity, Ada will warn you that it can't determine which package SetExchangeRates is in.

Package bodies may have main programs, a block at the end of all declarations marked with **begin**. This allows you to setup your package before it's actually used. In this case, we don't need one.

Package specs may have **private** sections at the end of the spec. There will be times when you will have to provide information so that Ada can compile your spec, but you don't want the application programmer to be able use this information. For example, you might create a package to maintain a customer list, but you don't want the programmer to access the internal fields of a customer list since you might change them at some point.

Just about anything in a package spec can be marked **private**, and the compiler expects the details to be specified in the private section. Declarations can also be **limited private**, meaning that besides having the details inaccessible to the programmer, the programmer can't assign between variables of the type. For example, use limited private if you think you may include pointers in the type at some time in the future.

```
package CustomerList  is

   type aCustomerNumber is new positive range 1..1000;
   type aCustomerList is limited private;

private

   type aCustomerArray is array( aCustomerNumber ) of string(1..120);

      type aCustomerList is record
         CurrentCustomer : aCustomerNumber;
         Customers : aCustomerArray;
      end record;

   -- no pointers yet, but we may some day, so it's limited private
   end CustomerList;
```

In this example, a programmer can declare customer lists, but he cannot access the fields CurrentCustomer or Customers (because it's private), nor can he copy lists with assignment statements (because it's limited private).

C: In C++, privacy is limited to classes. In Ada, virtually anything can be private.

Packages can have children. A **child package** is a package that extends the capabilities of the parent package. You can use child packages to add features to existing packages, such as the standard Ada libraries, or to break up large packages into groups of related functions.

Suppose you had a package called accounting that contains tools to handle accounting in general. You can create a child package for accounts payable begins like this:

```
   package Accouting.Accounts_Payable is
```

In GNAT, save this package spec as "accounting-accounts_payable.ads", with a minus sign instead of a period.

A child package inherits the package spec from it's parent package. You can access anything in the accounting package, including anything private to which access is normally denied.

When a program uses the statement

```
   with Accouting.Accounts_Payable;
```

the parent package, accounting, is automatically with'ed as well (although you still have to use separate use's).

In large projects, child packages have the advantage of reducing the number of dependant packages that you have to name (that is, it reduces the number of "includes" in C terminology). The name of the packages indicates packages that the child package depends on. Suppose that you are creating an 3D animation package. You might break up your project as follows:

```
three_d.ads
   three_d-opengl.ads
   three_d-animation.ads
      three_d-animation-sequences.ads
```

For the three_d.animation.sequences package, Ada knows that you will need three_d.ads and three_d.animation: these are implicitly **with**'ed and **use**'ed. So package lists at the start of a package are shorter when you use child packages.

Child packages can also be **private** so that they can only be accessed by their siblings. In the example above, if three_d.opengl is a binding to OpenGL that is not supposed to be used outside of the three_d hierarchy, then it can be declared as a **private package**. This will prevent outsiders from using your OpenGL binding while leaving it accessible to three_d.animation and three_d.animation.sequences..

In Ada 95, private child package cannot be **with**'ed in a package spec even if it's only used in the private part of the spec. In Ada 2005, use

private with in a package specification if you need to refer to a private child package in the private part of a package spec.

11.2 Controlling Elaboration

Ada Pragma	*Description*	*C Equivalent*
pragma `Pure;`	The simplest kind of package.	-
pragma `Pure_Function( function_name);`	When an entire package isn't pure...	-
pragma `Preelaborate;`	A package with simple elaboration.	-
pragma `No_Elaboration_Code;`	Similar to Preelaborate.	-
pragma `Elaborate( package );`	Force "package" to be elaborated first, if possible.	-
pragma `Elaborate_Body( package );`	Elaborate the body before the specification.	-
pragma `Elaborate_All;`	Short for Elaborate() for each package.	-

Elaboration is the initialization of a package that's done before the main program of the package is executed. Assigning values to constants, for example, is elaboration.

C: Since C is a has no packages, the order of elaboration between files is determined strictly by the compilation order.

For most projects, gnat will work out a good elaboration order on its own. However, large projects with packages referring to each other in complicated ways may require complex elaboration orders. gnat searches every possible elaboration order until it finds one that solves any ambiguities. To speed up projects with slow elaboration times, Ada and gnat provide a number of pragmas to give the compiler hints on the best compilation order and to solve any potential ambiguities.

11.2.1 First line of defense: Pure, Preelaborate and No_Elaboration_Code

Pragma Pure and **Preelaborate** are elaboration restrictions. They are hints to the Ada compiler to cut down the compiler's work when trying to solve elaboration order. Pragma Pure tells Ada that the package requires no elaboration and contains no global variables. For example, a package of nothing but type declarations (with no default values) is pure.

package money_types **is**

```
   pragma pure;

   -- a simple package, nothing to elaborate

   subtype aCurrency is float;
   type aSalary is new aCurrency;

end money_types;
```

Pragma Preelaborate tells Ada that the package requires minimal elaboration. You should try pragma pure, and if it fails, try pragma preelaborate.

gnat 3.11 introduced the gnat specific **pragma No_Elaboration_Code**. Sometimes this will work when Preelaborate will not.

Sometimes you can declare a function as pure using **pragma Pure_Funciton.** A pure function is one with no side effects and one that's value doesn't change when executed with the same parameters each time. If the only thing standing in your way to using pure or preelaborate are some functions used to assign initial values, try declaring them as pure functions.

If your package or program fails to meet the elaboration restriction requirements, the compiler will issue an error.

11.2.2 Second line of defense: Elaborate, Elaborate_Body, Elaborate_All

Sometimes the hints are not enough. For example, a package that assigns a value to constant using a function like

```
   Sin45 : float := Ada.Numerics.Elementary_Functions.Sin( 45, 360 );
```

is neither pure nor preelaborate because a function must be called when the package is initialized. In these cases, you can tell Ada specifically which package should be elaborated first.

Pragma Elaborate(package) tells Ada the specified package should be elaborated first. For example, all generics must be elaborated before they are used, so it's a good idea to use pragma elaborate on every generic package.

```
with generic_linked_list;
pragma Elaborate( generic_linked_list );
```

Pragma Elaborate_All indicates that a particular package and all the packages used by that package must be elaborate before the current package. For example, if package userio uses package common,

```
with userio;
pragma Elaborate_All( userio );
```

will elaborate both userio and common prior to this package

Because Elaborate_All will affect multiple packages, it can cause unnecessary binding errors when used indiscriminately. This pragma, when used everywhere, effectively requires all packages to be arranged in a strict hierarchy. Make sure this is the effect you want.

11.2.3 Other Elaboration Pragmas

Another pragma, **pragma Elaborate_Body,** forces the package body to be elaborated before the package specification. This is especially useful in generic packages.

Pragma Elaborate and **Elaborate_All** can also be used to resolve ambiguous elaborations.

11.3 Objects

Ada	*Description*	*C Equivalent*
`type...tagged record`	Define an object	`class`
`type...new` `parenttype` **`with record`**	Extend an object	`class ...: parenttype`
`type...access all` `sometype`	Define a pointer to a class	`sometype *`
`'class`	Class-wide type	`virtual`
`abstract`	Abstract types/subprograms	`function...=0`

C: Ada developed its own object oriented terminology because C's terminology can be ambiguous and confusing. Ada differentiates between a class and the structure that defines that class.

An object in Ada is known as a **tagged record**. That is, it is a normal Ada record that has an invisible tag attached to it. The record

```
type anEmployeeProfile is record
   name : string(1..80);
   age : anAge;
end record;
```

can be changed to a tagged record by adding the keyword **tagged**:

```
type anEmployeeProfile is tagged record
   name : string(1..80);
   age : anAge;
end record;
```

Although these two records look the same, if we use the 'size attribute to see how much memory the records take up, we'll see that the tagged record is bigger. The extra space is used to store the invisible tag.

Unlike normal records, fields can be added tagged record and a new tagged record can be created. This is called extending the record. To create a related record with additional fields, we use the keyword **new**:

```
type anHourlyEmployee is new anEmployeeProfile with record
   hourlyRate : float;
end record;
```

A tagged record extended in this way has all the fields of anEmployeeProfile, but has the additional field of hourlyRate. anEmployeeProfile and anHourlyEmployee are said to be in the anEmployeeProfile class: the class is the collection of anEmployeeProfile and all record extended from it.

Now we can create a access type (commonly called a pointer, though technically it isn't a pointer) to any record in the class:

```
type anEmployeeProfilePtr is access all anEmployeeProfile'class;
```

This pointer can be assigned either anEmployeeProfile record or anHourlyEmployee record. This is the purpose of the tagged record's invisible tag. The tag indicates the type of tagged record a pointer points to since these kinds of pointers can refer to more than one type of tagged record. This is sometimes called*late binding*.

```
ptr1 : anEmployeeProfilePtr := new anEmployeeProfile( "Bob Smith", 45 );
ptr2 : anEmployeeProfilePtr := new anHourlyEmployee( "Denise Jones", 37, 17.50 );
```

Access variables are null until assigned a different value. They are the only variables in Ada to have a default value.

There may be cases where you want to extend a type without adding new fields. Ada provides a shorthand phrase for this. For example, if you want to distinguish hourly employees that work at night as being separate from other hourly employees, use

```
type aNightHourlyEmployee is new anHourlyEmployee with null record;
```

In complex classes, there will be times when you'll want to define a record that you never intend to assign variables to. For example, anEmployeeProfile doesn't contain enough fields to completely describe any employee: only the tagged records derived from anEmployeeProfile are usable. When particular record exists only to be extended it called an abstract record. You declare abstract records with the keyword **abstract**:

```
type anEmployeeProfile is abstract tagged record
   name : string(1..80);
   age : anAge;
end record;
```

If you try to create anEmployeeProfile record, Ada will report an error since you said that this record can only be extended into other records.

Ada requires that subprograms that work with tagged records be declared immediately after the type declaration. Each procedure or function can only take one tagged record as a parameter.

C: Methods are normal subprograms with an object being referred to as a parameter. myobj.method(x) would be method (myobj, x) or method(x, myobj) in Ada.

```
type aSalaryEmployee is new anEmployeeProfile with record
   salaryRate : float;
end record;

procedure SetSalaryRate( s : in out aSalaryEmployee'class; rate : float ) is
begin
   s.salaryRate := rate;
end SetSalaryRate;

function GetSalaryRate( s : aSalaryEmployee'class ) return float is
begin
   return s.salaryRate;
end GetSalaryRate;
```

We've declared two one line subprograms that will work on any tagged record derived from aSalaryEmployee.

C: Ada does not require constructors or destructors. Creating objects with these are discussed below.

Subprograms can be marked as unusable in the same way as abstract tagged records. An abstract procedure or function is a placeholder. Declaring one requires that all types extended from this type

must have this subprogram defined. If any do not have this subprogram, Ada will report an error. For example, if anEmployeeProfile had a procedure like

```
procedure WriteEmployeeName( e : anEmployeeProfile ) is abstract;
```

all employee profile records would be required to have a procedure called WriteEmployeeName. ASalaryEmployee will have a compilation error unless we add such a function:

```
procedure WriteEmployeeName( e : aSalaryEmployee )is
begin
   Text_IO.Put_Line( "Employee Name: " & e.Name );
end WriteEmployeename;
```

WriteEmployeeName could also use aSalaryEmployee'class to refer aSalaryEmployee or any records we extend from it.

To avoid ambiguity, only one tagged record subprogram can refer to any one type. This is different from some other object oriented languages where you can*override* a classwide subprogram with one that refers to a specific type. The advantage of no overriding is that someone reading your class knows exactly which subprogram will be used for a particular tagged record type--they don't need to read the entire class to make sure the subprogram isn't overridden later on. The disadvantage is that if you can't use a classwide type, you'll have to write subprograms for each and every type in that class. In these cases, typecasting is useful.

Tagged record types can be typecast as other tagged record types (in the same class) using typecasting. You need to do this if you want a dispatching call inside of a dispatching call. For example, if anEmployeeProfile has a GetSalaryRate function, we could call it by:

```
procedure WriteEmployeeSalary( e : aSalaryEmployee'class ) is
begin
   Text_IO.Put_Line( "The salary is" & GetSalaryRate( anEmployeeProfile( e ) ) );
end WriteEmployeeSalary;
```

In Ada 2005, objects can be called using prefix notation (e.g. employee.WriteEmployeeSalary) as well as Ada notation (WriteEmployeeSalary(employee).

11.4 Objects with Automatic Initialization/Finalizaton

Ada Controlled Object Call	*Description*	*C Equivalent*
`Initialize`	Initialize an object	*constructor*
`Adjust`	Fix object after assignment	*copy constructor*
`Finalize`	Clean up an object	*destructor*

Basic Ada tagged records don't do anything special when they are created or destroyed. If you want special actions to be executed automatically when a tagged record is created or destroyed, you can use the **Ada.Finalization** package. In this package is defined a special tagged record called Controlled. When you extend this tagged record, you can overload special procedures that Ada will automatically execute when necessary. This saves you from having to explicitly call such procedures yourself.

```
type businessDepartment is new Finalization.Controlled with record
   departmentHead : aEmployeeProfilePtr := new
      aSalaryEmployee ( "Bob Smith", age => 45, rate => 42_000.0);
end record;
```

In this example, every time you allocate a businessDepartment variable, DepartmentHead is initialized with a dynamic allocation. We could write a procedure to free up this memory, but then we would have to remember to call it every time a variable is about to be discarded. A better way to handle this is to let Ada do the discarding for us. **Finalize** is the name of the procedure Ada calls when cleaning up a controlled tagged record. We create our own Finalize for our businessDepartment tagged record:

```
procedure Finalize(bd : in out businessDepartment) is
begin -- Finalize
   Free( bd.departmentHead );
end Finalize;
```

Now Ada will automatically run this procedure before any businessDepartment variable is destroyed and we never have to worry about forgetting to free up the memory used by departmentHead.

C: Finalize is the destructor.

There are two other such automatic procedure we can use with controlled tagged records. Procedure **Initialize** is used when an object is first created, and procedure **Adjust** is used after an object is assigned in an assignment statement.

 Adjust is very smart. Temporary storage is used for self-assignment. If Adjust fails because of an exception, Finalize is not executed.

C:Adjust is like a C++ copy constructor, except that Ada will copy the object before calling Adjust. With a copy constructor, you must copy the object yourself in the body of the constructor. Unlike a copy constructor, Adjust only executes during assignment.

```
with text_io, sample;
use text_io, sample;

procedure controlledtest is
   srp1, srp2 : SampleRecPtr;
   sr3 : SampleRec;
begin
   Put_Line( "(This is the first line of the program)");
   New_Line;
   Put_Line( "This is an example of controlled tagged records:" );
   New_Line;
   Put_Line( "Executing 'srp1 := new SampleRec'");
   srp1 := new SampleRec;
   New_Line;
   Put_Line( "Executing 'srp2 := srp1' (copying a pointer)" );
   srp2 := srp1;
   New_Line;
   Put_Line( "Executing 'sr3 := srp1.all' (copying a record pointed to)" );
   sr3 := srp1.all;
   New_Line;
   Put_Line( "(This is the last line of the program)");
end controlledtest;
```

```
with ada.finalization;
use ada.finalization;

package sample is
```

```
      type SampleRec is new Controlled with null record;
      type SampleRecPtr is access all SampleRec;
      -- sample controlled tagged record (and a pointer to same)

      procedure Initialize( sr : in out SampleRec );
      procedure Finalize( sr : in out SampleRec );
      procedure Adjust( sr : in out SampleRec );
      -- these are required for controlled tagged records

end sample;
```

```
with Ada.text_io;
use Ada.text_io;

package body sample is
   -- just print messages to show that these are working

procedure Initialize( sr : in out SampleRec ) is
   begin
   Put_Line( "Initialize: Initialized tagged record" );
end Initialize;

procedure Finalize( sr : in out SampleRec ) is
begin
   Put_Line( "Finalize: Finalized tagged record " );

end Finalize;

procedure Adjust( sr : in out SampleRec ) is
begin
   Put_Line( "Adjust: Adjusted tagged record " );
end Adjust;

end sample;
```

Here is the output. The records being affected are noted in bold.

Initialize: Initialized tagged record **[sr3]**
(This is the first line of the program)

This is an example of controlled tagged records:

Executing 'srp1 := new SampleRec'
Initialize: Initialized tagged record **[srp1.all]**

Executing 'srp2 := srp1' (copying a pointer)

Executing 'sr3 := srp1.all' (copying a record pointed to)
Finalize: Finalized tagged record **[sr3 (before record is copied)]**
Adjust: Adjusted tagged record **[sr3 (after record is copied)]**

(This is the last line of the program)
Finalize: Finalized tagged record **[srp1.all]**
Finalize: Finalized tagged record **[sr3]**

Ada also provides a second tagged record, **Limited_Controlled**, which is a controlled record that can't be assigned. Consequently, it has no adjust procedure.

11.5 Multiple Inheritance

Like a tree, tagged records can only be extended from a single parent. Extending from multiple parents is called multiple inheritance.

In Ada 95, muliple inheritance is not allowed In Ada 2005, multiple inheritance is handled by interfaces.

In Ada 95, the Ada designers considered multiple inheritance a feature that adds ambiguity to a language: for example, if an employee tagged record has two different parents, each with a salary field, do you merge the salary fields into one or do you have two copies of the field in your record? Because there is no consistent solution to this kind of problem, the Ada designers decided to not to support multiple inheritance.

However, you can add tagged records as nested *fields*, just like you would a normal record. This workaround guarantees that each object will only have one parent it can extend from.

```
type aClass is tagged record
   F1 : integer;
end record;

type anUnrelatedClass is tagged record
   U1 : integer;
   end record;

type anMIExample is new aClass with
   UC : anUnrelatedClass; -- a field, not an extension
end record;
```

anMIExample is a tagged record belonging to aClass, not to anUnrelatedClass. If you extend anMIExample, it will inhert F1 from its parent class. Since anUnrelatedClass UC is nested, you can still access it as a field using the prefix "UC.".

[BETTER EXAMPLE -- KB]

In Ada 2005, a new feature called "interfaces" is available. The interfaces handle the problem of multiple parents with conflicting fields.

 Ada interfaces are similar to Java interfaces.

You declare an interface is a similar way to declaring a tagged record.

```
type car is interface;
type buyer is interface and car;
```

Like an object, procedures and functions can be declared that use the interface.

```
type car is interface;
type buyer is interface and car;
procedure selling_price( c : access car'class; b : access buyer'class );
```

"selling_price" is a procedure that has two interfaces (not objects) as parameters: actually, two pointers to anything in the car or buyer hierarchies.

To use an interface, included it in the object declaration using the word **and**:

```
   type household is tagged private;
   type canadian_household is new household and buyer with private;
```

The class "canadian_household" is implementing the interface "buyer". "selling_price" is available. If it was abstract, then the tagged record would have to declare a new "selling_price" to be complete.

[AGAIN, BETTER EXAMPLE WOULD BE GOOD HERE -- KB]

11.6 Private Objects

Unless a tagged record class is very small, it's kept in it's own package. When you put a class in a package, you can use the package to hide the details of the class and to make parts of the class inaccessible to the outside world.

```
package customers is

   type AbstractCustomer is abstract tagged private;

   type BasicCustomer is new AbstractCustomer with private;

private

   type AbstractCustomer is new abstract tagged record
      name : string( 1..80 );
   end record;

   type BasicCustomer is new AbstractCustomer with
      SalesCategory : integer;
   end record;

end customers;
```

By declaring a tagged record as private, we eliminate all access to the fields in the record. Anybody who wants to change the value of the fields must do it through any subprograms we provide. In object oriented programming, it's a good idea to make as many tagged records private as possible, the same as other private types. That way, if you want to change the contents of the tagged record, programs that use your class will not have to be changed.

In Ada 95, this reliance on packages created a problem with complicated declarations where packages referred to one another (the so-called "Multi-Package Cyclic Type Structures" problem). For example, you couldn't declare an "owner" class containing a pointer to the pet and a "pet" class containing a pointer to the owner because they referred to one another.

The Ada 2005 standard fixes this problem with limited with clauses. A regular **with** shows everything in a package specification. A **limited with** imports a package that contains only type and package declarations and Ada understands that the types may not be completely defined. This creates a kind of "forward declaration" for packages where common features can be named before they are used.

```
   package animal_classes is
      type animal is tagged;
      -- animal is some kind of tagged record which has not been defined yet
   end animal_classes;
```

Then in package for pet owners:

```
    limited with animal_classes;
    package owners_classes is
        type animal_ptr is access all animal_classes.animal;
        -- a pointer to a type that hasn't been defined yet
...
    end animal_classes;
```

The package for pet owners can now create pointers to pet animals even though the "animal_classes" package hasn't been defined yet. At some later point you'll define the completed "animal_classes" package.

In Ada 2005, you explicitly state whether your are overriding an abstract subprogram or overloading a non-abstract subprogram by explicitly typing "overriding" or "not overriding" before the declaration. This helps to detect errors.

C: Ada does not have protected objects, but protected objects can be simulated using Ada packages.

11.7 Generics

Ada allows you to create source code templates, called *generics*. With generics, you can specify a routine with only general information about the kinds of types the routine can work on. Later a programmer instantiates the generic by giving a type. Generics are useful for things like sort routines, which work basically the same way on different data types.

C: Generics are similar to C++ templates.

Contrary to what you might think, Ada must compile the generic packages before they are used. Although they act as templates, Ada needs the information in the .ali files to solve some rare file dependancy issues. A .o object file is also created, although it may be empty.

Ada contains several standard generic procedures. The most important one is **unchecked_deallocation**. This long winded procedure deallocates memory allocated with **new**. In order to free up memory, you must instantiate an unchecked_deallocation for each access type you are going to use new on. You have to specify both the access type and type of data that is being pointed to.

```
with unchecked_deallocation;

type booleanPtr is access all boolean;

procedure Free is new unchecked_deallocation( boolean, booleanPtr );
```

Free is a new version of unchecked_deallocation compiled for booleanPtr's to a boolean type.

Ada was designed for the possibility automatic storage recovery, that everything that was allocated in a subprogram would be deallocated automatically when the subprogram is left. Unfortunately, gnat was implemented without this feature and all memory has to be explicitly deallocated.

Another standard generic you'll run into is **unchecked_conversion**. This converts a variable of one type to another by physically copying to data from one to the other, such as an array of 8 bits to a short_short_integer.

Although you can write generic subprograms, most of the time you use generics will be for creating generic packages. Generic packages have a spec and body like normal Ada packages, but they begin with a list of parameters to the package that must be filled in when the generic is instantiated.

```
generic

   -- these are the parameters for the generic

   type ListElement is <>; -- unspecified list element

   procedure ">="( left, right : ListElement); -- a procedure to sort by

package SimpleList is

   type List is array( 1..100) of ListElement;
   procedure Add( l : list; e : ListElement);
   procedure Sort( l : list );
   procedure Display( l : list );

end SimpleList;
```

(check--KB)

In this example, SimpleList takes some kind of data type called a ListElement, the items that compose the lists. Besides <>, Ada offers a number of other non-specific type descriptors to give the compiler an idea of what kind of types are acceptable. Since the ListElement could be an aggregate and we can't assume we can do simple comparisons, the programmer must also specify a procedure to sort the elements by.

Once you write the package body (no generic section in the package body, just a regular package body), you can instantiate the generic package in a program. After instantiation, use the package like any other package.

```
with SimpleList;

procedure ListTest is

package BooleanList is new SimpleList( boolean, ">=" )

   -- in this case, the normal ">=" will sort booleans

begin

   BooleanList.Add( mylist, true );
   BooleanList.Add( mylist. False );

end ListTest;
```

Now you'll notice that generics and tagged records share a lot of capabilities. You can use both to write subprograms that work on a variety of types. Tagged records are referred to as *dynamic polymorphism*, because which subprogram to call gets determined while the program is running. Generics are referred to as*static polymorphism*, because the subprograms are generated when the generic is instantiated by the compiler and which subprogram to call is known when the program is compiled. The better approach depends on what you are doing. In general, generics run faster but take up more space because the compiler generates separate subprograms for each type. Tagged records take up less space but tend to run slower.

Variant records and tagged records, likewise, share much in common. Although variant records can be simulated with tagged records, you'll need to decide which is the best choice depending on what you are trying to accomplish. Variant records tend to be smaller and faster, but are harder to extend than tagged records.

Since tagged records are naturally used to create variables that are similar to one another, you might wonder if you'd ever create a single variable of a tagged record type. These are called *singletons*, and are used in frequently in languages like C++. They are popular because they have a specific elaboration order and provide access to features only available in objects (such as private members). Programmers doing this have no need to create a class of several objects. However, Ada has an easily controlled elaboration process and features such as privacy are not specific to tagged records. As a result, there is rarely a need for singletons in Ada programs.

11.8 Exceptions

Ada	*Description*	*C Equivalent*
`e : exception`	Declare an exception	-
`raise e`	Raise/throw an exception	`throw` *`type`*
`exception` clause	Handle/catch an exception	`try...catch`

C:In C++, exceptions are performed by throwing types (typically objects). In Ada, an exception is a separate type. Only exceptions can be thrown--you can't throw types. Exceptions aren't related to objects in any way.

When you throw in C++, you do so in a try block. In Ada, you can throw an exception in any block. The exception clause (equivalent to `catch`) at the end of the block is optional.

What do you do when something unexpected error occurs? Unexpected errors are called exceptions in Ada. For example, running out of memory is an exception.

Ada has a number of predefined exceptions, and examples of when they occur:

- CONSTRAINT_ERROR - number out of range, like assigning -1 to a positive variable
- NUMERIC_ERROR - dividing by zero
- SELECT_ERROR - caused by task select statement with no else part
- STORAGE_ERROR - running out of memory
- TASKING_ERROR - failure of a task to handle an entry call
- PROGRAM_ERROR - hitting the end of a function without returning anything
- ASSERT_ERROR - a pragma assert failed

You can turn off checking for most of these errors using pragmas, but they are usually a sign that something if fundamentally wrong with your program. Gnat provides a compiler option to turn these checks off for the release version of a program.

The standard libraries have additional exceptions defined.

To handle an exception, you need an **exception** part at the end of a subprogram. When the exception is raised (occurs), execution immediately jumps down to the exception part. The exception part contains a list of exceptions to check for, plus an others part, similar to a case statement.

procedure exceptional(Total : out integer) is

```
begin
      -- do some stuff
exception
    when constraint_error => Total := 0;
    when storage_error => Total := -1;
    when others => raise;
end exceptional;
```

Raise causes the exception to be reraised to the calling subprogram.

If one subprogram doesn't handle the exception, it's raised in the subprogram that called it. This continues until the exception is handled by an exception part, or the main program is it. In the worst case, if the main program has no exception part, the program terminates and the exception name is printed on the screen.

One use for exception parts to deallocate access types so the memory isn't lost when unexpected errors occur.

You can define and raise your own exceptions.

```
procedure exceptional2 is
    Accounting_Error : exception;
```

C: Ada exceptions do not carry any additional information when raised. You can simulate error messages and other information with global variables.

Pragma Assert provides an exception for debugging programs. Assert is described in 8.2.

For more advanced manipulation of exceptions, you'll need to use Ada.Exceptions and its related packages. These are described in 12.15.

11.9 Dynamic Allocation

Ada	*Description*	*C Equivalent*
`new`	Allocate new memory	`malloc/mallopt`
`unchecked_deallocation`	Deallocate memory	`free`

Dynamic allocation is reserving memory for variables while the program is running. The memory is allocated from a region called the **default storage pool**. GNAT sometimes refers to this as the **unbounded no reclaim pool** because the only limit on the amount of memory you can allocate is the physical limit of the machine, and memory is not reclaimed through garbage collection.

C: The default storage pool is effectively the "heap".

Ada uses access types to create a handle to dynamically declared variables.

```
type IntegerPtr is access all Integer;
ip : integerPtr := new Integer;
```

In this example, IP access a dynamically declared integer. IP is only large enough to hold the address of where the integer is located. To access the integer, we have to add the suffix .**all** to IP. This is called dereferencing.

```
ip.all := 5;
```

If you are dereferencing multiple pointers, the all is only required at the end to indicate that the final pointer is to be dereferenced (for example,. ptr1.ptr2.ptr3.all).

The word **all** in **access all** is not strictly required. If all is included, the IntegerPtr type will be compatible on any other integer pointer. Without all, Ada imposes certain restrictions on what the access type can point to, but in general always use access all.

Memory allocated with new is freed with **unchecked_allocation** (see the section on generics).

You can assign initial values when you create a new dynamic variable. In the following example, we declare, allocate memory and assign an initial value all at once.

```
ip : integerPtr := new Integer( 5 );
```

To create a record with a pointer to itself, have an empty type statement with the record first:

```
type LinkedListRecord;

type LinkedListPtr is access LinkedListRecord;
type LinkedListRecord is record
   info : string(1..80);
   next : LinkedListPtr;
end record;
```

To point to variables you've declared, you must declare those variables as **aliased** to indicate they can be pointed to. To get the address of something, use the**'access** attribute.

```
type CustomerArray is array(1..100) of CustomerRecord;
type CustomerArrayPtr is access all CustomerArray;

ca : aliased CustomerArray;
cp : CustomerArrayPtr := ca'access;
```

cp now points to ca. Individual array elements can also be aliased.

```
type CustomerArray is array(1..100) of aliased CustomerRecord;

type CustomerArrayPtr is access all CustomerArray;

  ca : CustomerArray;
  c15p : CustomerArrayPtr := ca(15)'access;
```

Ada will give you an error when you try to use 'access when the object pointing to may disappear after the pointer does. If you're absolutely certain that this won't happen, you can circumvent the error by using '**unchecked_access**.

An access type is necessarily just the address of a dynamic object. To get the address of an access type, it's best to use gnat's generic package**System.Address_To_Access_Conversions**.

```
type intacc is access all integer;
package strConvert is new System.Address_To_Access_Conversions(intacc);
...
string_address := strConvert.To_Address( SomeIntAccVar );
```

Ada 83: The original use of .**all** created too many ambiguities. Ada 95 requires greater use of .**all**.

In Ada 2005, access parameters can also be declared as "not null" (do not allow a null pointer).

11.10 Callbacks

A callback is a pointer to a subprogram. They are called callbacks because you usually give the pointer to another subprogram that calls the procedure pointed to whenever it needs to. You can declare callbacks using **type**.

```
type UpdateWindow is access procedure;
type DisplayNewValue is access procedure( newval : integer );
```

One important restriction is that Ada requires that callbacks to refer to global subprograms. This is done to ensure that the access variable always points to an existing subprogram. You cannot create a callback to a local procedure or function, even if it's perfectly safe to do so. If you try, you'll get an obscure error message about one level being deeper than another.

The gnat equivalent for **'unchecked_access** for callbacks is **'unrestricted_access**, which you can use if you're absolutely sure the subprogram you're using will not save the access when it's finished running.

You can get the address of a procedure using **'access**. Suppose MyUpdateProcedure is a procedure fitting the description of UpdateWindow, a procedure with no parameters.

```
updatePtr : UpdateWindow := MyUpdateProcedure'access;
procedure DoComplexSlowComputation( updatePtr);
```

To call a callback, use the dereference operator .**all**.

```
UpdatePtr.all;
```

In Ada 2005, callback parameters can be declared in a subprogram without having to create a named type.

11.10.1 Storage Pools

Unlike languages like C that have only one storage pool, Ada allows you to define your own storage pools. Authors of real-time applications, for example, can create a pool with a maximum size limit or a fixed access time.

Use a for clause to make an access type use a pool other than the default storage pool.

```
type AccountingRecPtr is access all AccountingRec;
for AccountingRecPtr'storage_pool use my_pool;
```

Gnat defines several storage pools besides the default storage pool. Perhaps the most useful is the **debug pool**. This storage pool, available in version of Gnat before 3.12, works the same as the default storage pool except that it performs run-time checks for several different pointer related problems. If a check fails, an exception is raised.

The following program illustrates the errors caught by debug pool access types.

```
with Ada.Text_IO, System.Pool_Local, System.Debug_Pools;
use Ada.Text_IO;

with unchecked_deallocation;

-- This is an example of using the GNAT-specific debug_pool
-- storage pool.

procedure debpools is
```

```
    type sales_record is record

        salesman_code : integer;
        sales_amount : float;

    end record;

      -- just a typical record

    type salesptr_normal is access all sales_record;

    --
    -- This is a normal access type. It is allocated
    -- in the default storage pool (aka "the heap").
    -- The default storage pool is called
    -- Unbounded_No_Reclaimed_Pool. That is, there's
    -- no size limit, and memory is not reclaimed by
    -- garbage collection.
    -- A debug pool
    ----------------------------------------

sales_debug_pool : System.Debug_Pools.Debug_Pool;

-- declare a new debug pool
--
-- Debug_Pool is a GNAT-specific pool.

type salesptr_debug is access all Sales_Record;

for salesptr_debug'storage_pool

    use Sales_Debug_Pool;

-- This access type has no garbage collection
-- but raises exceptions on allocation or
-- deallocation errors, useful for tracking down
-- storage leaks. All 4 possible exceptions are
-- shown in this program.

procedure Free is new Unchecked_Deallocation( sales_record,
    salesptr_debug );
-- procedure to deallocate salesptr_debug access types

    sr : aliased Sales_Record;
    spd, spd2, spd3 : salesptr_debug;

begin

    Put_Line( "Fun with debug storage pools!");
    New_Line;

    -- Debug Pool Exception #1

    begin

        Put_Line( "Accessing a non-allocated access type is an exception:" );
        Put_Line( "spd.salesman_code := 1");

        spd.salesman_code := 1; -- error: not allocated
```

```
   exception when System.Debug_Pools.Accessing_Not_Allocated_Storage =>
        Put_Line( "***Accessing_Not_Allocated_Storage raised" );
   when others =>
        Put_Line( "***Unexpected exception" ); raise;
   end;

   New_Line;
   -- Debug Pool Exception #2

   begin
      Put_Line( "Freeing a non-allocated access type is an exception:" );
      Put_Line( "spd2 := sr'access --not allocated in pool" );
      Put_Line( "Free( spd2 )" );
      spd2 := sr'access;
      Free( spd2 );
   exception when
      System.Debug_Pools.Freeing_Not_Allocated_Storage =>
      Put_Line( "***Freeing_Not_Allocated_Storage raised" );
   when others =>
      Put_Line( "***Unexpected exception" ); raise;
   end;
   New_Line;

   spd := new Sales_Record'( salesman_code => 1, sales_amount => 55.50 );
   -- Debug Pool Exception #3

   begin
      Put_Line( "Accessing deallocated access type is an exception:" );
      Put_Line( "spd := new Sales_Record...");
      Put_Line( "Free( spd )" );
      Put_Line( "spd.salesman_code := 1");
      Free( spd );
      spd.salesman_code := 1; -- error: not allocated
   exception when System.Debug_Pools.Accessing_Not_Allocated_Storage =>
      Put_Line( "***Accessing_Deallocated_Storage raised" );
   when others =>
      Put_Line( "***Unexpected exception" ); raise;
   end;
   New_Line;

   spd := new Sales_Record'( salesman_code => 1, sales_amount => 55.50 );
   -- Debug Pool Exception #4

begin
     Put_Line( "Freeing deallocated access type is an exception:" );
     Put_Line( "spd := new Sales_Record...");
     Put_Line( "spd2 := spd" );
     Put_Line( "Free( spd )" );
     Put_Line( "Free( spd2 )" );
     spd2 := spd;
     Free( spd );
     Free( spd2 );
exception when System.Debug_Pools.Freeing_Deallocated_Storage =>

   Put_Line( "***Freeing_Deallocated_Storage raised" );
when others =>
   Put_Line( "***Unexpected exception" ); raise;
end;
New_Line;
```

```
end debpools;
```

Program Result:

```
Fun with debug storage pools!

Accessing a non-allocated access type is an exception:
spd.salesman_code := 1
***Accessing_Not_Allocated_Storage raised

Freeing a non-allocated access type is an exception:
spd2 := sr'access --not allocated in pool
Free( spd2 )
***Freeing_Not_Allocated_Storage raised

Accessing deallocated access type is an exception:
spd := new Sales_Record...
Free( spd )
spd.salesman_code := 1
***Accessing_Deallocated_Storage raised

Freeing deallocated access type is an exception:
spd := new Sales_Record...
spd2 := spd
Free( spd )
Free( spd2 )
***Freeing_Deallocated_Storage raised
```

To create your own storage pools, you need to extend the Root_Storage_Pool tagged record found in the System.Storage_Pools package.

[give example--KB]

11.10.2 Access Parameters

Because pointers are offen passed as parameters, Ada provides a special parameter type just for access types. **access** parameters are access types behave the same as an **in** parameter: you cannot assign a new value to the parameter. However, because it is an access type, you can change what the access parameter points to.

Access parameters offer some advantages over in parameters with an access type:

- Ada will verify that the parameter isn't null
- Access parameters can be used in functions where in out parameters are not allowed
- They avoid access type accessibility errors (without resorting to 'unchecked_access)

Access parameters can't be compared or assigned, but you can typecast an access parameter into a normal access type and then compare values or assign it.

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure accparm is
   -- An example of access parameters

   -- Create a customer account record
```

```
    type money is new float;
    type aPercent is new float;

    type aCustomerAccount is record
            moneyOwing : money := 0.0;      -- money on the account
            interest   : aPercent := 0.15;  -- 15% interest
    end record;
    type aCustomerPtr is access all aCustomerAccount;


    procedure chargeInterest( cp : access aCustomerAccount ) is
       -- update the customer record by charging the interest
    begin
       cp.moneyOwing := cp.moneyOwing * money(1.0 + cp.interest );
    end chargeInterest;

    procedure chargeInterest2( c : in out aCustomerAccount ) is
       -- update the customer record by charging the interest
    begin
       c.moneyOwing := c.moneyOwing * money(1.0 + c.interest );
    end chargeInterest2;

    function chargeInterest3( cp : access aCustomerAccount ) return boolean is
       -- update the customer record by charging the interest
       -- if under 1000, don't charge interest and return false
    begin
       if cp.moneyOwing < 1000.0 then
            return false;
       end if;
       cp.moneyOwing := cp.moneyOwing * money(1.0 + cp.interest );
       return true;
    end chargeInterest3;

      cp : aCustomerPtr;

begin

      Put_Line( "An Example of Access Parameters" );
      New_Line;

      cp := new aCustomerAccount;

      Put_Line( "chargeInterest uses an access parameter" );
      cp.moneyOwing := 1500.0;
      Put_Line( "Charging interest on" & cp.moneyOwing'img );
      chargeInterest( cp );
      Put_Line( "After interest, money owing is" & cp.moneyOwing'img );
      New_Line;

      Put_Line( "chargeInterest2 uses an in out parameter" );
      cp.moneyOwing := 1700.0;
      Put_Line( "Charging interest on" & cp.moneyOwing'img );
      chargeInterest2( cp.all );
      Put_Line( "After interest, money owing is" & cp.moneyOwing'img );
      New_Line;

      Put_Line( "chargeInterest3 is a function with an access parameter" );
      cp.moneyOwing := 1900.0;
      Put_Line( "Charging interest on" & cp.moneyOwing'img );
      if chargeInterest3( cp ) then
```

```
        Put_Line( "After interest, money owing is" & cp.moneyOwing'img );
    else
        Put_Line( "No interest was charged" );
    end if;
    New_Line;

    Put_Line( "A null pointer for an access parameter causes an exception" );
    cp := null;
    Put_Line( "Charging interest on a null pointer" );
    if chargeInterest3( cp ) then
        Put_Line( "After interest, money owing is" & cp.moneyOwing'img );
    else
        Put_Line( "No interest was charged" );
    end if;
    New_Line;

exception
    when constraint_error =>
            Put_Line( Standard_Error, "Constraint error exception raised" );
    when others =>
            Put_Line( Standard_Error, "Unexpected exception raised" );

end accparm;
```

An Example of Access Parameters

```
chargeInterest uses an access parameter
Charging interest on 1.50000E+03
After interest, money owing is 1.72500E+03

chargeInterest2 uses an in out parameter
Charging interest on 1.70000E+03
After interest, money owing is 1.95500E+03

chargeInterest3 is a function with an access parameter
Charging interest on 1.90000E+03
After interest, money owing is 2.18500E+03

A null pointer for an access parameter causes an exception
Charging interest on a null pointer
Constraint error exception raised
```

In Ada 2005, access parameters can also be declared as "not null" (do not allow a null pointer) or "constant" (pointer to a constant).

11.11 Multithreading

Versions of Gnat prior to GCC 3.x come with two alternative libraries for multithreading support. It can either use the native Linux (for example, starting with pthreads for libc6, now standard in Linux), or it can use FSU (Florida State University) threads that are included with Gnat. By default, the Linux version of gnat is compiled for Linux native threads.

Rumour has it one of the side effects is how the threads are implemented on multiple CPU machines. FSU threads will only use one processor for all threads. Native threading will use multiple processors.

GCC 3.x supports native threads only.

 The number of threads you can create depends on the operating system and the size of the stack. For example, on Windows XP there is a limit of 252 threads. Increase the stack size if you need more threads.

11.11.1 FSU verses Native Threads

The FSU threads provide better concurrency at very small time slices, but are incompatible with Linux's pthreads library. This means you can't use the FSU version of gnat with the standard Linux libraries unless you recompile the libraries for FSU threads as well. FSU threads also force blocking on system calls and can cause blocking problems multiprocessors, and as a result most people don't use them. One exception is Florist, a POSIX (that is, Linux O/S calls) binding using FSU threads. The main benefit of FSU threads is that they are Ada Annex C & D compliant.

Linux kernel 2.6 introduced better threads so it may be that FSU threads are no longer necessary. I haven't investigated this.

To use FSU threads, you need to compile gnat 3.x from its sources.

11.11.2 Tasks

In Ada, a thread is referred to ask a **task**. It has nothing to do with multitasking, as its name might imply. A task runs independently of the main program, either by true parallelism on a multiprocessor computer, or as a separate job on a single processor computer. There is no way to specify which processor will receive a task: Linux takes care of this automatically.

 Multithreaded programs have limits on the stack size for each thread--this is true for all Linux computer languages. Gnat 3.13 has an 8 Meg stack size limit per thread. Older versions had limits as low as 1 Meg per thread because of limits imposed by the Linuxthreads library.

A task can take on several forms. In its simplest form, a task is structured like a package, with a specification and a body. The following is an example of a simple thread that waits 5 seconds after a program is started and displays a message.

```
task SimpleTask;
task body SimpleTask is
begin
   delay 5.0;
   Put_Line( "The simple task is finished");
end SimpleTask;
```

The specification, like a package, indicates what identifiers are available to the outside world. The SimpleTask thread doesn't communicate with the main program: it's specification is only one line long.

Communicating with the rest of the program can be difficult. For example, with the tasks and main program running in parallel, sharing variables can be difficult. How does one task know when another task or the main program is finished changing the value of a variable? If a task works with a variable that's only partially been udpated, the data will be corrupt.

Ada provides two ways for a thread to communicate with the rest of the program.

The first communication method is called a **rendezvous**. One task communicates with another by sending a request. A task may send a request to another task to update a certain variable, or to perform a certain action, that would otherwise risk data corruption.

Because this communication happens "on the fly", it's declared in two parts. First, in the task specification, a list of all requests the task is prepared to accept. These are called entry statements and look much like procedure declarations.

Suppose we write a task to keep a running total, to be shared between several other tasks. We use a separate task to keep the total from being corrupted.

```
task CountingTask is
   entry add( amount : integer );
end CountingTask;
```

In the task body, a task indicates when it's ready to accept messages using the accept statement. This statement checks for outstanding requests from the rest of the program.

```
task body CountingTask is
   runningTotal : integer := 0;
begin
   loop
      accept add( amount : integer ) do
      runningTotal := runningTotal + amount;
      end add;
   end loop;
end CountingTask;
```

When this thread runs, accept statement will check for an add request. If there are no outstanding add requests, the thread suspends itself until a request is sent, waiting indefinitely for a new request. Suspending for a request is known as blocking.

An accept statement with do part will cause your task to wait for that request and then do nothing. You can do this to synchronize two tasks.

```
accept WaitUntilITellYouToGo;
```

Suppose we add another entry statement to read the current value of the running total.

```
task CountingTask is
   entry add( amount : integer );
   entry currentTotal( total : out integer);
end CountingTask;
```

In this case, we want the task to check for two different requests. The Ada select statement keeps a task from blocking and instead checks for multiple messages.

```
task body CountingTask is
   runningTotal : integer := 0;
begin
   loop
      select
         accept add( amount : integer ) do
            runningTotal := runningTotal + amount;
         end add;
      or
         accept currentTotal( total : out integer ) do
            total := runningTotal;
         end currentTotal;
      end select;
```

```
    end loop;
end CountingTask;
```

In this example, the task will repeatedly check for an add request or a currentTotal request.

To communicate with the task, we make calls to the task as if it were a package. For example, to send a message to add 5 to the running total, we'd use

```
CountingTask.Add( 5);
```

Because accept is a statement that executes at run-time, you can create any kind of message policy you want. Some messages can block. Some messages can be checked. You can force certain message to be handled before others.

The final "or" part of a select can contain instructions to execute when none of the accepts statements are executed. For example, a task can end itself with the terminate command. If you want a task to terminate when there are no more requests, add a

```
or
    terminate
```

at the end of your select statement.

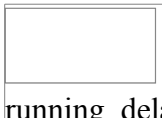
If select statement doesn't give you enough control over blocking, select can include **when** clauses. The clauses work like an if statement, executing the accept statement only if a condition is true. If time2accept is a boolean variable, you could write

```
select
    when time2accept =>
        accept add( amount : integer ) do
```

A when clause is referred to as a "guard" because, like a crossing guard, the accept will not execute unless the guard gives permission.

Ada also has a delay statement. Delay forces a task (or a program) to suspend itself for a number of seconds. To wait for three-and-a-half seconds, use

```
delay 3.5;
```

 If you change the operating system clock backwards while an Ada application is running, delay will wait until the point of time originally calcuated when delay was executed. That is, it waits until the system clock returns to where it was before and then waits 3.5 seconds.

You can place a delay statement in the last "or" part of a select statement to force the task to suspend itself for a few seconds if no requests were found.

Delay can also wait for a particular time. The time is expressed using the Ada.Calendar time format. If Tomorrow is a variable with the time of midnight tomorrow in the Ada.Calendar time format, you can delay the start of a task with

```
delay until Tomorrow;
```

In an emergency, one task can terminate another with the abort statement.

```
abort CounterTask;
```

Ada provides a variation of the select statement for tasks that timeout when they cannot complete their work in time. This version has two parts: the first part consists of what to do if the task isn't

completed in time, and the second consists of the task to complete. For example, if BigCalculation is a slow process that we wish to timeout after 30 seconds,

```
select delay 30.0;
   Put_Line( "Timeout!" );
then abort
   BigCalculation;
end select;
```

In this example, BigCalculation will continue for up to 30 seconds. If it doesn't finish, "Timeout!" is displayed and BigCalculation is aborted.

11.11.3 Task Types

Often multithreaded programs will need a set of identical tasks. For example, you many want to sort a customer's records for several different customers using different threads. You can't do this with the simple tasking examples shown so far.

To create a set of identical tasks, you must create a template of the tasks to run. Ada calls these templates a **task type**: they look just like a regular task except the specification begins with "task type" instead of "task" by itself. The following is a task template using the CountingTask example.

```
task type CountingTask is
   entry add( amount : integer );
   entry currentTotal( total : out integer );
end CountingTask;
```

This template will not run by itself. If we want to create a CountingTask task, we create one by declaring it.

```
Task1, Task2 : CountingTask;
```

Task1 and Task2 are two copies of CountingTask. This is equivalent to creating two separate tasks:

```
task Task1 is
   entry add( amount : integer );
   entry currentTotal( total : out integer);
end Task2;

task Task2 is
   entry add( amount : integer );
   entry currentTotal( total : out integer);
end Task2;
```

Because task types can be declared, we can create 20 tasks at once by declaring an array of 20 CountingTask's.

```
CountingTasks : array(1..20) of CountingTask;
```

Tasks can also be declared in records as well, or allocated dynamically using the new operator.

```
type aTaskPtr is access CountingTask;
tp : aTaskPtr;
...
tp := new CountingTask;
```

Using new, you can create as many copies of a particular task as you need.

11.11.4 Protected Items/Types

Ada tasks are useful for many kinds of multithreading. However, Ada provides a second method of multithreading called protected objects. These are similar to the "monitors" used by Java.

In an Ada task, you specify when and how different tasks communicate. When items are declared protected, Ada controls the interaction for you.

Protected objects are declared the same was as a package, using a specification and a body. They act like a package that allows only one task access to its contents at a time. While one task is using the contents, any other task wanting access is blocked until the first task is finished.

Here is our CountingTask rewritten as a protected item.

```
protected CountingType is
   procedure add( amount : integer);
   procedure currentTotal( total : out integer );
private
   runningTotal : integer := 0;
end CountingType;

protected body CountingType is
   procedure add( amount : integer ) is
   begin
      runningTotal := runningTotal + amount;
   end add;

   procedure currentTotal( total : out integer ) is
   begin
        total := runningTotal;
   end currentTotal;

end CountingType;
```

In this case, any task may execute the add or currentTotal procedures, but only one task may execute them at a time. This ensures that runningTotal will not be corrupted.

Unlike a package, you can't declare variables in the spec or body unless you put them in a "private" part in the spec.

Protected items can include entry declarations. Since there is no "main program" to the protected body, the protected body contains no accept statements. Instead, the entry declarations are filled in as if they were a subprogram. Any guarding conditions are attached to the end of the entry header.

For example, to create an a version of our add procedure that blocks if the total is higher than 100, we could write

```
   protected CountingType is
      entry add( amount : integer );
...
   protected body CountingType is
      entry add( amount : integer ) when runningTotal<100 is
   begin
      runningTotal := runningTotal + amount;
   end add;
...
```

Like tasks, you can create **protected types** by declaring the specification with "protected type" instead of "protected". You can then create arrays of protected items, or declare protected items dynamically.

This covers the basics of Ada's multithreading capabilities. There's much more that Ada can do. If you are writing complicated multithreading programs, you're encouraged to investigate the Ada Reference Manual for more information.

11.12 Ada Text Streams

A stream is a sequential transmission of different types of data. When data is written to a stream, the stream converts the data to a form suitable for transmission. When the data is read, it's converted from the stream's format back to its original form.

A practical example is saving tagged records belonging to the same class to a file.

Ada's syntax for using streams is a bit cumbersome. Like tagged records, some of the features are implemented using attributes, and others are found in standard packages.

The **Ada.Streams.Stream_IO** can write heterogeneous data to a text file, and then read it back again. This package contains a number of subprograms similar to Ada.Text_IO, including, Open, Close, Reset, Delete, Is_Open and End_Of_File.

However, there are no Get or Put procedures. Instead, there are stream subprograms for working with the data in the file.

- **Stream** - returns a stream to the file
- **Read** - read data from the stream
- **Write** - write data to the stream

Read and Write are not used directly. Instead, the attributes 'read and 'write will read and write an item to the stream. That is, Gnat does the necessary conversion to stream data for you.

For classes of tagged records, 'input and 'output read and write any child of the class. These attributes are implicitly defined for all class-wide types (that are not also limited). As a result, you usually combine 'class with the stream attributes for 'class'input and 'class'output. If you don't supply 'class, nothing will be written and no exception will be raised.

```
with ada.text_io, ada.streams.stream_io;
use  ada.text_io, ada.streams.stream_io;
with ada.unchecked_deallocation;

procedure class_stream is

   -- Contact_Info: A simple class
   -- the base class must not be abstract when using 'input and 'output

   type contact_info is tagged null record;
   type contact_ptr is access all contact_info'class;

   type phone_number is new contact_info with record
         phone : string(1..14);
   end record;

   type office_number is new contact_info with record
         office : integer;
   end record;

   procedure free is new ada.unchecked_deallocation(
```

```
        contact_info'class, contact_ptr);

    -- A Stream File and Its Stream

    stream_file : Ada.Streams.Stream_IO.File_Type;
    the_stream  : Ada.Streams.Stream_IO.Stream_Access;

    contact : contact_ptr;

begin

    Put_Line( "An example of Ada.Streams.Stream_IO and a Class" );
    Put_Line( "-----------------------------------------------" );
    New_Line;

    Create( stream_file, out_file, "contact_list.stream" );
    Put_Line( "Created a stream file" );
    -- open the stream file

    the_stream := stream( stream_file );
    -- get a stream representing the file's contents

    contact := new phone_number'( phone => "1-905-555-1023" );
    contact_info'class'output( the_stream, contact.all );
    free( contact );
    Put_Line( "Wrote a phone number" );
    -- write a record

    contact := new office_number'( office => 8023 );
    contact_info'class'output( the_stream, contact.all );
    free( contact );
    Put_Line( "Wrote an office number" );
    -- write a record

    Close( stream_file );
    New_Line;
    -- close the stream file

    -- Read Them

    Open( stream_file, in_file, "contact_list.stream" );
    Put_Line( "Opened a stream file" );
    -- open the stream file

    the_stream := stream( stream_file );
    -- get a stream representing the file's contents

    while not End_of_File( stream_file ) loop
        declare
            contact : contact_info'class := contact_info'class'input( the_stream );
        begin
            -- if this were more sophisticated, we could write a Put procedure
            -- for each tagged record and use dynamic dispatching
            if contact in phone_number then
                Put_Line( "read a phone number" );
            elsif contact in office_number then
                Put_Line( "read an office number" );
            else
                Put_Line( "read something else" );
            end if;
```

```
        end;
    end loop;

    Close( stream_file );
    -- close the stream file

end class_stream;
```

An example of Ada.Streams.Stream_IO and a Class

```
Created a stream file
Wrote a phone number
Wrote an office number

Opened a stream file
read a phone number
read an office number
```

Files of items of the same time are more easily created with Ada.Sequential_IO or Ada.Direct_IO.

Custom streams can be created to save or trasmit data in other ways such as in memory or through a network connection. Custom streams are created as tagged records extended from a root class, **Ada.Streams.Root_Stream_Type'class**.

```
type My_Stream is new Root_Stream_Type with record
...
```

Your stream type must override the abstract Red and Write subprograms to add and remove data from the stream.

The following is an in-memory stream creating by Warren Gay. This stream can share data between programs, buffering the data as text in memory. If a buffer overflow occurs, an END_ERROR is raised.

```
-- $Id: memory_stream.ads,v 1.1 2000/11/26 05:00:18 wwg Exp $
-- (c) Warren W. Gay VE3WWG ve3wwg@home.com, ve3wwg@yahoo.com
--
-- Protected under the GNU GPL License

with Ada.Finalization, Ada.Streams;
use  Ada.Finalization, Ada.Streams;

package Memory_Stream is

        type Stream_Access is access all Ada.Streams.Root_Stream_Type'Class;
        type Memory_Buffer(Max_Elem: Stream_Element_Offset) is new Controlled with private;

        ------------------------------------------------
        -- The new Stream Type, which must be derived from
        -- Root_Stream_Type. Note that Root_Stream_Type is
        -- NOT derived from Controlled, so if
        -- controlled attributes are necessary, they must
        -- be defined separately, and embedded into this
        -- object, as is done with Memory_Buffer here.
        ------------------------------------------------
        type Memory_Buffer_Stream(Max_Elem: Stream_Element_Offset) is new Root_Stream_Type with record
                        Mem_Buf:        Memory_Buffer(Max_Elem);   -- Object with Finalization
        end record;
```

```
        type Memory_Buffer_Stream_Ptr is access all Memory_Buffer_Stream;

        -- The overloaded abstract for Read
        procedure Read(Stream: in out Memory_Buffer_Stream;
                Item: out Stream_Element_Array; Last: out Stream_Element_Offset);

        -- The overloaded abstract for Write
        procedure Write(Stream: in out Memory_Buffer_Stream;
                Item: in Stream_Element_Array);

        -- Rewind the Read Memory Buffer Index
        procedure Rewind_Read(Stream: Stream_Access);

        -- Rewind the Write Memory Buffer Index
        procedure Rewind_Write(Stream: Stream_Access);

        -- To permit easy destruction of this stream
        procedure Free(Stream: Stream_Access);

private

        ------------------------------------------------
        -- To create a Memory_Buffer stream with an
        -- Initialize procedure, it must be derived from
        -- a Controlled type. Unfortunately, the type
        -- Root_Stream_Type is not derived from the
        -- Controlled type, so it is done privately here.
        ------------------------------------------------
        type Memory_Buffer(Max_Elem: Stream_Element_Offset) is new Controlled with record
                        Read_Offset:    Stream_Element_Offset;
                        Write_Offset:   Stream_Element_Offset;
                        Buffer:         Stream_Element_Array(1..Max_Elem);
        end record;

        procedure Initialize(Buf: in out Memory_Buffer);
        procedure Write(Buf: in out Memory_Buffer; Item: in Stream_Element_Array);
        procedure Read(Buf: in out Memory_Buffer;
                Item: out Stream_Element_Array;
                Last: out Stream_Element_Offset);
        procedure Rewind_Read(Buf: in out Memory_Buffer);
        procedure Rewind_Write(Buf: in out Memory_Buffer);

end Memory_Stream;
```

```
-- $Id: memory_stream.adb,v 1.1 2000/11/26 05:00:18 wwg Exp $
-- (c) Warren W. Gay VE3WWG ve3wwg@home.com, ve3wwg@yahoo.com
--
-- Protected under the GNU GPL License

with Ada.Text_IO; use Ada.Text_IO;

with Ada.Finalization; use Ada.Finalization;
with Ada.IO_Exceptions; use Ada.IO_Exceptions;
with Ada.Unchecked_Deallocation;

package body Memory_Stream is
```

```
	-------------------------------------------------
	-- Read from a Memory Buffer Stream :
	-------------------------------------------------
	procedure Read(Stream: in out Memory_Buffer_Stream; Item: out Stream_Element_Array; Last: out
Stream_Element_Offset) is
	begin
		Read(Stream.Mem_Buf,Item,Last);
	end Read;

	-------------------------------------------------
	-- Write to a Memory Buffer Stream :
	-------------------------------------------------
	procedure Write(Stream: in out Memory_Buffer_Stream; Item: in Stream_Element_Array) is
	begin
		Write(Stream.Mem_Buf,Item);
	end Write;

	-------------------------------------------------
	-- Rewind the Read Memory Buffer Index
	-------------------------------------------------
	procedure Rewind_Read(Stream: Stream_Access) is
		Mem_Str:    Memory_Buffer_Stream_Ptr := Memory_Buffer_Stream_Ptr(Stream);
	begin
		Rewind_Read(Mem_Str.Mem_Buf);
	end Rewind_Read;

	-------------------------------------------------
	-- Rewind the Write Memory Buffer Index
	-------------------------------------------------
	procedure Rewind_Write(Stream: Stream_Access) is
		Mem_Str:    Memory_Buffer_Stream_Ptr := Memory_Buffer_Stream_Ptr(Stream);
	begin
		Rewind_Write(Mem_Str.Mem_Buf);
	end Rewind_Write;

	-------------------------------------------------
	-- Free a Memory Buffer Stream :
	-------------------------------------------------
	procedure Free(Stream: Stream_Access) is
		type Memory_Buffer_Stream_Ptr is access all Memory_Buffer_Stream;
		procedure Free_Stream is new
Ada.Unchecked_Deallocation(Memory_Buffer_Stream,Memory_Buffer_Stream_Ptr);
		Str_Ptr:    Memory_Buffer_Stream_Ptr := Memory_Buffer_Stream_Ptr(Stream);
	begin
		Free_Stream(Str_Ptr);
	end Free;

	-------------------------------------------------
	-- Private Implementation :
	-------------------------------------------------

	-------------------------------------------------
	-- Initialize a Memory_Buffer Object :
	-------------------------------------------------
	procedure Initialize(Buf: in out Memory_Buffer) is
	begin
		Buf.Read_Offset := Buf.Buffer'First;
		Buf.Write_Offset := Buf.Buffer'First;
```

```
        end Initialize;

        ------------------------------------------------
        -- Write to a Memory Buffer Object :
        ------------------------------------------------
        procedure Write(Buf: in out Memory_Buffer; Item: Stream_Element_Array) is
                Count:      Stream_Element_Offset := Item'Last + 1 - Item'First;
                Last:       Stream_Element_Offset := Buf.Write_Offset + Count - 1;
        begin

                if Last > Buf.Buffer'Last then
                        raise Ada.IO_Exceptions.End_Error;
                end if;

                Buf.Buffer(Buf.Write_Offset..Last) := Item;
                Buf.Write_Offset := Buf.Write_Offset + Count;
        end Write;

        ------------------------------------------------
        -- Read from a Memory Buffer Object :
        ------------------------------------------------
        procedure Read(Buf: in out Memory_Buffer; Item: out Stream_Element_Array; Last: out
Stream_Element_Offset) is
                Xfer_Count: Stream_Element_Offset := Item'Last + 1 - Item'First;
                Data_Count: Stream_Element_Offset := Buf.Write_Offset - Buf.Read_Offset;
        begin

                if Xfer_Count > Data_Count then
                        Xfer_Count := Data_Count;
                end if;

                Item(1..Xfer_Count) := Buf.Buffer(Buf.Read_Offset..Buf.Read_Offset+Xfer_Count-1);
                Buf.Read_Offset := Buf.Read_Offset + Xfer_Count;
                Last := Item'First + Xfer_Count - 1;
        end Read;

        ------------------------------------------------
        -- Rewind the Read offset in the Memory Buffer
        ------------------------------------------------
        procedure Rewind_Read(Buf: in out Memory_Buffer) is
        begin
                Buf.Read_Offset := Buf.Buffer'First;
        end Rewind_Read;

        ------------------------------------------------
        -- Rewind the Write offset in the Memory Buffer
        ------------------------------------------------
        procedure Rewind_Write(Buf: in out Memory_Buffer) is
        begin
                Buf.Read_Offset := Buf.Buffer'First;    -- Implies a Read offset rewind
                Buf.Write_Offset := Buf.Buffer'First;   -- Rewind the write offset
        end Rewind_Write;
end Memory_Stream;
```

```
-- $Id: main.adb,v 1.1 2000/11/26 05:00:18 wwg Exp $
-- (c) Warren W. Gay VE3WWG ve3wwg@home.com, ve3wwg@yahoo.com
--
```

```
-- Protected under the GNU GPL License

with Ada.Text_IO;
use Ada.Text_IO;

with Memory_Stream;
use Memory_Stream;

-------------------------------------------------
-- This is a demo main program, that makes use of
-- our home-brewed Memory_Buffer_Stream.
--
-- To demonstrate, a record of type my_rec is
-- written to the stream with known values, and
-- then is read back twice, into records T and U.
--
-- Then the write offset is rewound, and a new
-- float variable F is written, and then read
-- back into float variable G.
-------------------------------------------------
procedure Main is
	type my_rec is record		-- A demonstration record
			A:	natural;
			B:	integer;
			S:	string(1..8);
			Z:	float;
	end record;

	Str:	Stream_Access := null;		-- A Stream
	R:	my_rec := ( 23, -95, "oink    ", 1.414 );   -- An initialized record

	T:	my_rec := ( 0, 0, "        ", 0.0 );	-- For 1st read
	U:	my_rec := T;				-- For 2nd read

	F:	float := 29.99;				-- An initialized float
	G:	float := 0.0;				-- For 3rd read

begin

	put_line("Demonstration has begun:");

	Str := new Memory_Buffer_Stream(4096); -- Create in-memory buffer stream (4096 bytes)

	my_rec'write(Str,R);			-- Write record R to stream

	my_rec'read(Str,T);			-- Read stream back to record T

	put_line("T.A :=" & natural'image(T.A));    -- Dump out T
	put_line("T.B :=" & integer'image(T.B));
	put_line("T.S := '" & T.S & "'");
	put_line("T.Z := " & float'image(T.Z));

	Rewind_Read(Str);			-- Rewind the read pointer

	my_rec'read(Str,U);			-- Now read into record U

	put_line("U.A :=" & natural'image(U.A));    -- Dump out U
	put_line("U.B :=" & integer'image(U.B));
	put_line("U.S := '" & U.S & "'");
	put_line("U.Z := " & float'image(U.Z));
```

```
        Rewind_Write(Str);                  -- Implies a read rewind also

        float'write(Str,F);                 -- Write F to stream

        float'read(Str,G);                  -- Read stream into G

        put_line("G :=" & float'image(G));      -- Report G for verification

        Free(Str);                          -- Delete stream

        put_line("Demonstration complete.");
end Main;
```

```
Demonstration has begun:
T.A := 23
T.B :=-95
T.S := 'oink    '
T.Z :=  1.41400E+00
U.A := 23
U.B :=-95
U.S := 'oink    '
U.Z :=  1.41400E+00
G := 2.99900E+01
Demonstration complete.
```


http://www.ibpaus.de/downloads/index.html has an example of streams that always use network byte order.

11.13 Pragmas

Pragmas, sometimes called compiler directives, are statements that provide additional information to the compiler build a better executable. Pragmas never change the meaning of a program.

The following are the predefined Ada 95 pragmas:

Abort_Defer	defer abouts over a block of statements*
Ada_83	enforce Ada 83 conventions, even if compiler switches say otherwise*
Ada_95	enforce Ada 95 conventions, even if compiler switches say otherwise*
All_Calls_Remote	All subprograms in a RPC package spec are RPC callable
Annotate	add information for external tools*
Assert	create an assertion*
Asynchronous	call to remote subprogram can complete before subprogram is done
Atomic	identifier must be read/written without interruption
Attach_Handler	install a signal handler procedure
C_Pass_By_Copy	when calling C functions, use pass by copy (not by reference) when able*
Comment	same as Ident *
Common_Object	for Fortran, create variables that share storage space*

Complex_Representation	use gcc's complex number format (for speed)*
Component_Alignment	indicate how record components should be stored*
Controlled	turn off garbage collection for a type (no effect in gnat)
Convention	apply a convention to an identifier
CPP_Class	treat a record or tagged record as a C++ class*
CPP_Constructor	treat imported function as a C++ class constructor*
CPP_Destructor	treat imported function as a C++ class destructor*
CPP_Virtual	import a C++ virtual function*
CPP_Vtable	specify a virtual function table*
Debug	specify a debugging procedure call*
Discard_Names	discard ASCII representation of identifiers, as used by 'img
Elaborate	elaborate a certain package before this one
Elaborate_All	elaborate all **with**'ed packages before this one
Elaborate_Body	elaborate a package's body immediate after it's spec
Elaboration_Checks	select dynamic (Ada Reference Manual) or static (Gnat default) elaboration checks
Eliminate	indicate an identifier that is not used in a program, created by gnatelim *
Error_Monitoring	treat errors as warnings during a compile*
Export	export an identifier from your program so it can be used by other languages
Export_Function	export an Ada function with additional information over pragma Export*
Export_Object	export an Ada tagged record with additional information over pragma Export*
Export_Procedure	export an Ada procedure with additional information over pragma Export*
Export_Valued_Procedure	export an Ada side effect function with additional information over pragma Export*
Extend_System	obsolete*
External_Name_Casing	Set the default capitalization for a case-sensitive langauge imports. May be set to Uppercase, Lowercase, or As_Is (Gnat default)*.
Finalize_Storage_Only	no finalize on library-level objects, primarily for gnat's internal use *
Ident	object file identification string (no effect with Linux) *
Import	import an identifer from another language so it can be used in your program
Import_Function	import a non-Ada function with additional information over pragma Import*
Import_Object	import a non-Ada object with additional information over pragma Import*
Import_Procedure	import a non-Ada procedure with additional information over pragma Import*
Import_Valued_Proce	import a non-Ada side effect function with additional information over

dure	pragma Import*
Initialize_Scalars	set scalars variables to illegal values whenever possible
Inline	the indicated subprogram may be inlined.
Inline_Always	forces inter-unit inlining, regardless of compiler switches*
Inline_Generic	for compatibility with other Ada compilers*
Inspection_Point	specify that an identifier's value must readable at the given point in the program (for code validation)
Interface_Name	for compatibility with other Ada compilers*
Interrupt_Handler	declare a signal handler procedure
Interrupt_Priority	Specify the task/protected object's priority where blocking occurs
Linker_Alias	select an alternative linker[?] for a package (or other llinkable unit)*
Linker_Options	pass a string of options to the linker
Linker_Section	the gcc linker section to use *
List	list source code while being compiled
Locking_Policy	Spcify how protected objects are locked and when blocking occurs
Machine_Attribute	specify GCC machine attributes*
No_Return	specify a procedure that is deliberately never returned from, to avoid compiler warnings*
No_Run_Time	ensures no gnat run-time routines are use (e.g. for creating device drivers)*
Normalize_Scalars	set scalars variables to illegal values whenever possible (Usually initialize_scalars should be used instead)
Optimize	indicates how statements should be optimzed
Pack	indicates that the type should be compressed as much as possible
Page	start a new page in a program listing
Passive	for compatibility with other Ada compilers*
Polling	if on, enables exception polling*
Priority	Specify the priority of a task
Preelaborate	preelaborate the specified package
Propagate_Exceptions	specify imported subprogram that can handle Ada exceptions; used with zero cost handling *
Psect_Object	Same as common_object*
Pure	specifies the package is pure
Pure_Function	specify a function without side-effects*
Queuing_Policy	How task/protected objects are sorted when queued
Ravenscar	enforce the Ravenscar real-time policies*
Remote_Call_Interface	Ensure a package can be callable by remote procedure calls
Remote_Types	used for communication between RPC partitions
Restricted_Run_Time	like Ravenscar, turns on a number of restrictions for real-time

	programming *
Restrictions	Disable certain language features
Reviewable	Provide a run-time profiling (like gprof)
Share_Generic	for compatibility with other Ada compilers*
Shared_Passive	used for sharing global data with separate RPC partitions
Source_File_Name	overrides normal Gnat file naming conventions*
Source_Reference	for use with gnatchop*
Storage_Size	amount of storage space for a task
Stream_Convert	simplified way of creating streams I/O subprograms for a given type*
Style_Checks	Select which Gnat style source code style checks to enforce (if enabled when compiled)*
Subtitle	for compatibility with other Ada compilers*
Suppress	turn off specific checks for common exceptions
Suppress_All	for compatibility with other Ada compilers*
Suppress_Initialization	disable initialization of variables of a given type*
Task_Dispatching	specify how tasks sort dispatches (e.g. FIFO_Within_Priorities)
Task_Info	specify information about a task*
Task_Storage	specify the guard area for a task*
Time_Slice	specify tasking time slice for main program [in Linux?]*
Title	for compatibility with other Ada compilers*
Unchecked_Union	treat a record as a C union type*
Unimplemented_Unit	for unfinished units, produces a compiler error if they are compiled*
Unreserve_All_Interrupts	allow reassigning of signals normally handled by gnat, eg. SIGINT*
Unsuppress	opposite of suppress*
Use_VADS_Size	for older Ada code, 'size is equivalent to 'vads_size *
Validity_Checks	activites the equivalent gnatmake switch
Volatile	value of variable may change unexpectedly
Volatile_Components	array components may change unexpectedly
Warnings	turn compiler warnings on or off*
Weak_External	specify an identifier that doesn't have to be resolved by the linker *

* - GNAT specific

The use of these pragmas are covered in detail in the GNAT and Ada 95 reference manuals.

11.14 Low-Level Ada

Ada	Description	C Equivalent

`pragma volatile`	Variable is influnced outside of program	`volatile` declaration
`for x'address use a`	Specify an absolute address for a pointer	`p = (typecast) integer;`
`for x'alignment use b`	Position the identifier x on b byte boundaries	?
`for x'bit_order use o;`	Store record using bit order o	?
`8#value#`	Specify a octal numeric literal	0Value
`16#value#`	Specify a hexadecimal numeric literal	0xValue
`asm( inst, in, out)`	Assemble an instruction	`asm( inst : in : out )`

Ada contains a number of low-level features and libraries, such as the ability to add machine code to a program or to access a hardware register in memory. I'll mention a few of these features here.

If you need to specify a specific number of bits, say to access a hardware register, use the **for** statement.

```
type Reg is new integer;
for Reg'size use 4; -- register is 4 bits
type RegPtr is access all Reg;
```

You can refer an access type to a particular address using for **var'address** clause. Together with **'size** , you can theoretically refer to an bit in the computer's memory. However, on Linux the address refers to a location in your address space and doesn't refer to a real physical location.

var'alignment will align storage to a particular byte boundary. **var'bit_order** can specify a different bit order for a record than the default for your machine. The bit orders are defined in the System package. (Gnat 3.12 doesn't fully support bit orders.)

```
for Reg'address use 16#2EF#; -- hex address 2EF in your memory space
for Reg'alignment use 2; -- align to 16-bit boundaries
for myRecord'bit_order use system.low_order_first; -- low bits to high machine
```

If the register value can change independently of the Ada program (usually true), we need to mark the pointer with pragma volatile to make sure the compiler will make no assumptions while optimizing.

```
pragma volatile( RegPtr);
```

Ada will do bit-wise logic operations, but it will only do them on packed arrays of booleans or modular types. You can't do bit-wise operations on integers, for example. The bit-wise operators are **and**, **or**, **xor** and **not**.

```
type byte is array(1..8) of boolean;
pragma pack( byte );

  b1, b2, b3 : byte;
  ...
  b3 := b1 and b2;
```

If you need octal or hexadecimal numbers, Ada denotes these by using a leading base designation between 2 and 16, with the value delimited by number signs.

```
Hex := 16#0FE9#;
```

```
Bin := 2#01101010#;
```

The **System.Machine_Code** package can embed assembly language instructions in an Ada program. Since Gnat is based on Gcc, it uses Gcc's inlining features. If you've inlined assembly code in a C program, you'll find Gnat's capabilities virtually identical.

Assembly code is embedded using the **Asm**.

```
Asm( "nop" ); -- do nothing
```

A tutorial by Jerry van Dijk is available from AdaPower.com.

Large sections of assembly code should be linked in separately.

12 Standard Gnat Packages

This section summarizes some of the more than 100 packages that come with the Gnat compiler. These include string handling, operating system binding, and sorts.

12.1 Standard String and Character Packages

Ada Package	*Description*	*C Equivalent*
Ada.Characters.Handling	Character operations	Strings.h?
Ada.Strings.Fixed	Ada string operations	Strings.h
Ada.Strings.Bounded	Bounded strings and operations	-
Ada.Strings.Unbounded	Unbounded strings and operations	-
Gnat.Case_Util	Just case conversions	-
Ada.Strings.Unbounded.Text_IO	Text_IO for Unbounded Strings	-

Ada's built-in strings, or "fixed" strings, are made of array of characters. The length of the array determines the bounds of the string. A string that's too short for an array is padded with blanks. Although these strings are fast, they are cumbersome to use and not practical for string-intensive applications. One problems is, although the string type is an array with an undefined upper bound, sooner or later you have to specify an upper bound and run the risk of constraint errors working with arrays of different sizes.
Ada operator "&" concatenates fixed strings: this is the only built-in operator for fixed strings.
There are two alternative strings in Ada. **Bounded strings** are arrays of strings with a definite maximum size, separate from the length, which eliminates to constraint errors. These strings are still relatively fast, but waste a lot of storage on small strings and you run the risk of overflowing the string. I use 255 character bounded strings as general purpose strings in my programs.
The standard Ada library **Ada.Strings.Bounded** contains the definition of bounded strings and similar operations to Ada.Strings.Fixed. Because bounded strings have a definite upper bound, the package is generic and has to be instantiated for the maximum length. The library also includes a function to convert a bounded string to a fixed string.

Unbounded strings are strings that can be of any size. They are typically implemented by dynamic allocation, which makes them slow, but they don't waste memory the way bounded strings do and there's no risk over a string overflow. The standard Ada library **Ada.Strings.Unbounded** contains the definition of unbounded strings and operations on them, including a function to convert an unbounded string to a fixed string.

C: Unbounded strings are not exactly the same as C strings. For one thing, unbounded strings don't end in null characters. C String support is in the packages Interfaces.C.

```
with Ada.Text_IO, Ada.Strings.Unbounded.Text_IO;
use Ada.Text_IO, Ada.Strings.Unbounded, Ada.Strings.Unbounded.Text_IO;
procedure unbio is
-- this program demonstrates basic input/output with
-- unbounded strings. These routines are more efficient
```

```
-- because they avoid conversion into standard Ada
-- strings
us : Unbounded_String;
begin
    Put_Line( "This program displays information on the screen" );
    Put_Line( "and reads information from the keyboard" );
    New_Line;
    Put_Line( "Type in a string" );
    us := Get_Line;
    New_Line;
    Put_Line( "Put_Line displays a line of text and advances to" );
    Put_Line( "the next line." );
    Put( "The string you typed was " );
    Put_Line( us );
    New_Line;
end unbio;
```

```
This program displays information on the screen
and reads information from the keyboard
Type in a string
Uptown Girl, she been looking for a downtown man...
Put_Line displays a line of text and advances to
the next line.
The string you typed was Uptown Girl, she been looking for a downtown man...
```

For characters, the standard Ada library **Ada.Characters.Handling** provides basic operations such as conversions between case, tests for types of characters, and conversions two and from 16-bit wide characters.

```
    Text_IO.Put_Line( Ada.Characters.Handling.To_Upper( 'r' ) );
```

This example prints 'R' on the screen.

For string handling capabilities, you need to use a package. The standard Ada library **Ada.Strings.Fixed** contains operations for fixed strings, including extracting substrings, mapping characters from one set to another (for example, upper to lower case), and string searching. There is also an**Ada.Strings.Unbounded** package containing the same subprograms for unbounded strings, and likewise an **Ada.Strings.Bounded** for bounded strings.

Figure: Standard String Subprograms

Set_Bounded_String / Set_Unbounded_String — (Ada 2005) assign a fixed string to another type of string
Append / **&** — concatenate one string to another
Element — return the character at a particular index
Replace_Element — replace a character at a particular index
Slice — return a substring (In Ada 2005, also bounded_slice and unbounded_slice)
Replace_Slice / **Overwrite** — replace a substring
Insert — add a string in the midst of the original string
Delete — remove a string in the midst of the original string
Count — return the number of occurrences of a substring
Index — locate a string in the original string
Index_Non_Blank — locate the first non-blank character

Head — return the first character(s) of a string
Tail — return the last character(s) of a string
Trim — remove leading or trailing spaces
***** - duplicate the string a specific number of times
Tokenize —
Translate — convert a string to a new set of characters using a mapping function
The following program demonstrates many of the standard Ada string subprograms using unbounded strings.

```
with Ada.Text_IO, Ada.Strings.Unbounded.Text_IO;
use Ada.Text_IO, Ada.Strings.Unbounded,
    Ada.Strings.Unbounded.Text_IO;
procedure strdemo is
-- demonstrate some of the Ada strings subprograms
    teststr : string := "The rich get richer";
    us : Unbounded_String;
begin
    Put_Line( "This program shows some Ada string capabilities" );
    New_Line;
    Put( "Our test string is " );
    Put_Line( teststr );
    New_Line;
    Put_Line( "To_Unbounded_String converts a string to an unbounded string" );
    us := To_Unbounded_String( teststr );
    Put_Line( us );
    New_Line;
    Put_Line( "The length of the string is " & length( us )'img );
    Put_Line( "If we append, ' but not happier', the string is" );
    Append( us, " but not happier" );
    Put_Line( us );
    New_Line;
    Put_Line( "The ampersand will work as well: " & us );
    New_Line;
    Put_Line( "The fifth character is " & Element( us, 5 ) );
    New_Line;
    Put_Line( "Replacing the 20th character, we get" );
    Replace_Element( us, 20, ',' );
    Put_Line( us );
    New_Line;
    Put_Line( "The 5th to 8th charcaters is " & Slice( us, 5, 8 ) );
    New_Line;
    Put_Line( "The first occurence of 'ch' is at " &
        Index( us, "ch" )'img );
    New_Line;
    Put_Line( "The first non-blank character is at " &
        Index_Non_Blank( us )'img );
    New_Line;
    Put_Line( "Replacing the first 'rich' with 'RICH' we get" );
    Replace_Slice( us, 5, 8, "RICH" );
    Put_Line( us );
    New_Line;
```

```
    Put_Line( "Inserting 'really ' at the 5th character, we get" );
    Insert( us, 5, "really " );
    Put_Line( us );
    New_Line;
    Put_Line( "Overwriting characters 5 to 8, we get" );
    Overwrite( us, 5, "most" );
    Put_Line( us );
    New_Line;
    Put_Line( "Deleting characters 5 through 11, we get" );
    Delete( us, 5, 11 );
    Put_Line( us );
    New_Line;
    Put_Line( "The first 8 characters at the head of the string are" );
    Put_Line( Head( us, 8 ) );
    New_Line;
    Put_Line( "The last 8 characters at the tail of the string are" );
    Put_Line( Tail( us, 8 ) );
    New_Line;
    -- Count is ambiguous because of the use clauses
    Put_Line( "The count of 'er' is " &
        Ada.Strings.Unbounded.Count( us, "er" )'img );
    New_Line;


end strdemo;
```

```
This program shows some Ada string capabilities
Our test string is The rich get richer
To_Unbounded_String converts a string to an unbounded string

The rich get richer

The length of the string is 19

If we append, ' but not happier', the string is

The rich get richer but not happier

The ampersand will work as well: The rich get richer but not happier

The fifth character is r

Replacing the 20th character, we get

The rich get richer,but not happier
The 5th to 8th charcaters is rich
The first occurence of 'ch' is at 7
The first non-blank character is at 1

Replacing the first 'rich' with 'RICH' we get

The RICH get richer,but not happier
Inserting 'really ' at the 5th character, we get
The really RICH get richer,but not happier
Overwriting characters 5 to 8, we get
The mostly RICH get richer,but not happier
```

Deleting characters 5 through 11, we get
The RICH get richer,but not happier
The first 8 characters at the head of the string are
The RICH
The last 8 characters at the tail of the string are
happier
The count of 'er' is 2
There are also a number of libraries dealing with wide strings, strings with 16-bit characters.
If you are only interested in doing case conversions, gnat provides a small package
called **case_util** that does case conversions (and only case conversions) on characters and strings.
Use case_util to avoid loading the entire Ada.Characters.Handling library.

The following sample program demonstrates the uses of case_util:

```
with text_io, gnat.case_util;

use text_io;
procedure casetest is
   teststr : constant string := "This is a TEST_string";
   tempstr : string := ".....";
begin
   Put_Line( "This is an example of the Gnat string case conversion tools:" );
   New_Line;
   Put_Line( "The original string is '" & teststr & "'" );
   New_Line;
   TempStr := TestStr;
   Gnat.Case_Util.To_Upper( TempStr );
   Put_Line( "Upper case is '" & TempStr & "'" );
   tempstr := teststr;
   Gnat.Case_Util.To_Lower( TempStr );
   Put_Line( "Lower case is '" & TempStr & "'" );
   tempstr := teststr;
   Gnat.Case_Util.To_Mixed( TempStr );
   Put_Line( "Mixed case is '" & TempStr & "'" );
end casetest;
```

```
This is an example of the Gnat string case conversion tools:
The original string is 'This is a TEST_string'
Upper case is 'THIS IS A TEST_STRING'
Lower case is 'this is a test_string'
Mixed case is 'This is a test_String'
```

Ada defines a number of character sets. ASCII is the standard ASCII character set.
To put an "æ" character on displays that support the Latin character set, use

```
Put( Ada.Characters.Latin_1.LC_AE_Dipthong );
```

12.1.1 String Performance

The choice of string type to use depends on functionality and performance. Here's a chart showing a series of tests using different string types. The tests were run on my Pentium II 350 MHz.

Fixed String (2K) Bounded String Unbounded

			(2K)
Character Append	0.01 s	72.8 s	16.0 s
Concat.Two Strings	14.9 s	22.5 s	76.4 s
Equality	0.18 s	0.09	0.08 s
			0.43 s 0.24 s
Conversion to		70.0 s	12.4 s
Fixed	-	71.8 s	13.6 s
Insert/Delete	11.8 s	30.6 s	115.2 s
		15.2 s	
Replace Character	0.3 s	0.7 s	0.9 s
Determining Length	0.3 s	0.4 s	0.6 s
Duplication	3.1 s	1.4 s	2.1 s

This is a very informal benchmark and is intended only to demonstrate the kinds of problems associated with different strings. The actual performance will depend on the length of the strings, the optimization in the packages and nature of the string operations. This case used 2K strings (where applicable) and each test was performed 1 million times.

- Character concatenation (2000 chars x 500 = 1 million characters): s(n) := 'x'; n := n + 1; or s := s & 'x';
- String comparision (1 million 2K strings): b := s = s1; (s1 = null or s1 = full)
- String Conversion (1 million conversions):(s1 = 2000 'x's or s1 = 2000 spaces)
- Insert or Delete One Character (1 million changes): s := insert(delete(s, 1000, 1000), 100, "x"); or s := delete(s, 1000, 1000); s := insert(s, 1000, "x"); (in a declare block for fixed strings)
- Replace Character (2000 x 500 iterations): s(1000) := 'x'; or replace_element function 1 million 2 + 2000 character strings

Fixed strings are almost always faster, but they often involve writing work-arounds due to the possibility of constraint errors. Unbounded strings have no constraints but almost always the slower than either fixed or bounded strings. Bounded strings offer a balance between convenience and performance and are the fastest for copying values.

12.2 Advanced Input/Output

12.2.1 GNAT.IO

For small programs that don't need the full capabilities of Text_IO, GNAT provides a package called GNAT.IO. This package can get and put integers, characters and strings. Unlike Text_IO, it's also preelaborated.

```
with GNAT.IO;
use GNAT.IO;
procedure giodemo is
-- this program demonstrates basic input/output using the
-- GNAT.IO package, a stripped down version of Text_IO
   c : character; -- this is a letter
begin
   Put_Line( "This program displays information on the screen" );
   Put_Line( "and reads information from the keyboard" );
   New_Line;
   Put_Line( "Put_Line displays a line of text and advances to" );
   Put_Line( "the next line." );
```

```
    Put( "Put " );
    Put_Line( "displays text, but it doesn't start a new line" );
    Put_Line( "New_Line displays a blank line" );
    New_Line;
    Put_Line( "Get waits for a character to be typed." );
    Put_Line( "Type a key and the Enter key to continue." );
    Get( c );
    Put_Line( "The character you typed was '" & c & "'" );
end giodemo;
```

```
This program displays information on the screen
and reads information from the keyboard
Put_Line displays a line of text and advances to
the next line.
Put displays text, but it doesn't start a new line
New_Line displays a blank line
Get waits for a character to be typed.
Type a key and the Enter key to continue.
g
The character you typed was 'g'
```

[Not complete]

These packages are only useful for simple programs. Usually you will rely on packages/libraries provided for your project.

Text_IO file operations are very limited and are only intended for quick and dirty programs. There are other libraries for more extensive file operations, such as**Ada.Sequential_IO** and **Ada.Direct_IO**.

There is also a subpackage for displaying formatted text, such as columns of numbers.

12.2.2 IO_Aux

GNAT's IO_Aux package provides three commonly used functions to Text_IO programs: testing for a file's existence, and reading an unlimited length strings from a text file or a console.

```
with Ada.Text_IO, GNAT.IO_Aux;
use Ada.Text_IO, GNAT.IO_Aux;
procedure ioaux is
-- this program demonstrates the features of the IO_Aux
-- package
    TestFile : string := "/etc/passwd";
    procedure ScanString( s : string ) is
    begin
        Put_Line( "The string you typed was " & s );
        Put_Line( "It is" & s'length'img & " characters long" );
    end ScanString;
begin
    Put_Line( "This program demonstrates the features of the" );
    Put_Line( "IO_Aux package. This package adds three functions" );
    Put_Line( "to simple Text_IO programs." );
    New_Line;
    Put_Line( "File_Exists tests for a file's existence." );
```

```
    if File_Exists( TestFile ) then
        Put_Line( TestFile & " exists" );
    else
        Put_Line( TestFile & " doesn't exist" );
    end if;
    New_Line;
    Put_Line( "Get_Line is the same as Ada.Text_IO's Get_Line" );
    Put_Line( "except that reads a string of unlimited length" );
    Put_Line( "and doesn't return an explicit length value." );
    New_Line;
    Put_Line( "Please type in a string of any length" );
    ScanString( GNAT.IO_Aux.Get_Line );
    New_Line;
    Put_Line( "The third function is a version of Get_Line" );
    Put_Line( "that reads any string from a Text_IO files." );
    New_Line;
end ioaux;
```

```
This program demonstrates the features of the
IO_Aux package. This package adds three functions
to simple Text_IO programs.
File_Exists tests for a file's existence.
/etc/passwd exists
Get_Line is the same as Ada.Text_IO's Get_Line
except that reads a string of unlimited length
and doesn't return an explicit length value.
Please type in a string of any length
Mary had a little lamb
The string you typed was Mary had a little lamb
It is 22 characters long
The third function is a version of Get_Line
that reads any string from a Text_IO files.
```

12.3 Sequential_IO

A sequential file is a list of similar items saved on a disk (or other long-term storage media). They are similar to a one dimensional array except there is no upper bound, and each item must be processed in sequence (hence the name "sequential"). You can create sequential files of same-length strings, or integer, but most commonly records are used.

You can **open** an existing sequential IO file, or you can **create** a new one. When you open or create a file, you have to indicate what file mode you'll be using. "In" mode files can only be read. "Out" mode files can only be written to. "Append" is like out mode except that records are added to the end of an existing file.

The **reset** procedure changes to a new mode and repositions your program accordingly to the end or beginning of the file.

When you are finished with a sequential file, you can either **close** it or **delete** it if you don't need it again.

Because there is no way of knowing how many records are remaining in the file, there is a function called **End_of_File** that you can check after each read to see if the last item has been read. You can only use End_of_File in In mode--it makes no sense to use it in Out or Append modes since you always write at the end of the file.

The following program writes a couple of customer records to a sequential file and reads them back again:

```
with Ada.Text_IO, Ada.Sequential_IO, Ada.IO_Exceptions;
use  Ada.Text_IO;

procedure sequentio is
   -- Ada.Sequential_IO example

   type aCustomer is record
            name       : string(1..40);
            amountOwing : float := 0.0;
   end record;
   -- a customer record with two fields

   package aCustomerFile is new Ada.Sequential_IO( aCustomer );
   use aCustomerFile;
   -- instantiate a new package for sequential IO on a file of
   -- customer records

   CustomerFile : aCustomerFile.File_Type;
   -- our customer file
   -- use "aCustomerFile" because Text_IO and Sequential_IO have File_Type

   cr : aCustomer;

begin

   Put_Line( "This is a Ada.Sequential_IO example" );
   New_Line;

   -- create the file

   Create( CustomerFile,
                Mode => Out_File,
                Name => "customer.seq" );

   -- display some statistics

   Put_Line( "We created the file " & Name( CustomerFile ) );
   Put_Line( "We're currently using " & Mode( CustomerFile )'img & " mode" );
   if Is_Open( CustomerFile ) then
        Put_Line( "The file is open" );
   else
        Put_Line( "The file isn't open" );
   end if;
   New_Line;

   -- write the first record

   cr.name := "Tokyo Book Distributors                ";
   Write( CustomerFile, cr );
```

```
    Put_Line( "Writing " & cr.name );

    -- write another record

    cr.name := "General Pizza Inc.                    ";
    Write( CustomerFile, cr );
    Put_Line( "Writing " & cr.name );
    Put_Line( "End_of_File not allowed on Out files" );
    begin
    if End_Of_File( CustomerFile ) then
          Put_Line( "We are at the end of the file" );
    else
          Put_Line( "We aren't at the end of the file" );
    end if;
    exception when Ada.IO_Exceptions.Mode_Error =>
       Put_Line( Standard_Error, "End_of_File caused Ada.IO.Exceptions.Mode_Error" );
    when others =>
       Put_Line( Standard_Error, "Unexpected exception occurred" );
    end;
    New_Line;

    -- change modes using Reset

    Put_Line( "Reset can change the file mode" );
    Put_Line( "Changing to In_File mode" );
    Reset( CustomerFile, In_File );

    -- read first record

    Put_Line( "Reading the next customer" );
    Read( CustomerFile, cr );
    Put_Line( "Read " & cr.name );
    New_Line;

    -- read second record

    Put_Line( "Reading the next customer" );
    Read( CustomerFile, cr );
    Put_Line( "Read " & cr.name );
    New_Line;

    -- check the end of the file

    Put_Line( "End_of_File works on In files" );
    if End_Of_File( CustomerFile ) then
          Put_Line( "We are at the end of the file" );
    else
          Put_Line( "We aren't at the end of the file" );
    end if;
    New_Line;

    Put_Line( "Closing file" );
    Close( CustomerFile );

end sequentio;
```

This is a Ada.Sequential_IO example

```
We created the file /home/ken/ada/trials/customer.seq
We're currently using OUT_FILE mode
The file is open

Writing Tokyo Book Distributors
Writing General Pizza Inc.
End_of_File not allowed on Out files
End_of_File caused Ada.IO.Exceptions.Mode_Error

Reset can change the file mode
Changing to In_File mode
Reading the next customer
Read Tokyo Book Distributors

Reading the next customer
Read General Pizza Inc.

End_of_File works on In files
We are at the end of the file

Closing file
```

Sequential_IO does not have 64-bit "large file" support. If you are going to create very large files, you will have to create your own bindings to your operating system.

[Form not covered--KB]

12.4 Direct_IO

In relational database programming, you create tables of information. The tables act like arrays that are limited in length by the amount of disk space you have. Each table consists of a series of rows, and each row is divided up into subcategories called columns.

A telephone book, for example, can be considered one large table. Each row contains information about a different person. Each row is subdivided into columns of names, addresses and phone numbers.

Although you could represent a database table using a sequential IO file, it would be very difficult to use. To look up the 1000th entry in the file, you would have to read through the first 999 entries.

The Ada equivalent to a database table is called a **direct IO file**. Some languages refer to this kind of file as a "random access" file. A direct IO file is called "direct" because you can move directly to any row in the file without having to read any other rows.

The rows in a direct IO file are typically represented by records (athough they can be any data of a known length) and the columns are the fields in the records. Direct IO files can also use variant records--Ada will ensure there is enough space in each entry for the largest variation.

You can **open** an existing direct IO file, or you can **create** a new one. When you open or create a file, you have to indicate what file mode you'll be using. "In" mode files can only be read. "Out" mode files can only be rewritten. Unlike sequential IO files, there is also an "In Out" mode which allows you to both read and write records. This is the most common mode for accessing direct IO files.

If you move to a position beyond the end of the file, such as trying to write to row 100 when there are only 50 rows, the other unused rows will be created and filled with zero bytes--ASCII.NUL in characters or strings, 0 in integers and long_integers, and so forth. The only way to shorten a direct IO file is to create a new one, delete the original and copy the new one in place of the original.

There are several useful functions for direct IO files:

- Is_Open is true if the file has been opened
- End_Of_File is true if you have read the last record in the file. (This is unavailable in out mode.)
- Name is the path of the file
- Mode is the current file mode
- Size is the number of rows in the file
- Index is the number of the current row

The following example program reads and writes customer information using the Ada.Direct_IO package.

```
with Ada.Text_IO, Ada.Direct_IO, Ada.IO_Exceptions;
use Ada.Text_IO;

procedure dirio is
   -- Ada.Direct_IO example

   type aCustomer is record
            name       : string(1..40);
            amountOwing : float;
   end record;
   -- a customer record with two fields

   package aCustomerFile is new Ada.Direct_IO( aCustomer );
   use aCustomerFile;
   -- instantiate a new package for direct IO on a file of
   -- customer records

   CustomerFile : aCustomerFile.File_Type;
   -- our customer file
   -- use "aCustomerFile" because Text_IO and Direct_IO have File_Type

   cr : aCustomer;

begin

   Put_Line( "This is a Ada.Direct_IO example" );
   New_Line;

   -- create the file

   Create( CustomerFile,
               Mode => Out_File,
               Name => "customer.dir" );

   -- display some statistics

   Put_Line( "We created the file " & Name( CustomerFile ) );
   Put_Line( "We're currently using " & Mode( CustomerFile )'img & " mode" );
   Put_Line( "There are" & Size( CustomerFile )'img & " records" );
   Put_Line( "We are on row " & Index( CustomerFile )'img );
   if Is_Open( CustomerFile ) then
      Put_Line( "The file is open" );
   else
      Put_Line( "The file isn't open" );
   end if;
   New_Line;

   -- write the first record
```

```
cr.name := "Midville Electric                    ";
Write( CustomerFile, cr );
Put_Line( "Writing " & cr.name );
Put_Line( "There are" & Size( CustomerFile )'img & " records" );
Put_Line( "We are on row " & Index( CustomerFile )'img );
New_Line;

-- write the next record on row 7

cr.name := "New York Distributors                ";
Write( CustomerFile, cr, To => 7 );
Put_Line( "Writing " & cr.name & " to row 7" );
Put_Line( "There are" & Size( CustomerFile )'img & " records" );
Put_Line( "We are on row " & Index( CustomerFile )'img );
Put_Line( "End_of_File not allowed on In files" );
begin
if End_Of_File( CustomerFile ) then
      Put_Line( "We are at the end of the file" );
else
      Put_Line( "We aren't at the end of the file" );
end if;
exception when Ada.IO_Exceptions.Mode_Error =>
    Put_Line( Standard_Error, "End_of_File caused Ada.IO_Exceptions.Mode_Error" );
when others =>
    Put_Line( Standard_Error, "Unexpected exception occurred" );
end;
New_Line;

-- change modes using Reset

Put_Line( "Reset can change the file mode" );
Put_Line( "Changing to InOut_File mode" );
Reset( CustomerFile, InOut_File );

-- read first record

Put_Line( "Reading the next customer" );
Read( CustomerFile, cr );
Put_Line( "Read " & cr.name );
New_Line;

-- read second (undefined record)

Put_Line( "Reading from row 2" );
Read( CustomerFile, cr );
Put_Line( "Read " & cr.name );
New_Line;

-- read 7th row

Put_Line( "Reading from row 7" );
Read( CustomerFile, cr, From => 7 );
Put_Line( "Read " & cr.name );
New_Line;

-- check the end of the file

Put_Line( "End_of_File works on InOut files" );
if End_Of_File( CustomerFile ) then
```

```
        Put_Line( "We are at the end of the file" );
    else
        Put_Line( "We aren't at the end of the file" );
    end if;
    New_Line;

    Put_Line( "Closing file" );
    Close( CustomerFile );

end dirio;
```

```
This is a Ada.Direct_IO example

We created the file /home/ada/customer.dir
We're currently using OUT_FILE mode
There are 0 records
We are on row  1
The file is open

Writing Midville Electric
There are 1 records
We are on row  2

Writing New York Distributors                to row 7
There are 7 records
We are on row  8
End_of_File not allowed on In files
End_of_File caused Ada.IO_Exceptions.Mode_Error

Reset can change the file mode
Changing to InOut_File mode
Reading the next customer
Read Midville Electric

Reading from row 2
Read

Reading from row 7
Read New York Distributors

End_of_File works on InOut files
We are at the end of the file

Closing file
```

Note: In this example, reading from the unassigned second record put a row of 40 ASCII.NUL characters on the screen. Because these are non-printable characters, nothing is visible in the results.

[What about objects? How are tags treated? --KB]

Direct_IO files are suitable for small database tables. If you need to work with large amounts of data, you should consider installing one of the free Linux databases (such as PostgreSQL or mySQL) and using them to store and retrieve your data. This is discussed in upcoming chapters.

Alternately, you can write your own database package using a the Linux kernel. seqio, a sequential IO package, is developed in chapter 16.

Direct_IO does not have 64-bit "large file" support. If you are going to create very large files, you will have to create your own bindings to your operating system.

12.5 Formatted Output

Formatted output refers to displaying a value based on a template showing, in general, how the output should look. Because the template is a string, it's easy to visualize the results. This idea is used in languages like COBOL and BASIC (with its PRINT USING command).

The Ada.Text_IO.Editing provides formatted output. The string template is called a **picture**. The picture can contain the following symbols.

- '+' - the number will be printed with a leading + or -
- '-' - a negative numbers will be printed with a leading -
- '<' and '>' - a negative number will be printed with (..)
- "CR" - a negative number will be printed with a leading "CR" (credit)
- "DB" - a negative number will be printed with a leading "DB" (debit)
- '$' - the currency symbol will be printed, or a floating dollar sign if multiple instances
- '.' - marks the actual position for a decimal point
- 'V' - marks the assumed position for a decimal point
- '9' - space for a number with leading zeros
- '#' - same as '$', except only the leading character is shown
- 'Z' - space for a numbers with leading blanks
- '_', 'B', '0', '/' - inserted. 'B' is a blank
- '*' - space for a number with leading asterisks

A PICTURE_ERROR is raised if there is a mistake in the layout. A LAYOUT_ERROR is raised if the layout can't be used with a particular value. Using a negative number without specifying a format symbol that allows negative numbers causes a LAYOUT_ERROR.

Before using Text_IO.Editing, the internal generaic package Decimal_Output must be instantiated for a particular numeric type. Only decimal types are allowed.

```
type money is delta 0.01 digits 18;
package formatted_io is new ada.text_io.editing.decimal_output( money );
```

To_Picture converts a string to a picture type. **Pic_String** returns the string of the picture type.

```
p : picture := To_Picture( "###9.99" );
s : string := Pic_String( p );
```

Valid returns true if a string is a valid picture. When **Blank_When_Zero** parameter is true, a zero represented as an empty string is allowed. By default, the picture string must show something for a zero. Blank_When_Zero can also be used with To_Picture.

```
if not Valid( "####9.99" ) then
    Put_Line( Standard_Error, "This is a bad picture string" );
end if;
```

Put displays the formatted decimal value. There is also an **Image** function that returns the results as a string instead of displaying it on the screen. **Length** returns the length of the formatted output. There is no Put_Line.

```
Put( 455.32, pic );
str := Image( 455.32, pic );
Put( str, 455.32, pic );
```

Here is an larger example:

```
with ada.text_io.editing;
use ada.text_io;
use ada.text_io.editing;

procedure formatted is
      type money is delta 0.001 digits 18;
      package formatted_io is new ada.text_io.editing.decimal_output( money );
      use formatted_io;

      procedure ShowValues( s : string ) is
      begin
         put( "  0.0  and " &s & " => " );
         put( 0.0, To_Picture( s ) );
         new_line;
         put( " 75.12 and " &s & " => " );
         put( 75.12, To_Picture( s ) );
         new_line;
         put( "-75.12 and " &s & " => " );
         begin
              put( -75.12, To_Picture( s ) );
         exception when others =>
              put( "LAYOUT_ERROR" );
         end;
         new_line;
      end ShowValues;

begin
    put_line( "This is an example of Formatted Output" );
    put_line( "---------------------------------------" );
    new_line;

    put_line( "Default currency symbol is " & Default_Currency  );
    put_line( "Default fill character is '" & Default_Fill & "'" );
    put_line( "Default separator character is '" & Default_Separator & "'" );
    put_line( "Default radix mark is '" & Default_Radix_Mark & "'" );
    new_line;

    ShowValues( "99999.99" );
    New_Line;

    ShowValues( "ZZZZ9.99" );
    New_Line;

    ShowValues( "****9.99" );
    New_Line;

    ShowValues( "-$$$9.99" );
    New_Line;

    ShowValues( "+###9.99" );
    New_Line;

    ShowValues( "<###9.99>" );
    New_Line;
end formatted;
```

```
This is an example of Formatted Output
---------------------------------------
```

```
Default currency symbol is $
Default fill character is ' '
Default separator character is ','
Default radix mark is '.'

    0.0  and ZZZZ9.99 =>     0.00
  75.12 and ZZZZ9.99 =>    75.12
-75.12 and ZZZZ9.99 => LAYOUT_ERROR

    0.0  and -9999.99 =>  0000.00
  75.12 and -9999.99 =>  0075.12
-75.12 and -9999.99 => -0075.12

    0.0  and ****9.99 => ****0.00
  75.12 and ****9.99 => ***75.12
-75.12 and ****9.99 => LAYOUT_ERROR

    0.0  and -$$$9.99 =>    $0.00
  75.12 and -$$$9.99 =>   $75.12
-75.12 and -$$$9.99 => - $75.12

    0.0  and +###9.99 => +  $0.00
  75.12 and +###9.99 => + $75.12
-75.12 and +###9.99 => - $75.12

    0.0  and <###9.99> =>    $0.00
  75.12 and <###9.99> =>   $75.12
-75.12 and <###9.99> => ( $75.12)
```

Put has many parameters used to override default values.

- **Currency** - the currency string to use
- **Fill** - the fill character to use
- **Separator** - the separator character to use
- **Radix_Mark** - the radix mark to use

There is also a Wide_Text_IO.Editing for wide string.

12.6 Calendar Package

Calendar is the standard Ada package for telling time. You can get the current time, compare time values, do time arithmetic and comparisons. There is also a GNAT.Calendar package which extends the Ada.Caledar package with days of the week, second duration, and other features.

Table: North American Federal Holidays and Celebrations

Work schedules (not including small retail stores) often affected by these holidays.

- New Year's Day, January 1st.
- Birthday of Martin Luther King (U.S.), third Monday in January.
- Inauguration Day (U.S.), January 20th every four years, starting in 1937.
- Washington's Birthday (U.S.), third Monday in Febrauary.
- Inauguration Day (U.S.), March 4th every four years, pre-1937.

- Good Friday and Easter Sunday (see below).
- Armed Forces Day (U.S.), third Saturday in May.
- Memorial Day (U.S.), last Monday in May.
- Flag Day (U.S.), June 14th.
- Canada Day (Canada), July 1.
- United States of America's Independence Day (U.S.), July 4.
- Labor Day, first Monday in September.
- Columbus Day, second Monday in October.
- Thanksgiving Day (Canada), second Monday in October.
- Election Day (U.S.), Tuesday on or after November 2.
- Veterans Day (U.S.), November 11th.
- Remembrance Day (Canada), November 11th.
- Thanksgiving Day (U.S.), fourth Thursday in November.
- Christmas Day, December 25th.

North American Banking (and postal) Holidays include Easter Monday and Victoria Day (Canada).

Daylight Savings Time

Daylight Savings time begins, first Sunday in April (but not in Arizona, Hawaii, and parts of southern Indiana).

Daylight Savings Time ends, last Sunday in October (but not in Arizona, Hawaii, and parts of southern Indiana).

Table: Other Widely Celebrated North American Observances

- Groundhog Day, February 2.
- Lincoln's Birthday (U.S.), February 12.
- Valentine's Day, February 14.
- Washington's Birthday (U.S.), February 22.
- St. Patrick's Day, March 17.
- April Fools's Day, April 1.
- Mothers' Day, second Sunday in May (36 USC Sec. 142).
- Victoria Day (Canada), second last Monday in May [KB?]
- Fathers' Day, third Sunday in June (36 USC Sec. 142a).
- St. Jean Baptiste Day (Quebec/Canada), last Saturday in June [KB?].
- Parents' Day, fourth Sunday in July (36 USC Sec. 142c).
- Grandparents' Day, Sunday after Labor Day (36 USC Sec. 142b).
- Columbus Day (U.S., traditional), October 12.
- United Nations Day (U.S.), October 24.
- Halloween, October 31.
- Boxing Day, December 26.

The following program demonstrates the basic operations of the calender package.

```
with text_io, calender;
use calender;

procedure caldemo is
   Year : Year_Number;
```

```
    Month : Month_Number;
    Day : Day_Number;
    Seconds : Day_Duration;
    Christmas94 : time;
begin
    Text_IO.Put_Line( "A simple calendar example" );
    Text_IO.New_Line;

    Split( Clock, Year, Month, Day, Seconds );
    Text_IO.Put_Line( "The current date is" &
        Year'img & "/" &
        Month'img & "/" &
        Day'img );
    Text_IO.Put_Line( "It's" & seconds'img &
        " seconds into the day" );
    Text_IO.New_Line;

    Christmas94 := Time_Of( 1994, 12, 25 );
    if Christmas94 < Clock then
        Text_IO.Put_Line( "It's after Christmas 1994" );
    else
        Text_IO.Put_Line( "It's before Christmas 1994" );
    end if;
    Text_IO.New_Line;

    Split( Clock+12.5, Year, Month, Day, Seconds );
    Text_IO.Put_Line( "In 12.5 seconds it will be " &
        Year'img & "/" &
        Month'img & "/" &
        Day'img );
    Text_IO.Put_Line( "And" & seconds'img &
        " seconds into the day" );
end caldemo;
```

```
A simple calendar example

The current date is 1998/ 12/ 17
It's 59775.185023000 seconds into the day

It's after Christmas 1994

In 12.5 seconds it will be 1998/ 12/ 17
And 59787.686581000 seconds into the day
```

The GNAT.Calendar.Time_IO package will write a time value according to a format string, similar to the Linux strftime function.

Easter is one of the hardest holidays to calculate. The following is a program to calculate the date of Easter Sunday:

[This should be rewritten for Ada.Calender -- KB]

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure easter is
```

```
procedure findEaster( year : integer; easter_month, easter_day : out integer ) is
   -- based on the public domain algorithm
   -- by Ed Bernal

   a,b,c,e,g,h,i,k,u,x,z : integer;

begin
         --
         --  "Gauss' famous algorithm (I don't know how or why it works,
         --  so there's no commenting)" -- Ed Bernal
         --

         a := year mod 19;
         b := year / 100;
         c := year rem 100;
         z := b / 4;
         e := b rem 4;
         g := (8*b + 13) / 25;
         h := (19*a + b - z - g + 15) rem 30;
         u := (a + 11*h) / 319;
         i := c / 4;
         k := c rem 4;
         x := (2*e + 2*i - k - h + u + 32) rem 7;
         easter_month := (h-u+x+90) / 25;
         easter_day := (h-u+x + easter_month +19) rem 32;
end findEaster;

   month, day : integer;

begin

   findEaster( 2000, month, day );
   Put( "Easter Sunday 2000 is month " & month'img );
   Put_Line( " and day " & day'img );

end easter;
```

```
Easter Sunday 2000 is month  4 and day  23
```

Ada 2005 introduces three additional packages: Ada.Calendar.Arithmetic, Ada.Calendar.Formatting and Ada.Calendar.Time_Zones.

12.7 Tags Package

Ada.Tags contains some utility procedures to for the invisible tags that accompany tagged records, including converting tags to and from strings. The following program shows what the tags package can do and shows tag comparison with the **in** operator.

```
with text_io, ada.tags;

procedure t is

   type ParentRec is tagged record

      i : integer;
```

```
    end record;
    type ChildRec is new ParentRec with record
       j : integer;
    end record;
    child : ChildRec;
begin
    Text_IO.Put_Line( "Working with Tagged Record Tags:" );
    Text_IO.New_Line;
    Text_IO.Put_Line( "ParentRec has an expanded name of " &
    Ada.Tags.Expanded_Name( ParentRec'Tag ) );
    Text_IO.Put_Line( "ChildRec has an expanded name of " &
    Ada.Tags.Expanded_Name( ChildRec'Tag ) );
    Text_IO.New_Line;
    Text_IO.Put_Line( "ParentRec has an external tag of " &
    Ada.Tags.External_Tag( ParentRec'Tag ) );
    Text_IO.Put_Line( "ChildRec has an external tag of " &
    Ada.Tags.External_Tag( ChildRec'Tag ) );
    Text_IO.New_Line;
    if child in ParentRec'class then
       Text_IO.Put_Line( "child (a child rec) is in ParentRec'class" );
    else
       Text_IO.Put_Line( "This should not happen" );
    end if;
    if child in ChildRec'class then
       Text_IO.Put_Line( "child (a child rec) is in ChildRec'class" );
    else
       Text_IO.Put_Line( "This should not happen" );
    end if;
end t;
```

```
Working with Tagged Record Tags:

ParentRec has an expanded name of T.PARENTREC
ChildRec has an expanded name of T.CHILDREC
ParentRec has an external tag of T.PARENTREC
ChildRec has an external tag of T.CHILDREC
child (a child rec) is in ParentRec'class
child (a child rec) is in ChildRec'class
```

12.8 Tables

Gnat 3.11 introduces **gnat.table**, a gnat package for creating an aribrary length array (or, for that matter, a link list).

[expand and give example program]

12.9 Hash Tables

Gnat provides a generic package for hash tables called **gnat.htable**. You provide gnat's package with information on the size of the tables, the elements it contains, and a hash function and the instantiation provides Get and Set procedures to put values in and take values out of your hash table.

Gnat 3.11: This version of gnat adds remove and iterator subprograms for hash tables.

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO; use Ada.Text_IO;
with GNAT.HTable;

procedure String_Hash is

   -- Items required by gnat.HTable
   subtype Hash_Index is Integer range 1 .. 100;
   subtype Hash_Element is Unbounded_String;
   Hash_Empty : constant Hash_Element := To_Unbounded_String("");

   subtype Hash_Key is Unbounded_String;
   function Hash_String is new GNAT.HTable.Hash(Hash_Index);
   function Hash_Function (Key : Hash_Key) return Hash_Index is
      Key_String : String(1 .. Length(Key)) := To_String(Key);
   begin
      return Hash_String(Key_String);
   end Hash_Function;

   -- A simple hash table using Unbounded_String keys and elements.
   package String_Table is new GNAT.HTable.Simple_HTable(
      Header_Num => Hash_Index,    -- table size
      Element    => Hash_Element,  -- table element type
      No_Element => Hash_Empty,    -- element when key not found
      Key        => Hash_Key,      -- table key type
      Hash       => Hash_Function, -- function to generate the hash
      Equal      => "=");          -- test for equal keys

   procedure Set(Key, Value : String) is
   begin
      String_Table.Set(To_Unbounded_String(Key), To_Unbounded_String(Value));
   end Set;

   procedure Get(Key : String) is
      K : Unbounded_String := To_Unbounded_String(Key);
   begin
      Put_Line(Key & " => " & To_String(String_Table.Get(K)));
   end Get;
```

```
begin
    New_Line;
    Put_Line("Starting string hash.");
    Put_Line("Storing state data in the hash table." );
    Put_Line("Empty positions are " & To_String(Hash_Empty));
    Set("Indiana", "Indiana, Indianapolis, <http://wikipedia.org/wiki/Indiana>");
    Set("Kentucky", "Kentucky, Frankfort, <http://wikipedia.org/wiki/Kentucky>");
    Set("Ohio", "Ohio, Columbus, <http://wikipedia.org/wiki/Ohio>");
    Get("Indiana");
    Get("Kentucky");
    Get("Ohio");
    Get("Magrathea");
end String_Hash;
```

```
Starting string hash.
Storing state data in the hash table.
Empty positions are <null>
Indiana => Indiana, Indianapolis, <http://wikipedia.org/wiki/Indiana>
Kentucky => Kentucky, Frankfort, <http://wikipedia.org/wiki/Kentucky>
Ohio => Ohio, Columbus, <http://wikipedia.org/wiki/Ohio>
Magrathea => <null>
```

12.10 Bubble and Heap Sorts

Gnat provides two packages for bubble sorting. Both assume that your information is in an array with a lower bound of zero. The zero element is used as temporary space for the sort.

The first, **gnat.bubble_sort_g**, is a generic. You provide the package with a procedure to move data in the array and a function to check for one value being less than another. The instantiation provides a sort procedure.

```
with text_io, gnat.bubble_sort_g;

use text_io;

procedure bubble1 is

-- Our table to sort

    type IntegerTable is array( 0..5 ) of integer;

    it : IntegerTable := ( 0, 13, 4, 5, 16, 8 );

    -- Define the items required by a generic gnat bubble sort

    procedure MoveIntegers( From, To : natural ) is

    begin
        it( To ) := it( From );
    end MoveIntegers;

    function CompareIntegers( left, right : natural ) return boolean is

    begin
        return it( left ) < it( right );
    end CompareIntegers;

    -- OK, instantiate the package
```

```
   --
   package IntSort is new gnat.bubble_sort_g(
      Move => MoveIntegers, -- how to move two things
      Lt => CompareIntegers ); -- how to compare to things
   procedure ShowTable is
   begin
      for i in IntegerTable'range loop
         Put_Line( i'img & " = " & it( i )'img );
      end loop;
   end ShowTable;
begin
   Put_Line( "This is an example of bubble sorting an integer table" );
   New_Line;
   Put_Line( "The table begins as:" );
   ShowTable;
   IntSort.Sort( it'last ); -- sort elements 1 to top of it array
   New_Line;
   Put_Line( "The sorted table is:" );
   ShowTable;
end bubble1;
```

```
This is an example of bubble sorting an integer table
The table begins as:

 0 = 0

 1 = 13
 2 = 4
 3 = 5
 4 = 16
 5 = 8

The sorted table is:

 0 = 13

 1 = 4
 2 = 5
 3 = 8
 4 = 13
 5 = 16
```

The second, **gnat.bubble_sort_a** uses callbacks instead of a generic package. Use this package if you want to conserve memory by avoiding a lot of instantiations of the generic bubble_sort_g. Remember that callbacks must be global, so we can't simple pass the local subprograms we created in bubble1. This time we must store the array and subprograms in a separate package.

```
with text_io, gnat.bubble_sort_a, inttable;
```

```
use text_io, inttable;
procedure bubble2 is
begin
    Put_Line( "This is an example of bubble sorting an integer table" );
    New_Line;
    Put_Line( "The table begins as:" );
    ShowTable;
    gnat.bubble_sort_a.Sort( n => it'last,
        Move => MoveIntegers'access,
        Lt => CompareIntegers'access );
    -- sort elements 1 to top of it array
    New_Line;
    Put_Line( "The sorted table is:" );
    ShowTable;
end bubble2;
package inttable is
-- Our table to sort
    type IntegerTable is array( 0..5 ) of integer;
    it : IntegerTable := ( 0, 13, 4, 5, 16, 8 );
    -- Define the items required by a callback gnat bubble sort
    -- these must be global to work
    procedure MoveIntegers( From, To : natural );
    -- move one item in the table from From position to To position
    function CompareIntegers( left, right : natural ) return boolean;
    -- compare two items in the table and determine if left is less than
    -- than right
procedure ShowTable;
end inttable;


with text_io;
use text_io;
package body inttable is
    procedure MoveIntegers( From, To : natural ) is
    begin
        it( To ) := it( From );
    end MoveIntegers;
    function CompareIntegers( left, right : natural ) return boolean is
    begin
        return it( left ) < it( right );
    end CompareIntegers;
```

```
    procedure ShowTable is
    begin
        for i in IntegerTable'range loop
            Put_Line( i'img & " = " & it( i )'img );
        end loop;
    end ShowTable;
end inttable;
```

This is an example of bubble sorting an integer table

The table begins as:

0 = 0

1 = 13
2 = 4
3 = 5
4 = 16
5 = 8

The sorted table is:

0 = 13

1 = 4
2 = 5
3 = 8
4 = 13
5 = 16

The heap sort package works identically, with both generic (**heap_sort_g)** and callback (**heap_sort_a**) versions as well. Heap sorts are better suited to large amounts of data. Here's the callback version using the same inttable package we used above.

```
with  text_io, gnat.heap_sort_a, p;

use text_io, p;

procedure heaptest is

begin
  Put_Line( "This is an example of heap sorting an integer table" );
  New_Line;
  Put_Line( "The table begins as:" );
  ShowTable;
  gnat.heap_sort_a.Sort( n => it'last,
    Move => MoveIntegers'access,
    Lt => CompareIntegers'access );
  -- sort elements 1 to top of it array
  New_Line;
  Put_Line( "The sorted table is:" );
  ShowTable;
end heaptest;
```

This is an example of heap sorting an integer table

The table begins as:

[this wasn't corrupted before—MS Word bug?]

12.11 Regular Expressions

"Regular Expressions" refers to pattern matching for strings: identifying all strings that adhere to a certain pattern. For example, listing all files that end with .ads using the shell command "ls *.ads" is an example of a regular expression.

GNAT has two built-in packages for dealing with regular expressions. The first, called "Regexp", performs pattern matching using two different standards. First, it supports standard UNIX shell "file globbing" expressions as described by "man bash". Second, it supports BNF patterns as described in the Ada Reference Manual.

Using the package is a two step process. First, you must compile the expression using the Compile function. Then, you check for a string that matches the expression using the Match function.

The following program demonstrates the Regexp package.

```
with Ada.Text_IO, GNAT.Regexp;

use Ada.Text_IO, GNAT.Regexp;

procedure regex is

procedure TestMatch( re : Regexp; s : string ) is

begin
   if Match( s, re ) then
      Put_Line( s & " matches the expression" );
   else
      Put_Line( s & " doesn't match the expression" );
   end if;
end TestMatch;

Criteria : Regexp;

begin

   Put_Line( "This program demonstrates GNAT's regular expression" );
   Put_Line( "capabilities. These are used to find text that match" );
   Put_Line( "a certain pattern." );
   New_Line;

   -- UNIX Regular Expressions

   Put_Line( "A 'globbing pattern' is a UNIX shell-style pattern matching" );

   Put_Line( "The pattern 'a*' matches anything starting with the letter 'a'" );
   Criteria := Compile( "a*", Glob => true, Case_Sensitive => true );
```

```
    New_Line;
    TestMatch( Criteria, "accounting" );
    TestMatch( Criteria, "President" );
    TestMatch( Criteria, "sundries" );
    New_Line;

    -- BNF Expressions

    Put_Line( "A non-globbing pattern is a BNF pattern, as used in the Ada" );

    Put_Line( "Reference Manual. For example, 'a[a-z]*' means characters" );
    Put_Line( "beginning with 'a' and with any number of letters following." );
    Criteria := Compile( "a[a-z]*", false, true );
    New_Line;
    TestMatch( Criteria, "accounting" );
    TestMatch( Criteria, "sales" );
    New_Line;
end regex;
```

```
This program demonstrates GNAT's regular expression
capabilities. These are used to find text that match
a certain pattern.

A 'globbing pattern' is a UNIX shell-style pattern matching

The pattern 'a*' matches anything starting with the letter 'a'

accounting matches the expression

President doesn't match the expression
sundries doesn't match the expression

A non-globbing pattern is a BNF pattern, as used in the Ada

Reference Manual. For example, 'a[a-z]*' means characters
beginning with 'a' and with any number of letters following.

accounting matches the expression

sales doesn't match the expression
```

The second Gnat pattern matching package is "Regpat" which interprets full UNIX V7 regular expressions as defined in the "man regexp" Linux man page. Don't be confused by the naming conventions: the Regexp package does not do Linux regular expressions.

12.12 Advanced String Processing

[spitbol-style string processing not finished]

12.13 GLADE Distributed Processing

GLADE is the free distributed processing package for TCP/IP and gnat. It is distributed separately from the gnat compiler. This should not be confused with the GTK's Glade, the GUI builder for the Gimp Toolkit widgets (http://glade.gnome.org), which has bindings for Ada (http://glade.pn.org/).

GLADE is built into the ALT version of GNAT.

To install GLADE, unpack it and type "configure" and "make install".

GLADE works on *partitions*, programs designed to run on other computers. Each partition has a *channel* between itself and another partition. Of course, the partitions can also run concurrently on one computer. You describe the partitions and channels using an Ada-like language called Garlic.

GLADE uses rsh to start partitions, so make sure you don't run the programs under the root login since root is not allowed to run programs via rsh.

[KB: I could install and compile programs with glade, but the communication wasn't working… error in my networking setup or did I not install it properly?]

12.14 Basic Math Packages

Type	*Ada Package*	*Description*
Generic	`Ada.Numerics.Generic_Elementary_Functions`	basic math for floating point numbers
`Short_Float`	`Ada.Numerics.Short_Elementary_Functions`	basic math for short_float type
`Float`	`Ada.Numerics.Elementary_Functions`	basic math for float type
`Long_Float`	`Ada.Numerics.Long_Elementary_Functions`	basic math for long_float type
`Long_Long_Float`	`Ada.Numerics.Long_Log_Elementary_Functions`	basic math for long_long_float type

Sooner or later, you will ask the question, "So, how do I compute the cosine of a number?" The answer found is the**Ada.Numerics.Generic_Elementary_Functions package**. This package with the unusually long name is the basic floating point math package. This is a generic package that you instantiate for a particular floating point type. For example, to set up the package for a custom floating point type called "percent",

```
with Ada.Numerics.Generic_Elementary_Functions;
type percent is new float range 0.0..1.0;
package percentMath is new Ada.Numerics.Generic_Elementary_Functions( percent );
use percentMath;
```

The "use percentMath" statement saves us from typing "percentMath." before every function we use.

With percentMath instantiated, we can now perform basic floating point math functions such as

```
Put_Line( "20% to the power of 3 is" & percent'image( 0.2**3.0 ) );
```

As shown in the table at the start of this section, elementary function packages for the basic floating point types are included with Gnat.

The elementary package includes:

Function	*Description*
`Sqrt( x )`	Square Root
`Log( x )`	Natural Logarithm (`ln` in some other languages)
`Log( x, b )`	Logarithm to base b
`Exp( x )`	Raise e by power x
`**`	Power operator
`Sin( x )`	Sine for x radians
`Sin( x, c )`	Sine for x where cycle range is c (eg. 360 for degrees)
`Cos( x )`	Cosine for x radians
`Cos( x, c )`	Cosine for x where cycle range is c
`Tan( x )`	Tangent for x radians
`Tan( x, c )`	Tangent for x where cycle range is c

There are corresponding functions for arctan, arccot, sinh, cosh, tanh, coth, arccosh, arctanh, artcoth.

Here's an example using the built-in functions for the float type, and creating our own functions for our own percent type:

```
with Ada.Text_IO, Ada.Numerics.Elementary_Functions,
   Ada.Numerics.Generic_Elementary_Functions;
use Ada.Text_IO, Ada.Numerics.Elementary_Functions;

procedure floatmath is
   type percent is new float range 0.0..1.0;
   package percentMath is new
      Ada.Numerics.Generic_Elementary_Functions( percent );
   use percentMath;

   half : percent := 0.5;

begin
   Put_Line( "Here's some floating point math!" );
   New_Line;

   Put_Line( "4.0 to the power 3.0 is" &
      float'image( 4.0 ** 3.0 ) );

   Put_Line( "The sine of 0.4 radians is" &
      float'image( sin( 0.4 ) ) );

   Put_Line( "The cosine of 180 degrees is" &
      float'image( sin( 180.0, 360.0 ) ) );

   Put_Line( "The square root of 81 is" &
      float'image( sqrt( 81.0 ) ) );

   Put_Line( "50% squared is" &
      percent'image( half ** 2.0 ) );

end floatmath;
```

Here's some floating point math!

```
4.0 to the power 3.0 is 6.40000E+01
The sine of 0.4 radians is 3.89418E-01
The cosine of 180 degrees is 0.00000E+00
The square root of 81 is 9.00000E+00
50% squared is 2.50000E-01
```

When you work with floating point subprograms in libraries outside of Ada, there's a chance that the library will change the floating point arithmetic settings for your CPU. When this happens, use the GNAT.Float_Control package to change your CPU back to GNAT's preferred defaults. There is only one subprogram in this package: reset.

If you are interested in integer operations not covered by the built-in Ada features, the **Interfaces** package (the package used to interface Ada to other languages) defines several bit-shifting functions. In order to use these functions, you'll need to convert (or derrive) your integer values to one of Interfaces' integer types:

Function	*Description*
`Rotate_Left`	Rotate bits in integer types leftward
`Rotate_Right`	Rotate bits in integer types rightward
`Shift_Left`	Shift bits in integer types leftward
`Shift_Right`	Shift bits in integer types rightward
`Shift_Right_Arithmetic`	Arithmetic shift bits in integer types rightward

C: Shift_Left is the equivalent of the C << operator. Shift_Right is the equivalent of the >> operator.

In Gnat, bit-shifting operations are intrinsic. That is, they act as built-in functions and execute quickly.

Here is an example of shifting integer values.

```
with Ada.Text_IO, Interfaces;
use Ada.Text_IO, Interfaces;

procedure shiftMath is
   six : unsigned_64 := 6;
begin

   Put_Line( "Time to do a little bit shifting" );
   New_Line;

   Put_Line( "Our integer is" & six'img );
   Put_Line( "In binary, this is" & six'img );

   Put_Line( "Shifted left once is" &
      Shift_Left( six, 1 )'img );
   Put_Line( "Shifted left twice is" &
      Shift_Left( six, 2 )'img );
   Put_Line( "Shifted right once is" &
      Shift_Right( six, 1 )'img );
   Put_Line( "Arithmetic Shifted right once is" &
      Shift_Right_Arithmetic( six, 1 )'img );

end shiftMath;
```

Time to do a little bit shifting

Our integer is 6
In binary, this is 6
Shifted left once is 12
Shifted left twice is 24
Shifted right once is 3
Arithmetic Shifted right once is 3

12.15 Exception Handling and Traceback Packages

Gnat includes packages for working with exceptions. Using these packages, you can add a message to your exceptions, save exceptions, and examine an exception occurences when they are raised.

Traceback is a technique to examine the run-time stack and identify where an exception occurred. Gnat can identify the specific source file and line where an exception occurred. To use tracebacks in Linux, you must compile your program with the -funwind-tables switch and bind with the -E switch.

[Zero-cost exceptions not covered yet--KB]

```
with Ada.Text_IO,Ada.Exceptions,Gnat.Current_Exception,Gnat.Traceback.Symbolic;
use  Ada.Text_IO,Ada.Exceptions,Gnat.Current_Exception,Gnat.Traceback.Symbolic;

procedure exc is
   e  : exception;
   saved_exception : Exception_Occurrence;

   procedure CrashMe is
   begin
        Raise_Exception( e'identity, "call exec development team" );
   end CrashMe;

begin
     Put_Line( "This is an example of the Ada.Exceptions package" );
     New_Line;

     -- Information about an exception that is not in progress

     Put_Line( "Exception_Name returns a unique name for an exception" );
     Put_Line( "The unique name our exception is " & Exception_Name( e'identity ) );
     New_Line;

     -- Raising an exception with a message

     Put_Line( "raise will raise an exception with no message" );
     New_Line;

     Put_Line( "Raise_Exception will raise an exception with a message" );
     Put_Line( "Raising " & Exception_Name( e'identity ) &
         " with the message 'call exc development team'" );
     Put_Line( "in the subprogram 'CrashMe'." );
     New_Line;
     CrashMe;

exception when occurrence: others =>

     -- Information about an exception that is is in progress
```

```
      Put_Line( "-------------------------------------------------------" );
      Put_Line( "An exception has been raised!  Now in exception handler" );
      Put_Line( "The name of the exception is " & Exception_Name( occurrence ) );
      New_Line;
      Put_Line( "Exception_Message returns the message for this exception" );
      Put_Line( "The message for this exception is '" & Exception_Message( occurrence ) & "'" );
      New_Line;
      Put_Line( "Exception_Information provides the name, message and any traceback information:" );
      Put_Line( Exception_Information( occurrence ) );
      New_Line;
      Put_Line( "The Gnat.Current_Exception package contains short-hand" );
      Put_Line( "versions of Exception_Name, Exception_Message, Exception_Information." );
      Put_Line( "These functions assume you're referring to the current exception" );
      Put_Line( "Gnat.Current_Exception.Exception_Name is " & Exception_Name );
      Put_Line( "Gnat.Current_Exception.Exception_Message is '" & Exception_Message & "'");
      New_Line;
      Put_Line( "The Gnat.Traceback.Symbolic package returns the contents of the" );
      Put_Line( "runtime stack.  That is, it shows which subprograms were being" );
      Put_Line( "executed when the exception occurred." );
      Put_Line( "The symbolic traceback is" );
      Put_Line( Symbolic_Traceback( occurrence ) );
      New_Line;
      Put_Line( "Ada.Exceptions can also save and re-raise in-progress exceptions" );
      New_Line;
      Put_Line( "Save_Occurence can save the in-progress exception" );
      Save_Occurrence( saved_exception, occurrence );
      Put_Line( "Exception now saved." );
      New_Line;
      Put_Line( "Reraise_Occurrence will raise an in-progress exception" );
      Put_Line( "Reraising the one we just saved..." );
      Reraise_Occurrence( saved_exception );

      --Allocate/DeallocateMachineState not covered--for zero-cost exceptions

end exc;
```

```
This is an example of the Ada.Exceptions package

Exception_Name returns a unique name for an exception
The unique name our exception is EXC.E

raise will raise an exception with no message

Raise_Exception will raise an exception with a message
Raising EXC.E with the message 'call exc development team'
in the subprogram 'CrashMe'.

-------------------------------------------------------
An exception has been raised!  Now in exception handler
The name of the exception is EXC.E

Exception_Message returns the message for this exception
The message for this exception is 'call exec development team'

Exception_Information provides the name, message and any traceback information:
Exception name: EXC.E
Message: call exec development team
```

The Gnat.Current_Exception package contains short-hand
versions of Exception_Name, Exception_Message, Exception_Information.
These functions assume you're referring to the current exception
Gnat.Current_Exception.Exception_Name is E
Gnat.Current_Exception.Exception_Message is 'call exec development team'

The Gnat.Traceback.Symbolic package returns the contents of the
runtime stack. That is, it shows which subprograms were being
executed when the exception occurred.
The symbolic traceback is
0x8049cb3 in exc at exc.adb:10

Ada.Exceptions can also save and re-raise in-progress exceptions

Save_Occurence can save the in-progress exception
Exception now saved.

Reraise_Occurrence will raise an in-progress exception
Reraising the one we just saved...
raised EXC.E : call exec development team
Call stack traceback locations:
0x80497eb

Because the reraised exception propogated all the way to the main program and caused it to fail, the final line was actually written to Standard_Error.

12.23 Ada.Real_Time.Timing_Events

A way to execute future events without resorting to creating your own tasks to do it.

12.24 Ada.Executing_Time

Get statistics about the performance of a program.

13 Linux Introduction

This section is an introduction to Linux programming.

13.1 Introduction to Processes

Linux is an operating system. It controls the execution of programs and access to system resources.

Linux can run multiple programs at one. Each running program is referred to as a **process**. The operating system switches to a new process every 100 milliseconds by default--like everything else in Linux, this value can be customized by changing the kernel source code.

Even on a Linux computer with only one user, there are usually several programs running in the background. The one process that is always running is "init": this is the root process, the first process Linux starts when the system is started.

13.1.1 Parents, Children and Families

Processes are grouped together into families. Each time a program starts another process, the new process is called a **child** and the original process is called a**parent**. When the parent unexpectedly stops running, Linux knows enough to stop all the related processes as well.

A **multitasking** program is a program that starts other processes to run simultaneously and assist it with its work. When a child process starts, Linux makes a copy of the parent's resources for the child. For example, all the files that where open to the parent are also open to the child process.

A **multithreading** program refers to independent streams of execution within a single process. Linux refers to each stream as a thread. Ada tasks and protected types, for example, are threads--they do not create entirely new programs they way multitasking works. Instead, they are miniature programs that run inside the parent task, sharing that parent's resources instead of getting their own copy.

Don't confuse Ada tasks with multitasking--the term "task" was chosen before the term "multithreading" became popular.

PID's

The **ps** command shows a list of all processes that are running. Each process has its own identifying number, called the PID (for Process Identification). Here's a typical output from the ps command:

```
$ ps
   PID TTY          TIME CMD
   579 tty1     00:00:00 login
   589 tty1     00:00:00 bash
   617 tty1     00:00:00 ps
```

PPID's

Processes also have a PPID (Parent Process ID) for identifying its parent process. The ps command l option (for "long") shows additional information about a process, including its PPID:

```
$ ps l
   F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
100 S     0   579     1  0  60   0    -   549 wait4  tty1     00:00:00 login
100 S     0   589   579  0  69   0    -   457 wait4  tty1     00:00:00 bash
100 R     0   624   589  0  70   0    -   634 -      tty1     00:00:00 ps
```

In this case, there are three processes in one family. The ps process (PPID 589) has a bash shell as its parent (PID 589). Likewise, the parent of the bash shell is the login command.

Process Groups

For complex programs with many processes, Linux can organize processes into process groups.

The ps lfw options (long, full, wide) will show the PID, the PPID and a simulated graph of groups of related processes.

```
$ ps lfw
   F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
100     0   579     1   0   0  2196 1148 wait4  S    tty1       0:00 login -- root
100     0   589   579  15   0  1828 1060 wait4  S    tty1       0:00 -bash
100     0   689   589  17   0  2484  824 -      R    tty1       0:00  \_ ps lfw
```

Here, it shows that "ps lfw" is a related to the bash shell process, but the bash shell is not related to the login command. The PPID number shows you which process started another, but not which group it belongs to. Because the bash shell process and the ps command are both members of the same group, if you were to kill the bash process, Linux would automatically kill the ps command as well because they are in the same group. With this arrangment, if the bash shell crashed, Linux can return control to the login program and allow you to log in again.

In the same way, if you have many processes in your program, you can group your processes so if they crash unexpectedly, the main program can continue running and take emergency actions. Process groups also provides an easy way to stop several processes at once. For example, a web server could put all the child processes into one group to make it easy to stop them when the server is being shut down.

Stopping Processes

Normally, a process stops when it reaches the end of the instructions it needs to execute.

You can stop a runaway program (or process) with the **kill** command, giving the command the PID number returned by the ps command.

```
kill 624  #killing ps
```

Below we'll discuss stopping processes from inside a program.

A process that runs continually, performing some kind of regular system functions, is called a **daemon** (pronounced "day-mon", a variation on the word "demon"). It's referred to as a daemon as if a little evil creature was running around doing work on its own. If you have a web server running, for example, you could refer to it's process as the web daemon.

13.1.2 Ownership and Permissions

For the sake of security, all programs belong to an **owner** and one or more **groups**. In the same way that all users have a login name (the "owner") and a list of groups they belong to, every program acts as if it's running on behalf of a particular user and their groups. If you're login is "bob", for example, any program you run will be owned by the "bob" login and whatever groups the "bob" login belongs to--your program can only access files that the "bob" login can access. If a program is owned by the superuser login, it doesn't automatically run with the full authority of the superuser.

This can be circumnavigated by the setuid and setgid permissions. When a program is marked with the setuid or setgid active, the program is owned by whatever owner and group owns the file. This

is used for system programs that have to schedule events between multiple people, like the printer daemon.

13.2 Using System and OsLib.Spawn

Often, the first question new Linux programmers ask is, "How to do you run a shell command like ls from a program?"

The easiest, though not necessarily most practical, way to do Linux programming is to use the standard C library's **system** call. System starts a new process, starts a shell running and gives the shell whatever command to execute that you specify. The command executes just as if you typed it in at the command prompt.

For example, to list the files in the current directory on the screen, you'd type:

```
function system( cmd : string ) returns integer;
pragma Import( C, system );
...
Result := system( "ls" & ASCII.NUL );
```

The result is the exit status returned by the command, usually zero if it executed successfully or non-zero if there was an error.

To capture the output of the system command, you can redirect the results to a file using the shell's output redirect, ">". This creates a text file for you to open with Ada.Text_IO.Open.

```
Result := system( "ls > /tmp/ls.out" & ASCII.NUL);
Ada.Text_IO.Open( fd, in_file, "/tmp/ls.out" );
...
```

The following is a simple program to print to the printer with the Text_IO library. It creates a text file and then uses system to run lpr to print it.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure printer is
-- a program for simple printing

function System( s : string) return integer;
pragma import( C, System, "system");
-- starts a shell and runs a Linux command

procedure PrintFile( s : string) is
-- run the lpr command
   Result : integer;
begin
      Result := System( "lpr " & s & ASCII.NUL );
      Put_Line( "Queuing " & s & "..." );
      if Result /= 0 then
           Put_Line( "system() call for lpr failed" );
      else
           Put_Line( "Printing is queued" );
      end if;
end PrintFile;

procedure CreateFile( s : string) is
-- run the touch command
   Result : integer;
begin
     Put_Line( "Creatinig " & s & "..." );
     Result := System( "touch " & s & ASCII.NUL );
```

```
        if Result /= 0 then
            Put_Line( "system() call for touch failed" );
        else
            Put_Line( "Spool file initialized" );
        end if;
    end CreateFile;

    -- the text file to print

    SpoolPath : constant string := "/tmp/spool.txt";
    SpoolFile : File_Type;

begin

    -- To open an out_file in text_io, it must exist.
    -- Create file will create a new spool file.

    -- Set_Output will redirect all output to the
    -- spool file.

    CreateFile( SpoolPath );
    Open( SpoolFile, out_file, SpoolPath);

    Set_Output( SpoolFile );
    -- write the report to printer

    -- Linux normally will not eject a page when
    -- printing is done, so we'll use New_Page.

    Put_Line( "Sales Report" );
    Put_Line( "------------" );
    New_Line;

    Put_Line( "Sales were good" );
    New_Page;
    -- Now, restore output to the screen, close
    -- the file and queue the file for printing
    -- using lpr.

    Set_Output( Standard_Output );
    Close( SpoolFile );

    PrintFile( SpoolPath );

    Put_Line( "Program done...check the printer" );

end printer;
```

Although this program will work for simple applications, another improved program to print using pipes is discussed below.

The system function is convenient but it has a couple of important drawback:

- It's slow: it always opens a shell even if you're only using the shell execute some other program
- It's a security risk: the shell could have aliases defined and what you thought you were running may not be what actually runs.

GNAT's OsLib provides a subprogram like system called **spawn**. It executes a Linux program without starting a shell first. Spawn requires a bit of setup to use. You have to define an array of

access type arguments for the command. Once you invoke spawn, it returns a boolean value indicating whether the spawn succeeded or failed. The following excerpt is from an example program in the OSLib section covered below.

```
Arguments : Argument_List( 1..1 );
-- an argument list for 1 argument

Ls : constant string := "/bin/ls";
-- the program we want to run

WasSpawned: boolean;
RootDir : aliased string := "/";

begin

Arguments(1) := RootDir'unchecked_access;
-- unchecked to avoid useless (in this case) accessibility warning

Spawn( Ls, Arguments, WasSpawned );

if WasSpawned then
   New_Line;
   Put_Line( "End of ls output -- Spawned worked" );
else
     Put_Line( "Spawn failed");
end if;
```

This fragment runs the ls command, prints the results on the screen, and then displays the success of the command on the screen. Notice there are differences between spawn and system:

- spawn only gives you a boolean result
- spawn does not start a shell, so it is more secure
- because there is no shell, you can't run built-in shell commands nor can you redirect output using ">"

If spawn is too limited, many UNIX and Linux programming books tell you how to create your own spawn style subprograms using fork, wait and the exec family of standard C commands. Here is an example of a short C function that executes a program and puts the results (standard output and standard error) to a text file of your choosing, allowing up to three parameters for the command.

```
int CRunIt( char * path, char * outfile,
   char * param1, char * param2, char * param3 ) {

pid_t child;
int fd0, fd1, fd2;
int status;
int i;
if ( !(child = fork()) ) {
   /* Redirect stdin, out, err */
   for (i=0; i< FOPEN_MAX; ++i ) close( i );
   fd0 = open( "/dev/null", O_RDONLY );
   if (fd0 < 0) exit( 110 );
   fd1 = open( outfile, O_WRONLY | O_CREAT | O_TRUNC );
         if (fd1 < 0) exit( 111 );
   fd2 = dup( 1 );
   if (param1[0]=='\0') {
      execlp( path, path, NULL );
   } else if (param2[0]=='\0') {
      execlp( path, path, param1, NULL );
```

```
        } else if (param3[0]=='\0') {
            execlp( path, path, param1, param2, NULL );
        } else {
            execlp( path, path, param1, param2, param3, NULL );
        }

        /* if we got here, file probably wasn't found */

        exit( errno );
    }
    wait( &status );
    if ( WIFEXITED( status ) != 0 )
        status = WEXITSTATUS( status );
    return status;
}
```

 It is possible to rewrite this subprogram into Ada, but it's easier in C because of the constants, macros and execlp takes a variable number of parameters.

This function returns some special exit status values: 110 if /dev/null couldn't be opened for standard input, and 111 if the output file you specified couldn't be opened.

```
function CRunIt( cmd, outfile, parm1, parm2, parm3 : string ) return integer;
pragma Import( C, CrunIt, "CRunIt" );
..
Result := CrunIt( "/bin/ls" & ASCII.NUL, -- executable to run
    "/tmp/ls.out" & ASCII.NUL, -- where output should go
    "" & ASCII.NUL, -- parameter 1 (none)
    "" & ASCII.NUL, -- parameter 2 (none)
    "" & ASCII.NUL ); -- parameter 3 (none)
```

An important part of running commands like this is deciding on temp file names that won't be used if two copies of the program are run at the same time. There's two ways to do this:

- Use the standard C function tmpfile (discussed below)
- Find the process number of your program with the standard C function getpid and add it to your filename

If you are interested in accessing Linux more directly, read the next section.

13.3 The Linux Environment

Because Linux is based on decades old UNIX, the Linux development environment is like an onion. Originally, all UNIX programs accessed the operating system through kernel calls. As time went on, new ways of accessing the kernel were added, and new libraries were made to encapsulate common problems. Finally, Ada itself comes with many standard packages and features to work with the operating system.

For the Ada programmer, it's not difficult to work with the operating system. Gnat comes with many standard libraries. These are built using the standard C libraries. The C libraries, in turn, work by accessing the kernel. The problem is to decide which of the many ways to access Linux is the best suited for your program.

For simple tasks, using the standard Ada packages is the most straightforward way of working with Linux. However, the standard Ada packages were designed for portability: they only allow access to the most basic Linux features, and they aren't particularly fast when doing it. Working with the standard C libraries is a compromise between speed and convenience: the C libraries give you more

features, but require you to import C function calls and convert between Ada and C data types. For maximum speed and flexibility, you can only work with kernel, but then you risk extra work by rewriting subprograms that already exist in the standard libraries.

To understand the differences between these layers, consider the problem of allocating dynamic memory. Usually you allocate memory with the Ada **new**statement. Where does new get its memory? It uses the standard C library's **malloc** function. But where does malloc get its memory? It gets it from the Linux kernel call **mmap** (memory map). The most direct way to get memory, and the method that gives you the most control, is mmap. On the other hand, the easiest way would be to use new and to let Ada allocate the memory and manage the details for you.

It's the same with multithreading and sequential files. Multitasking is based on LinuxThreads, a part of the standard C library, which in turn is based on the kernel's clone function. Sequential files are based on the standard C library's stream functions, which in turn are implemented using the kernel's file handling functions.

Figure: Ways of Doing the Same Thing

	Memory Allocation	Multithreading	Sequential Files
Standard Ada:	new	task / protected	Ada.Sequential_IO package
Standard C libraries:	malloc	LinuxThreads functions	C stream functions
Linux Kernel:	mmap	clone function	kernel file functions

As you move down the list from the standard Ada libraries to the Linux kernel, your program becomes more platform specific. A program that uses new will run on any operating system that can run Ada 95. If you use malloc, your program will run on any operating system that uses has the standard UNIX C libraries available. If you use mmap, your program will run on any Linux computer—not just Intel-based Linux, but any flavour of Linux, include Sun UltraLinux or Apple Mklinux. Remember that Linux is a portable operating system: all versions of Linux will have mmap available.

13.4 Standard C Libraries

We've already covered many of the standard Ada libraries and language features in the above sections.

The **standard C libraries** define standardized subprograms that exist across most version of UNIX. They are "wrappers": that is, they do not work directly with the kernel. For example, the standard C libraries are not a part of the kernel, but they use the kernel to implement standard C functions that you'd find across many UNIX'sThe main C library, called **libc**, is automatically loaded by Gnat and you can import subprograms from it directly. Other standard C libraries, such as C's math library, **libm**, or the password encryption library, **libcrypt**, need to be linked in at the linking stage. The standard C library calls are defined in the online manual pages.

13.5 The Linux Kernel

There are three basic ways to work with the Linux kernel: kernel calls, devices and the proc file system. Because Linux is organized around files, the last two ways let you operate devices and get system information simply by opening and working with files using standard Linux file operations. This is makes it easier to work with Linux, but it also can make some tasks hard to visualize because you have to do them through a sequence of abstract file operations.

In a few cases, standard libraries and kernel calls have names that overlap, which can be confusing.

In addition, Linux sometimes provides alternative versions of system calls based on different flavours of UNIX. For example, there are two different system calls to assign an environment variable, one based on BSD UNIX (setenv) and another based on the POSIX standard (putenv). Both do exactly the same thing, but their parameters are slightly different. Linux provides both to make it easier to move programs written for other versions of UNIX to Linux. But for the Ada programmer, you have to choose the one that's easiest to use in your program.

13.5.1 Kernel Calls

Kernel calls (sometimes called system calls or syscalls) are basic operations that are implemented directly in the kernel. There are C libraries which supply some thin wrappers on the calls which do some of the setup and cleanup for your. Most kernel calls are in described in the C header file unistd.h. The appendices contain a list of the Linux kernel calls.

The kernel calls are documented in the online manual pages, but these are sometimes out of date due to the ever-changing nature of Linux.

13.5.2 Devices

The /dev directory defines **device files**. These files let you work with devices connected to your computer by using standard file operations. Devices can include hard drives, sound cards and the system console (the text display).

Devices are recognized by their names:

- /dev/hd devices: IDE hard drives, CD-ROMs, etc.
- /dev/sd devices: SCSI hard drives, CD-ROMs, etc.
- /dev/fd devices: floppy drives
- /dev/console device: the text display
- /dev/tty devices: the pseudo terminals. /dev/tty is an alias for the current tty terminal or ttyS serial port
- /dev/ttyS devices: the serial ports
- /dev/lp devices: the parallel ports (lp stands for line printer)

In addition, most distributions define the following links:

- dev/cdrom: an alias for the main CD-ROM device
- /dev/modem: an alias for the the port/device the modem is connected to
- dev/mouse: an alias for the port/device the mouse is connected to
- /dev/fd0: an alias for the first floppy drive

There are more device files than there are devices on a computer. For example, there may be 32 serial port device files defined, but that doesn't mean that there are actually 32 serial port on the computer. You will have to open the device and check for an error if it does not exist.

For example, opening /dev/lp1 and writing a file to it writes the file as raw data to the first parallel port printer. Information on how these devices work is usually found in the How-To's and other system documentation.

Special functions specific to a device are programmed with the ioctl() function. For example, you'd use ioctl() on /dev/dsp to set the sound volume on your sound card.

The documentation for device files are often difficult to find. Sometimes documentation is contained in the kernel documentation (the /usr/doc/kernel.. directory) or in the kernel C header files (the /usr/src/linux/include/... directories). A list of some of the ioctl operations are listed in an appendix.

13.5.3 Proc File System

The /proc directory isn't really a directory at all. That is, it's a fake directory that's not physically on the hard disk, but you can still look into it and open the files it contains. The **proc file system** contains system information that you can access by opening the files and reading them. Some proc files may be written to in order to change system settings. For example, there are files that give you the information on how busy the CPU is, how much free memory you have, and the environment variables for the current process.

The contents of these files are described in the proc man page.

13.5.4 AudioCD: An Example Program

The following CD-ROM audio CD player illustrates kernel calls, a standard C library function, and using a device file.

```
with Ada.Text_IO, System;
use Ada.Text_IO;
procedure audiocd is
   -- Sample program for playing audio CD's

   -- DEVICES
   --
   -- This section deals with device files, in particular,
   -- the cdrom device

   DevCDROM : constant string := "/dev/cdrom";
   -- path to the CDROM device, usually /dev/cdrom

   type ioctlID is new integer;
   type aFileID is new integer;
   -- Define these as separate types for error checking.
   -- A ioctlID is never the same as a FileID.

   type byte is new integer range 0..255;
   for byte'size use 8;

   CDROMPLAYTRKIND : constant ioctlID := 16#5304#;
   CDROMSTOP : constant ioctlID := 16#5307#;
   CDROMSTART : constant ioctlID := 16#5308#;
   -- various CDROM ioctl functions as mentioned in the
   -- CDROM documentation in /usr/doc/kernel...

   type aDummyParam is new integer;
   -- define this as a separate type to make sure nothing
   -- important is used as a third parameter to ioctl_noparam

   -- a version of ioctl for functions that don't
   -- use a third parameter

   procedure ioctl_noparam( result : out integer;
         fid : aFileID;
         id : ioctlID;
         ignored : in out aDummyParam );
   pragma import( C, ioctl_noparam, "ioctl" );
   pragma import_valued_procedure( ioctl_noparam );
   type cdrom_ti is record
         start_track, start_index : byte;
         end_track, end_index : byte;
```

```
    end record;

    -- from /usr/src/linux/include/linux/cdrom.h
    -- PLAYTRKIND ioctl function uses cdrom_ti record

    procedure ioctl_playtrkind( result : out integer;
              fid : aFileID;
              id : ioctlID;
              info : in out cdrom_ti );
    pragma import( C, ioctl_playtrkind, "ioctl" );

    pragma import_valued_procedure( ioctl_playtrkind );
    -- KERNEL CALLS
    --
    -- Calls to the Linux kernel (besides ioctl).

    procedure open( id : out aFileID;
              path : string;
              flags : integer );
    pragma import( C, open, "open");
    pragma import_valued_procedure( open );
    -- open is a kernel call to open a file

    procedure close( result : out integer; id : aFileID );
    pragma import( C, close, "close");
    pragma import_valued_procedure( close );
    -- close is a kernel call to close a file

    -- C LIBRARY CALLS
    --
    -- Calls to the standard Linux C libraries

    procedure perror( prefixstr : string );
        pragma import( C, perror, "perror");
    -- perror is a standard C library call to print
    -- the last error message from a kernel call or the
    -- standard C libraries on the screen

    cd : aFileID;
    playinfo : cdrom_ti;
    dummy : ADummyParam;
    ioctl_result : integer;
    close_result : integer;
    ch : character;

begin

    Put_Line( "This program plays an audio CD in your CDROM drive" );
    New_Line;

    -- open the /dev/cdrom file so we can control the CDROM drive
    -- using ioctl

    Put_Line( "Openning " & DevCDROM & "..." );
    Open( cd, DevCDROM & ASCII.NUL, 0 );
    if cd < 0 then
          perror( "Error openning CDROM drive" );
    end if;
    -- start the CDROM drive
```

```
    Put_Line( "Spinning up cdrom..." );
    ioctl_noparam( ioctl_result, cd, CDROMSTART, dummy );
    if ioctl_result < 0 then
            perror( "Error spinning up the CDROM drive" );
    end if;

    -- display menu

    New_Line;
    Put_Line( "1 = Play, 2 = Quit" );
    New_Line;

    -- Main loop. Repeat until 2 is selected.

    loop
        Put( "Select a function (1-2): " );
        Get( ch );
        case ch is
        when '1' => playinfo.start_track := 1; -- first track
                                    playinfo.start_index := 0; -- no effect
                                    playinfo.end_track := 9; -- final track (inclusive)
                                    playinfo.end_index := 0; -- no effect
                ioctl_playtrkind( ioctl_result, cd, CDROMPLAYTRKIND, playinfo );
        when '2' => ioctl_noparam( ioctl_result, cd, CDROMSTOP, dummy );
                exit;
        when others => Put_Line( "Pardon?" );
        end case;

        if ioctl_result < 0 then
                perror( "Error controlling CDROM drive" );
        end if;

    end loop;

    -- Close the CDROM device

    Close( close_result, cd );
    if close_result < 0 then
                perror( "Error controlling CDROM drive" );
    end if;

end audiocd;
```

13.6 Standard Input/Output/Error

Linux defines three default I/O streams. Standard input (Linux file id 0) is the file form which all keyboard input normally comes to your program. Standard output (Linux file id 1) is the default where the output of your program is written. Standard error (Linux file id 2) is the file to which error messages are written. Usually, standard input is from the keyboard, and standard output and error are directed to the screen. There are two output streams so that if you redirect the results of a command to a file, such as "ls > temp.out", any errors that occur will still appear on the screen.

The following program writes messages to standard output and standard error using Text_IO:

```
with ada.text_io;
use ada.text_io;

procedure stderr is
-- an example of writing messages to standard error
```

```
begin

   Put_Line( "This is an example of writing error messages to stderr" );
   New_Line;

   -- Text_IO defines a file called Standard_Error, always open,
   -- that you can write error messages to.

   Put_Line( Standard_Error, "This message is on standard error" );
   Put_Line( "This message is on standard output" );
   New_Line;

   -- you can use Set_Output to send all Put_Line's to Standard_Error

   Set_Output( Standard_Error );
   Put_Line( "This is also on standard error");

   Set_Output( Standard_Output );

   Put_Line( "But this is on standard output");
end stderr;
```

```
This is an example of writing error messages to stderr

This message is on standard error
This message is on standard output

This is also on standard error

But this is on standard output
```

Everything looks normal until you redirect the output of the program. This is the result when the standard output is redirected to a file called "out.txt". The error messages aren't redirected.

```
$ stderr > out.txt
This message is on standard error
This is also on standard error
```

To include the file and line number where an error occurred, use the **GNAT.Source_Info.Source_Location** function. This function returns a string in the standard GCC error format, suitable for beginning an error message.

C: GNAT.Source_Info.Source_Location is the equivalent of `__FILE__:__LINENO__`.

```
with Ada.Text_IO, GNAT.Source_Info;
use  Ada.Text_IO, GNAT.Source_Info;

procedure source_error is
   -- example of GNAT.Source_Info.Source_Location
   i : integer;
   j : integer := 0;
begin
   i := 5/j; -- division by zero
exception when others =>
   put_line( standard_error, Source_Location & ": exception raised" );
end source_error;
```

```
$ ./source_error
source_error.adb:10: exception raised
```

The error message comes from the put_line on the 10th line in source_error.adb. To identify where an exception occurred, you'll need to use the Gnat exception handling packages described in 12.15.

13.7 Linux Binary Formats

Many people do not think of binary files as having a format because they contain machine code instructions. However, binary files are more than just raw microprocessor instructions. They contain information such as the type of the binary file and a list of all DLL's needed by the program to run.

Linux binary files come in two formats: **ELF** (Executable and Linking Format) and "**a.out**". They both have different characteristics and neither is better than the other. Most distributions let you install ELF or a.out compilers. The version of gnat for Linux is compiled for ELF. ELF is current Linux standard primarily because it provides better support for shared libraries.

Fun Fact: "a.out" is an abbreviation for "assember output".

Since the kernel has to do the loading and execution of programs, support for ELF, a.out or both must be selected when the kernel is compiled. Otherwise, the kernel will not recognize the format and will be unable to run the binary file.

13.9 Linux Libraries

A **library** is a set of object files that have been combined into a single file. You create a Linux library using the **ar** (archive), which takes ".o" object files can compiles them into or out of .a archive file. A Linux library files all start with "lib" and end with ".a". The command parameters for ar are a bit odd: check the man page for more details.

The ar comamand has many options. Two useful variations are **ar cr** (create a new archive and add object files to it) and **ar t** (show a list of all object files in the archive file). See the example below for how these work.

A library which is directly linked into a program, so that it is added to the executable, is called a **static library**. To link in a static library, just include it's name with the gnatlink command (or gnatmake on simple projects). By default, gnat's own libgnat library is always static, because of concerns over multithreading.

For example, if you install and compile the official JPEG library sources, a static library file is created named libjpeg.a. To link this library into your program, you'd include **-ljpeg** when linking. Note that you don't use the entire name of the file: gnat assumes there is a beginning "lib" and ending ".a".

A **shared library** (or DLL, dynamic link library) is a library that is loaded into memory and shared between programs. It's not actually saved as part of the executable. All Linux shared libraries end in .so ("shared object"). They are loaded when a program is executed.

You can create your own static and shared libraries. The GCC Ada user guide recommends using the GNAT project manager to create a project file for building your library. For example:

```
project mylib is
   for Source_Dirs use ("src1", "src2");
   for Object_Dir use "obj";
   for Library_Name use "mylib";
   for Library_Dir use "lib";
   for Library_Kind use "dynamic";
end mylib;
```

However, libraries can also be built without the project manager.

 if you forget the —fPIC switch, your shared library will still load, but you won't be able to share the code between applications that use the library.

 -fpic will also work on Intel processors because there is no maximum imposed for the global offset table, but it may not work on other processors: the —fPIC switch is the preferred method.

The shared libraries are usually stored in standard Linux directories, like /lib or /usr/lib. Once you copy a shared library into one of these directories, you have to run ldconfig to register the new shared library, otherwise Linux will not load it.

To load a shared library from the current directory instead of the standard Linux directories, use -L. (where period is the current directory).

Shared libraries have the advantage of making executable's smaller, but they are slower to load and execute and use up more memory than static libraries. The also have the advantage of being able to be ugraded separately from your program, provided you don't change the format of any of the subprograms.

Too many shared libraries mean that you have small files scattered throughout the lib directories, making your program harder to maintain.

Generally speaking, only subprograms that will be shared between programs should be shared libraries, and you should combine small, related libraries into one shared library. For example, all the standard C libraries are compiled into one shared library on Linux: libc.so.a.

Supposed you have a package called "stuff" and you want to package it within a shared library named "mylib". Create a dummy main program like this:

```
-- this main program will only be used by gnatmake to compile the
-- files I want in my library "mylib"
with stuff;
procedure shared_library is
begin
   null;
end shared_library;
```

This library "mylib" can be built like this (in a Makefile):

```
rm *.ali *.o
gnatmake -c shared_library -cargs -fPIC
rm shared_library.o shared_library.ali
gcc -shared -fPIC -o libmylib.so *.o
rm *.o
chmod -w stuff.ali
```

- Use gnatmake to compile the shared library using Position Independent Code.
- Remove the dummy main program (we don't want this in our library)
- Use gcc to build the shared library
- Remove the object files (they are in the library)

- Make the .ali files read-only so gnatmake won't try to build the missing .o files

If you have a program called "main.adb" that refers to packages in your shared library like this

```
with stuff;
with ada.text_io;

procedure main is
begin
  ada.text_io.put_line( stuff.i'img );
end main;
```

You can build and run main with these commands:

```
$ gnatmake main.adb -largs -lmylib
$ ./main
./main: error while loading shared libraries: libmylib.so: cannot open shared object file: No such file or directory
$ LD_LIBRARY_PATH="." ./main
  5
```

By default, a program will not look for shared libraries in the current directory. You will need to specify the search path with the LD_LIBRARY_PATH shell environment variable or ldconfig. In this case, LD_LIBRARY_PATH only exists when main is run but it could also be exported.

A static (.a) library can be made in a similar way, using ar instead of building files with Position Independent Code:

```
rm *.ali *.o
gnatmake -c shared_library
rm shared_library.o shared_library.ali
ar rc libmylib.a *.o
rm *.o
chmod -w stuff.ali
```

The main program can be built and run with:

```
$ gnatmake main.adb -largs -lmylib
$ ./main
  5
```

Use -L in the link options if mylib isn't visible to gnatmake. Since the library is static, it's compiled into main and there's no need for a library loading path.

If you want the contents of your packages to be used by non-Ada languages, then you'll have to use the usual pragmas (Convention, Export, etc.) in our source code to make them visible. Also, bind the library with -n (no main program): the non-Ada program will need to run adainit/adafinal in the library to start and stop the GCC Ada environment prior to making calls to the library. These procedures can be renamed with the -L option.

[windows from usenet]

makedll.bat

```
gcc -c beep.adb
gnatbind -n beep.ali
gnatlink beep.ali -o beep.jnk -mdll -Wl,--base-file,beep.base
dlltool --dllname beep.dll --def beep.def --base-file beep.base --output-exp
beep.exp --output-lib libbeep.a
gnatbind -n beep.ali
gnatlink beep.ali -o beep.jnk beep.exp -mdll -Wl,--base-file,beep.base
dlltool --dllname beep.dll --def beep.def  --base-file beep.base --output-exp
```

beep.exp --output-lib libbeep.a
gnatbind -n beep.ali
gnatlink beep.ali beep.exp -o beep.dll -mdll

This is a def file I am using.

beep.def

EXPORTS

DllGetClassObject=DllGetClassObject@12 @2
DllCanUnloadNow=DllCanUnloadNow@0@3
DllRegisterServer=DllRegisterServer@0@4
DllUnregisterServer=DllUnregisterServer@0@5

NOTE: Since the exported functions or STDCALL, I need to provide the number of bytes used for parameters. If you were using standard pragma C stuff it would be:

MYFunction@XXXX

where XXXX is what ordinal in the DLL - BTW you can leave this out all together if you don't need it and just put the name of the function on the line.

[end]

Gnat has an option called **-static** which will link all shared libraries into your executable as if they were static libraries. This makes your executable completely self-contained, but may violate GPL licensing restrictions on certain libraries.

Gnat comes with one shared library, libgnat.a. If you link a gnat program without the -static option, you have to copy this file into a standard library directory (e.g. /lib) and run ldconfig so that Linux will be able to find the gnat library when executing your programs. Gnat always automatically links in the library: you never have to type "-lgnat" explicitly when linking.

13.10 Libc5, Libc6 and Upward Compatibility

One of the difficulties with Linux programming is that the standard libraries are seldom upwardly compatible. For example, the ncurses 3 console drawing package is completely incompatible with ncurses 4. Likewise, ncurses 4 is completely incompatible with ncurses 5. In an open source enviroment, subprogram parameters can appear and disappear with each new release.

The Gnat compiler always links the standard C library into your programs. As a result, you have to be aware of the problems with Linux's standard C library, even if your Gnat program doesn't call its subprograms explicitly. Your Gnat executable always connected to the C library it was compiled against.

Like most open source libraries, the standard C library isn't upwardly compatible. Inspite of the fact libc version 5 and libc version 6 (now called glibc 2.0) share the same name, many functions and names have been changed between the two versions. Multithreading under libc5 is done with the linuxthreads library, an implementation of "pthreads". LinuxThreads is based the Posix 1003.1c thread model, with a few extensions. Linuxthreads is a built in part of libc6.

In the Linux world, even minor changes between libraries will create problems. There is very little upward compatibility. For example, a program may run on libc 6.0.x but won't run on libc 6.0.y because some symbol names have changed. Because of this dependency, your programs should be compiled against a specific Linux distribution. Don't assume that if a Red Hat disk and a Slackware disk are published in the same month that they are using exactly the same versions of the C library. By the same token, don't assume you can simply include libc 6.0.y with your program and update

the user's version of libc by installing yours overtop. This can cause many programs to crash if they can't find the particular version of libc that they need.

The same is true of the gnat library, libgnat.a. ACT does not guarantee that a program compiled for gnat 3.11's gnat library will run with gnat 3.12's gnat library. In fact, it probably won't.

13.11 Linux Basics

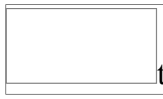
[To be filled in – KB]

14 Linux Programming

14.1 Gnat OS Library

Ada	*Description*	*C Equivalent*
`create_file`	Create a Linux file	`creat`
`delete_file`	Delete a Linux file	`unlink`
etc.		

The gnat OS library, **gnat.os_lib**, provides common UNIX operations independent of what flavour of UNIX gnat is running on. It provides an extensive set of file utilities as well as the ability to run blocked and non-blocked child processes.The price for this low-level OS support is the need to use a lot of addresses, 'access and C strings.

 there is also a thin binding available for basic C stream functions, described below.

```
with text_io, gnat.os_lib;

use text_io, gnat.os_lib;
procedure ostest is
    fd : File_Descriptor;
    FilePath : constant string := "testfile.xxx" & ASCII.NUL;

    -- for write test

    FirstLine : constant string := "This is the first line in the file";

    AmountWritten : integer;
    -- for time stamp test
    ts : OS_Time;
    Year : Year_Type;
    Month : Month_Type;
    Day : Day_Type;
    Hour : Hour_Type;
    Minute : Minute_Type;
    Second : Second_Type;
    -- for location test
    sp : String_Access;
    -- for delete test

    WasDeleted : boolean;

    -- for spawn test

    Arguments : Argument_List( 1..1 );
    Ls : constant string := "/bin/ls";
    WasSpawned: boolean;
    RootDir : aliased string := "/";
begin
    Put_Line( "This is an example of the Gnat's OS library:" );
```

```
New_Line;
Put_Line( "Creating a new file..." );
fd := create_file( FilePath'address, Binary );
if fd = invalid_fd then
    Put_Line( "Unable to create " & FilePath );
else
    Put_Line( "Created " & FilePath );
end if;
New_Line;
Put_Line( "Getting the timestamp on the file..." );
ts := File_Time_Stamp( fd );
GM_Split( ts, Year, Month, Day, Hour, Minute, Second );
Put_Line( "The time stamp is" &
    Year'img & "/" & Month'img & "/" & Day'img &
    Hour'img & ":" & Minute'img & ":" & Second'img );
New_Line;
Put_Line( "Writing to the file..." );
Put_Line( FirstLine );
AmountWritten := Write( fd, FirstLine'Address, FirstLine'Length );
Put_Line( "Wrote" & AmountWritten'img & " bytes" );
Put_Line( "The file length is" & File_Length( fd )'img );
New_Line;
Close( fd );
Put_Line( "Closed the file" );
New_Line;
Put_Line( "Locating the file we just made..." );
sp := Locate_Regular_File( File_Name => FilePath,
Path => GetEnv( "PATH" ).all );
Put_Line( "The file is '" & sp.all & "'" );
New_Line;
Put_Line( "Deleting the file..." );
Delete_File( FilePath'address, WasDeleted );
if WasDeleted then
  Put_Line( "File was deleted" );
else
    Put_Line( "File was not deleted" );
end if;
New_Line;
Put_Line( "Running ls / ..." );
New_Line;
Arguments(1) := RootDir'unchecked_access;
-- unchecked to avoid unless accessibility warning
Spawn( Ls, Arguments, WasSpawned );
if WasSpawned then
    New_Line;
    Put_Line( "End of ls output - Spawned worked" );
else
    Put_Line( "Spawn failed" );
end if;
```

```
    New_Line;


end ostest;
```

This is an example of the Gnat's OS library:

Creating a new file...

Created testfile.xxx
Getting the timestamp on the file...
The time stamp is 1998/ 12/ 18 23: 42: 24
Writing to the file...
This is the first line in the file
Wrote 34 bytes
The file length is 34
Closed the file

Locating the file we just made...

The file is './testfile.xxx'
Deleting the file...
File was deleted
Running ls / ...

STARTUP

System.map
System.old
bin
boot
cdrom
dev
dosc
etc
fd
home
lib
lost+found
mnt
opt
proc
root
sbin
tmp
usr
var
vmlinuz
vmlinuz.old
End of ls output - Spawned worked

14.2 Installing Binding Packages

A variety of Ada packages exist to allow you to call C libraries from Ada. These packages are called **bindings**. For example, there are Ada bindings to Motif, TCL, WWW CGI and Posix (that is, the kernel).
A **thin binding** gives you direct access to library calls. A **thick binding** provides indirect access, where the package does some setup before invoking the library calls. The gnat.os_lib library is an example of a thick binding to basic Linux file operations.

When installing binding libraries:

- Make sure that the filename endings are the right ones for gnat. Different compilers use different conventions.
- Compile the binding package(s).
- Install the library the binding is for (if necessary)
- Include -lname on the link line, where name is the library. Remember that the order of the -l's is important.

14.3 Catching Linux Signals

A programs has to be able to respond to unexpected events. What do you do when somebody types control-C? How do you gracefully stop the program when somebody kills it with the kill command? These unexpected events are referred to a *signals* in Linux, and Gnat provides libraries for you to "catch" these signals and respond to them gracefully.
The standard Ada 95 package **Ada.Interrupts** and its children handle unexpected operating system events. Under Linux, these packages provide support for signal handling.

A complete list of Linux signals is listed in an appendix. The
package **Ada.Interrupt.Names** defines the names of these signals for you.
Signal Handlers are protected type procedures with no parameters. The body of the procedure performs whatever actions you want to do when you receive a signal.

For example, to catch the SIGTERM signal, the signal that indicates that the program has been killed with the "kill" shell command, you can write a handler like this:

```
protectedbodySignalHandler is
   procedure HandleSIGTERM is
-- normal kill signal handler
   begin
      Put_Line( "Ouch! I've been killed!" );
   -- perform any other cleanup here
  end HandleSIGTERM;
end  SignalHandler;
```

To put the handler in place permanently, use **pragma Attach_Handler**.

```
pragma Attach_Handler( HandleSIGTERM, SIGTERM );
```

Now whenever your program receives a SIGTERM signal, your handler will automatically run.

If you don't want to install a permanent handler, a handler can be installed or changed while the program is running. To indicate that a procedure is an interrupt handler that can be installed at a later time, use **pragma Interrupt_Handler**.

```
pragma Interrupt_Handler( HandleSIGTERM );
```

Gnat automatically handles one signal for you: SIGINT, the interrupt signal. This is the signal that is sent to your program when control-c is pressed. If you want to handle control-c presses yourself, you have to use **pragma Unreserve_All_Interrupts**. Despite it's long name, this pragma simply tells Gnat to ignore SIGINT's.

Certain signals can never be caught. SIGUNUSED, the unused signal, can't be caught for obvious reasons. Some signals are used by the multithreading software and are not available for use in applications. In particular, if you are running native Linux threads, you can't catch SIGFPE, SIGILL, SIGSEGV, SIGBUS, SIGTRAP, SIGABRT, SIGINT, SIGVTALRM, SIGUNUSED, SIGSTOP, or SIGKILL. On 2.0 kernels or older, native Linux threads use SIGUSR1 and SIGSUR2 and are not available. If you're running FSU threads, then SIGALRM is also not available.

Ada.Interrupts also contains several subprograms for signal handling.

- **Is_Reserved** is true if a particular signal is uncatchable.
- **Is_Attached** is true if a particular signal has a handler attached.
- **Current_Handler** returns a pointer to the handler for a particular interrupt.
- **Exchange_Handler** will put a new handler in place and return a pointer to the previous handler.
- **Detach_Handler** will uninstall a handler

The following package sets up three signal handlers, which display a message at set the EMERGENCY_SHUTDOWN variable to true. The demo program demonstrates some of the Ada.Interrupts subprograms and enters into a slow loop. The main program was killed with the "kill —SIGPWR" shell command, simulating a power failure signal.

```
with Ada.Interrupts.Names;

use Ada.Interrupts, Ada.Interrupts.Names;
package SigHand is

   -- Package to handle basic Linux signals

   pragma Unreserve_All_Interrupts;

   -- Gnat will no longer handle SIGINT for us
   EMERGENCY_SHUTDOWN : boolean := false;
   -- set in the event of a signal to shut down the program
   -- SignalHandler will handle the signals independently
   -- from the main program using multithreading
   protected SignalHandler is

      procedure HandleControlC;

      pragma Attach_Handler( HandleControlC, SIGINT );
      -- SIGINT (Control-C) signals will be intercepted by
      -- HandleControlC
      procedure HandleKill;
      pragma Attach_Handler( HandleKill, SIGTERM );
      -- SIGTERM (kill command) signals will be intercepted by
      -- HandleKill
```

```
        procedure HandlePowerFailure;
        pragma Attach_Handler( HandlePowerFailure, SIGPWR );
        -- SIGPWR (power failure signal) intercepted by
        -- HandlePowerFailure
    end SignalHandler;

end SigHand;


with Ada.Text_IO;

use Ada.Text_IO;
package body SigHand is

    -- Package to handle basic Linux signals

protected body SignalHandler is

    -- This protected type contains all our signal handlers
    procedure HandleControlC is
    -- Control-C signal handler
    begin
        if EMERGENCY_SHUTDOWN then
            Put_Line( "HandleControlC: The program is already shutting down" );
        else
            Put_Line( "HandleControlC: Control-C was pressed, shutting down" );
      end if;
      EMERGENCY_SHUTDOWN := true;

    end HandleControlC;

    procedure HandleKill is

        -- normal kill signal handler
    begin
        if EMERGENCY_SHUTDOWN then
            Put_Line( "HandleKill: The program is already shutting down" );
        else
            Put_Line( "HandleKill: Program is shutting down" );
        end if;
        EMERGENCY_SHUTDOWN := TRUE;
    end HandleKill;
    procedure HandlePowerFailure is
    -- power failure handler
    begin
        if EMERGENCY_SHUTDOWN then
            Put_Line( "HandlePowerFailure: The program is already shutting down" );
        else
            Put_Line( "HandlePowerFailure: Program is shutting down" );
      end if;
        EMERGENCY_SHUTDOWN := TRUE;

    end HandlePowerFailure;

    end SignalHandler;
```

```
end SigHand;


with Ada.Text_IO, SigHand, Ada.Interrupts.Names;
use Ada.Text_IO, SigHand, Ada.Interrupts, Ada.Interrupts.Names;
procedure SigDemo is
   Handler : Parameterless_Handler;
   Counter : integer := 2;
begin
   Put_Line( "This program demonstrates signal handling." );
   Put_Line( "To stop this program, type Control-C or " );
   Put_Line( "kill it with the shell kill command." );
   New_Line;
   -- Is_Reserved example
   if Is_Reserved( SIGTERM ) then
      Put_Line( "The SIGTERM handler is reserved" );
   else
      Put_Line( "The SIGTERM handler isn't reserved" );
   end if;
   -- Is_Reserved example
   if Is_Attached( SIGINT ) then
      Put_Line( "There is a SIGINT handler installed" );
   else
      Put_Line( "There is no SIGINT handler installed" );
   end if;
   -- Current_Handler example
   Put_Line( "Testing SIGTERM handler..." );
   Handler := Current_Handler( SIGTERM );
   -- Current_Handler gives a callback to the handler
   Handler.all;
   -- run the handler callback
   if EMERGENCY_SHUTDOWN then
      Put_Line( "Handler works" );
   else
      Put_Line( "Handler doesn't work" );
   end if;
   -- test complete: reset emergency shutdown flag
   EMERGENCY_SHUTDOWN := false;
   -- a long loop
   New_Line;
   Put_Line( "The number is " & Counter'img );
   loop
      exit when EMERGENCY_SHUTDOWN;
      Counter := Counter * 2;
      Put_Line( "Doubling, the number is " & Counter'img );
      delay 1.0;
```

```
    end loop;
    Put_Line( "The program has shut down" );
end SigDemo;
```

```
This program demonstrates signal handling.
To stop this program, type Control-C or
kill it with the shell kill command.
The SIGTERM handler isn't reserved
There is a SIGINT handler installed
Testing SIGTERM handler...
HandleKill: Program is shutting down
Handler works
The number is 2
Doubling, the number is 4
Doubling, the number is 8
Doubling, the number is 16
Doubling, the number is 32
Doubling, the number is 64
Doubling, the number is 128
Doubling, the number is 256
Doubling, the number is 512
HandlePowerFailure: Program is shutting down
The program has shut down
```

14.4 Working with the Command Line

Ada	*Description*	*C Equivalent*
Function Command_Name return string;	The name of this command (path?).	argv[0]
Function ArgumentCount return natural;	The number of arguments.	argn
Function Argument(n : natural) return string;	The n'th argument.	argv[n]
Procedure Set_Exit_Status(e : Exit_Status);	The exit status to return.	exit(e)

Ada interacts with the outside world through the standard Ada package **Ada.Command_Line.** Suppose you have an Ada program called "myprog" and a user types in the following command: "myprog -v sally.com".

- "myprog" is the name of the command.
- "-v" and "sally.com" are arguments to the command.

Command_Name returns the name of the command. If the program was run from a shell, it returns the name as typed in by the user. In the above example, Command_Name returns "myprog". **ArgumentCount** returns the number of arguments, not including the name of the program. The shell determines how arguments are grouped together, but typically each argument is separated by a space. In the above example, there are two arguments, "-v" and "sally".

Argument returns an argument. In the above example, argument(1) returns "-v".

Set_Exit_Status gives Ada the error code you want to return when the program is finished running. Ada defines two Exit_Status values, **Success** and **Failure** . Since Exit_Status is just an

integer, you can return other values. Zero indicates that the program ran without error, non-zero values indicate an error. The predefined values of Success and Failure are 0 and 1.

Properly flagging errors is important for shell programming. For example, you have to return the proper exit status for "myprog && echo 'all is well'" to work properly. You can retrieve the exit status of the last command using "$?". For example:

```
#!/bin/bash

myprog -v sally
if [ $? -eq 0 ] ; then
echo "There were no errors"
else
echo "The program returned error code = $?"
fi
```

See the example program in the next section for an example using this package.

14.5 Linux Environment Variables

Ada.Command_Line.Environment is a gnat package for accessing Linux environment variables.

Ada	*Description*	*C Equivalent*
Function Environment_Count return natural;	The number of environment variables	?
Function Environment_Value(n) return string;	The name value of the nth variable	getenv(n)

The **Environment_Count** function returns the number of environment variables.

The **Environment_Value** function returns the name and value of a variable, separated by an equals sign. For example, Environment_Value(5) returns the name and value of the fifth environment variable.

The following program is an example of Ada.Command_Line and Ada.Command_Line.Environment. The results assume that you started the program by typing "cmdtest -v".

```
with text_io, Ada.Command_Line.Environment;

use text_io, Ada.Command_Line, Ada.Command_Line.Environment;
procedure cmdtest is

begin

   Put_Line( "This is an example of Ada.Command_Line" );
   New_Line;
   Put_Line( "The command to invoke this example was '" & Command_Name & "'" );
   Put_Line( "There is/are" & Argument_Count'img & " command line arguments" );
   if Argument_Count > 0 then
      Put_Line( "The first argument is '" & Argument(1) & "'" );
   end if;
   New_Line;
```

```
    Put_Line( "There is/are" & Environment_Count'img & " environment variables." );
    Put_Line( "The first environment variable is '" &
        Environment_Value( 1 ) & "'" );
    Set_Exit_Status( Success );
end cmdtest;
```

```
This is an example of Ada.Command_Line

The command to invoke this example was 'cmdtest'

There is/are 1 command line arguments
The first argument is '-v'

There is/are 24 environment variables.

The first environment variable is 'LESSOPEN=|lesspipe.sh %s'

Environment variables can be removed using the Gnat Ada.Command_Line.Remove package.
```

14.6 GNAT.Directory_Operations Package

This Gnat package allows you to create and explore directories. Although the package is portable to all operating systems, the format of the directory depends on the particular operating system.

For this package, a directory name string (Dir_Name_Str) is a pathname in the standard Linux format. The trailing '/' character is optional when using this package, but directory names returned will always have a trailing '/'. "." is the current directory. ".." is the parent directory of the current directory.

Get_Current_Dir returns the name of the current directory. **Change_Dir** changes the current directory to a new location.

```
with ada.text_io, gnat.directory_operations;
use ada.text_io, gnat.directory_operations;

procedure gdir is
    dir : string(1..80);
    len : natural;
begin
    Put( "The current working directory is " );
    Put_Line( Get_Current_Dir );

    Change_Dir( ".." );
    Put( "Moving up, the current working directory is " );
    Put_Line( Get_Current_Dir );

    Change_Dir( "work" );
    Get_Current_Dir( dir, len );
    Put( "Moving down to 'work', the current working directory is " );
    Put_Line( dir(1..len) );
end dir;
```

```
The current working directory is /home/kburtch/work/
```

```
Moving up, the current working directory is /home/kburtch/
Moving down to 'work', the current working directory is /home/kburtch/work/
```

For viewing directories, the package opens directories like a Text_IO file. Dir_Type is a limited private directory type corresponds to a file_type in Text_IO. Directories can only be read.

- **Open** - Open a directory for reading
- **Close** - Close a directory
- **Read** - Read a directory entry. When a null string is returned, there are no more entries
- **Is_Open** - True if the directory is open
- **Read_Is_Thread_Safe** - True if the directory can be read by separate tasks (threads). That is, if there is a readdir_r kernel call

Any error will raise a DIRECTORY_ERROR exception

```
with ada.text_io, gnat.directory_operations;
use ada.text_io, gnat.directory_operations;

procedure gdir2 is
   dir     : Dir_Type;
   dirname : string( 1..80 );
   len     : natural;
begin
   if Read_Is_Thread_Safe then
        put_line( "Tasks may read the same directory" );
   else
        put_line( "Tasks may not read the same directory" );
   end if;
   New_Line;

   Open( dir, "." );
   if Is_Open( dir ) then
        put_Line( "The directory was opened" );
   else
        put_Line( "The directory was not opened" );
   end if;
   loop
      Read( dir, dirname, len );
      exit when len = 0;
      Put_Line( dirname( 1..len ) );
   end loop;
   Put_Line( "End of directory" );

   Close( dir );

end gdir2;
```

```
Tasks may not read the same directory

The directory was opened
.
..
gdir.ads
gdir.ali
gdir.adb
gdir.o
gdir
gdir2.adb
```

```
gdir2.ali
gdir2.o
gdir2
End of directory
```

The directories "." and ".." are always returned.

New directories can be made with **Make_Dir**.

```
Make_Dir( "logs" ); -- make a new "logs" directory
```

If you need more features that these, the Linux kernel calls for directories are described in 16.9. The section includes a command to remove directories which cannot be done with Gnat.Directory_Operations.

14.7 GNAT.Lock_Files Package

This Gnat package contains subprograms for obtaining exclusive access to a particular file or directory. When a file is locked, only your program may use the file until the file is unlocked.

Locks are implemented using lock files. When a file is locked, Gnat checks for the presence of a separate file. If the file exists, the file has been locked by another application. If a file cannot be locked, a LOCK_ERROR is raised.

The programmer supplies the lock file name. Linux programs usually place lock files in the /var/lock/ directory.

The **Lock_File** procedure locks a particular file. By default, if the procedure will continue trying to relock the file every second forever (actually, for Natural'Last seconds, a very long time). The delay and the number of retries can be changed.

```
Lock_File( "/var/lock/", "customers.txt" );
Lock_File( "/var/lock/customers.txt" );
Lock_File( "/var/lock/customers.txt", Wait => 5.0, Retries => 10 );
```

Files are unlocked using **Unlock_File**. This procedure deletes the lock file.

```
Unlock_File( LockDir, "customers.txt" );
Unlock_File( "/var/lock/customers.txt" );
```

The lock file approach is a voluntary convention. Programs that honour the convention can share the file in an orderly way. A program that doesn't use the package will not be denied access. For true file locking, use the Linux kernel calls described in 16.7.

14.8 GNAT.Sockets

Incomplete

Note: The Gnat socket package is very simple and reportedly returns an error if a socket is blocked. If you need to connect to blockable sockets, you'll need to write your own O/S socket bindings. Simple Server with GNAT.Sockets. Since this uses Ada streams, you'll only be able to connect to this server with an Ada client using Ada streams.

```
with Ada.Text_IO;
with GNAT.Sockets;
use GNAT.Sockets;
use Ada.Text_IO;
```

```
procedure server is
    server_host : constant string := "localhost";
    server_port : constant Port_Type := 5876;

    server_address : Sock_Addr_type;
    server_socket  : Socket_Type;
    client_socket  : Socket_Type;
    client_stream  : Stream_Access;
begin
    server_address.addr := Addresses( Get_Host_By_Name( server_host ), 1 );
    server_address.port := server_port;
    Create_Socket( server_socket );
    Set_Socket_Option( server_socket, socket_level, (reuse_address, true ) );
    begin
        put_line( "Binding..." );
        Bind_Socket( server_socket, server_address );
    exception when Socket_Error =>
        put_line( standard_error, "Bind_Socket raised Socket_Error" );
    end;
    begin
        put_line( "Listening..." );
        Listen_Socket( server_socket );
    exception when Socket_Error =>
        put_line( standard_error, "Listen_Socket raised Socket_Error" );
    end;
    begin
        put_line( "Accepting..." );
        Accept_Socket( server_socket, client_socket, server_address );
    exception when Socket_Error =>
        put_line( standard_error, "Accept_Socket raised Socket_Error" );
    end;
    client_stream := Stream( client_socket );
    put_line( "Streamed..." );
    put_line( "Blocking..." );
    delay 0.2;
    put_line( "Reading channel.." );
    declare
       message : string := string'input( client_stream );
    begin
       put_line( "Message:" );
       put_line( "Message Addr: " & Image( Get_Address( client_stream ) ) );
       put_line( "Message Text: '" & message & "'" );
       put_line( "Message Len:  " & message'length'img );
    exception when END_ERROR =>
       put_line( "Premature end of data  error" );
    end;
    Close_Socket( server_socket );
    Close_Socket( client_socket );
end server;
```

15 Free Ada Bindings

15.1 Using Florist, the POSIX binding

Florist (Florida Statue University/Forest) is a GPL binding of the POSIX (IEEE Standard 1003.5b-1996) standard operating system functions. These include file operations, date and time functions, and multitasking—the same kinds of function provided by the standard C libraries and the Linux kernel.

If you are writing an application that will run on several different operating systems, Florist provides a level of operating system independence. Once you install gnat and Florist on your new platform, you should be able to recompile a Florist application without having to worry about variations in system calls.

Florist is designed for gnat and runs on Linux as well as Solaris, OFS1, AIX, IRIX and HP-UNIX. Florist works closely with the gnat run-time system: you must compile Florist against a particular gnat installation. If you change your gnat installation, you will need to recompile Florist.

 Older versions of Florist for Gnat 3.11 work best with a version of Gnat compiled for FSU threads. Native threads require some patching to work, and not all Florist features are supported--see the Florist documentation for details. For more information on FSU threads, read the multitasking section above.

Newer versions of Florist, such as the ALT RPM, works with the normal (native threads) version of Gnat.

Commercial support for Florist is available form ACT.

Because the Linux kernel largely adheres to the POSIX standard, many of Florist functions have the same parameters as their Linux counterparts.

Florist divides the POSIX functions into a set of 73 Ada packages, all prefixed with the name "posix". The main package is called "posix.ads" and contains the definition of data types and many of the basic POSIX functions.

To write Florist applications, you'll need to link in the Florist library with "-lposix" (check?) and, if necessary, use "-I" to indicate where you've installed the package specifications.

[Incomplete]

- Posix.File_Locking.Set_Lock - Lock.Lock = Unlock to unlock a file
- Posix.File_Locking.Wait_To_Set_Lock

15.2 Using Texttools

The Texttools packages are a GPL, ncurses-based library for the Linux console. Texttools contain more than 600 procedures and functions to create windows, draw scroll bars, handle the mouse and keyboard events, play sounds, and much more. The Texttools package also provides a thick binding to Linux kernel calls. You can create a wide variety of application programs using Texttools alone.

This is the same package used to implement TIA.

[to be rewritten—KB]

15.2.1 Installation

1. In the C_code directory, type "gcc -O -c *.c" to compile the C files.

2. The Ada files should compile when you build your project with Gnatmake. If TextTools are installed in a different directory than your project, you will need to use the gnatmake -I switch.

When linking, you'll need to include the "-lm" and "-lcurses" switches as well as the object files from C_code. TextTools uses the C math library and ncurses 4.0. For example,

```
gnatlink -lm -lncurses C_code/*.o ...
```

15.2.2 Introduction

Although there are over 600 procedures and functions in TextTools, to open window is fairly uncomplicated.

Everything in TextTools is drawn in a window. Everything in a window is a control (sometimes called a "widget"). To display a window, you must create a window, fill in the window with controls to display, and run the window manager's DoDialog command.

The following program opens a simple window.

```
with common, os, userio, controls, windows;
use common, os, userio, controls, windows;
procedure ttdemo is
-- Define Window Controls
   OKButton : aliased ASimpleButton;
   MessageLine : aliased AStaticLine;
-- The Dialog Record
   DT : ADialogTaskRecord;
begin
   -- Start TextTools
   StartupCommon( "demo", "demo" );
   StartupOS;
   StartupUserIO;
   StartupControls;
   StartupWindows;
   -- Create a new window. The window will not appear until the
   -- DoDialog procedure is used.
   OpenWindow( To255( "Demo Window" ), -- title at top of window
   0, 0, 78, 23, -- the coordinates of the window
   Style => normal, -- type of window, usually "normal"
   HasInfoBar => true ); -- true if control information is
   -- displayed at the bottom of the
   -- window
```

```
    -- Setup the controls in the window

    -- OK Button located near bottom of window

    Init( OKButton,

        36, 20, 44, 20, -- coordinates in window
        'o' ); -- hot key for OK button
    SetText( OKButton, "OK" ); -- button will have "OK"
    SetInfo( OKButton, To255( "Select me to quit" ) );
    AddControl( SimpleButton, OKButton'unchecked_access, IsGlobal => false );

    -- Message at top of window in bright red

    Init( MessageLine,

        1, 1, 78, 1 );
    SetText( MessageLine, "Welcome to TextTools" );
    SetStyle( MessageLine, Bold );
    SetColour( MessageLine, Red );
    AddControl( SimpleButton, MessageLine'unchecked_access, IsGlobal => false );

    -- Display the window and handle any input events. When dialog

    -- is finished, return control which completed the dialog.

    loop

        DoDialog( DT );
    exit when DT.Control = 1; -- first control is the OK button
    end loop;

    -- close the window

    CloseWindow;

    -- Shutdown TextTools

    ShutdownWindows;

    ShutdownControls;
    ShutdownUserIO;
    ShutdownOS;
    ShutdownCommon;
end ttdemo;
```

Package Overview

TextTools is broken into 5 main packages, based on what they do.

Common - this package contains all the basic data types used by TextTools, plus subprograms that work with those types. In particular, two important types are defined:

- **Str255** - most TextTools subprograms use this bounded, 255 character string type instead of the standard Ada fixed strings. The function To255 converts an Ada string to a Str255. ToString converts in the other direction.

- **Str255List** - some list controls display a block of text. These controls use the Str255List.List type, a linked list of Str255 strings. The subprograms for this type are defined the generic package gen_list.

Most TextTools calls do not return errors. There are some exceptions, such in the OS package. Error numbers are returned in the LastError variable. LastError is 0 if there is no error.

OS - this package contains subprograms for working with the Linux operating system: that is, for reading the current time, deleting files, and the like. Texttools pathnames are defined in this package. A path is a Str255 string. The OS package can define path prefixes, beginning with a "$". For example, "$HOME" is predefined as the user's home directory. To delete a file called "temp.txt" from the user's home directory, you can use the OS erase command:

```
Erase( To255( "$HOME/temp.txt" ) );
```

$SYS is another predefined prefix. This refers to a directory in the user's home directory named with the "short name" you specify in the StartupCommon procedure. Sounds, keyboard macros and the session_log file are located here.

UserIO - this package contains all the input/output routines for TextTools: it handles mouse clicks, draws text, and so forth. Normally, only people writing controls will need access to this package. However, the pen colours, beep sounds and text styles, are also defined here.

Controls - this package contains all the window controls and related subprograms. Currently defined controls are:

- Thermometer
- ScrollBar
- StaticLine
- EditLine (and family)
- CheckBox
- RadioButton
- WindowButton
- Rectangle
- Line
- HorizontalSep
- VerticalSep
- StaticList
- CheckList
- RadioList
- EditList
- SourceCodeList (used by TIA)

Windows - this is the window manager. It creates and draws windows, and DoDialog procedure lets a user interact with the window. It also handles the "Accessories" window that appears when ESC is pressed. The maximum window size is 180 by 180.

Each package is started with a "Startup" procedure, and shutdown with a "Shutdown" procedure. The only procedure to take parameters is StartupCommon: you need to specify a program name and a short name to use for temporary files.

15.2.4 Window Overview

The Window Manager draws all the windows on the screen. For simple programs, you will need to use only four Window Manager procedures.

OpenWindow - this procedure creates a new window. Each window has a title, coordinates on the screen, a "style", and an optional info bar.

AddControl - adds a control to the current window. If IsGlobal is false, the coordinates you specified in the control's Init call will be treated as relative to the top-left corner of the window, as opposed to the top left corner of the screen.

CloseWindow - closes the last window you created

DoDialog - this procedure displays the window and handles all interaction between the user and the window. It has one parameter, ADialogTaskRecord, which lets you set up callbacks (if necessary) and returns the number of the control which terminated the dialog.

5.2.5 Other Useful Window Manager Subprograms

Windows can be saved using the *SaveWindow* command, and loaded again using *LoadWindow*. When a window is loaded with LoadWindow, you don't need to open the window or set up the controls--the Window Manager does this automatically for you.

ShellOut will close the windows, run a shell command, and reopen the windows.

RefreshDesktop will redraw all the windows on the screen.

SetWindowTimeout will set a default control to be selected if there is no response after a certain amount of time.

15.2.6 Alerts

Alerts are small windows that show a short message.

NoteAlert - displays a message with an "OK" button. The status sound is played, if installed.

CautionAlert - displays a message with an "OK" button. The text is drawn to emphasize the message. The warning sound is played, if installed.

StopAlert - displays a message with an "OK" button. The text is drawn to emphasize the message. The warning sound is played, if installed.

YesAlert - display a message with "yes" (default) and "no" buttons. Plays an optional sound.

NoAlert - display a message with "yes" and "no" (default) buttons. Plays an optional sound.

CancelAlert - display a message with cancel button and a customized button (default). Plays an optional sound.

YesCancelAlert - display a message with "yes", "no", and "cancel" buttons and returns the number of the button selected. Plays an optional sound.

Example:

```
NoteAlert( "The database has been updated" );
```

15.2.7 Other Predefined Windows

SelectOpenFile - displays a dialog for opening files. It has parameter, ASelectOpenFileRec. You have to fill in certain before displaying this window.

SelectSaveFile - displays a dialog for saving files. It has one parameter, ASelectSaveFileRec. You have to fill in certain details before displaying this window.

ShowListInfo - displays a Str255List list in a window

EditListInfo - displays a Str255List list in a window and let's the user edit the list.

Example:

```
sof : ASelectOpenFileRec;

...
sof.prompt := To255( "Select a file to open" );
sof.direct := false; -- can't select directories
SelectOpenFile( sof );
if sof.replied then
    FilePath := sof.path & "/" & sof.fname;
else
    -- user cancelled
end if;
```

Control Overview

Every control must be initialized with the *Init* procedure. Init positions the control in the window and assigns a "hot key", a short cut key for moving to the control.

You can turn a control off (make it unselectable) using SetStatus. Setting the control's status to Standby will make it selectable. Some controls are automatically turned off, such as the static line control.

The following controls can be used in a TextTools window:

Thermometer

This is a thermometer bar graph. It shows the percentage between the maximum value and the current value, and is filled based on the percentage

ScrollBar

This is a scroll bar. A thumb is drawn at the relative location of the thumb value to the maximum value of the bar. The bar will be horizontal or vertical depending on the shape specified in the Init procedure.

StaticLine

This is an unchanging line of text.

EditLine (and family)

This is an editable line of text.

AdvanceMode - if set, the cursor will move to the next control when the edit field is full. This is useful in business applications where fixed-length product numbers are typed in.

BlindMode - if set, hides the characters typed. This is useful for typing in passwords.

SimpleButton

This is a button that, when selected, terminates the dialog.

Instant - if set, the button acts like a menu item. Pressing the hot key will immediately select the button and terminate the dialog. Otherwise, pressing the hot key only moves the cursor to the button.

CheckBox

A check box is an option which may be turned on or off.

RadioButton

A radio button is one of a set of options which may be turned on or off. Every radio button has a family number defined in the Init procedure. When a radio button is turned on, all other buttons in the family are turned off.

WindowButton

Loads a window from disk and displays it. The window must have been saved with the Window Manager's SaveWindow procedure.

Rectangle

A box which can be drawn around controls.

Line

A line--what else would it be--drawn between two corners of the enclosing rectangle defined by the Init procedure.

HorizontalSep

A horizontal line, often used to separate controls into groups.

VerticalSep

A vertical line, often used to separate controls into groups.

StaticList

A scrollable box of unchanging text.

CheckList

A scrollable box of check boxes.

RadioList

A scrollable box of radio buttons.

EditList

A scrollable box of editable text.

SourceCodeList (used by PegaSoft's TIA)

A scrollable box containing source code.

OS Package

This package contains various calls for working with the operating system. All calls support path prefixes as described above. Here are some of the subprograms:

- **UNIX** - run a UNIX shell command. The function variations return the result of the command.
- **RunIt** - runs a UNIX program.

- **ValidateFilename** - check for a syntactically correct filename.
- **NotEmpty** - true if a file is not empty
- **IsDirectory** - true if file is a directory
- **IsFile** - true if file is a "regular" file
- **MakeTempFileName** - creates a random file name for a temporary file
- **Erase** - deletes a file
- **LoadList** - load a Str255List list from a file
- **SaveList** - save a Str255List list to a file
- **MyID** - return the PID for your program
- **SessionLog** - write to the session log. If a $SYS directory exists, SessionLog creates a file called "session_log" in that directory. All SessionLog calls write to this file.

15.2.10 UserIO Overview

The UserIO package handles all the input and output for TextTools. Unless you are writing a game or new controls, you'll probably won't need to use UserIO at all. However, there are a few useful subprograms to be aware of:

- **Beep** - play a .wav file. Requires Warren Gay's wavplay program. These files must be saved in the $SYS directory, with the name of the beep sound in upper case.
- **Keypress** - get a keypress
- **DrawErr** - draw an error message. DrawErr draws the text on the left-side screen in white. Use only for emergencies.
- **GetDisplayInfo** - retrieve information about the current screen, such as whether it supports colour, and it's dimensions. Use this information to resize your windows for different screens.

Example:

```
Beep( Startup ); -- play startup sound
```

Keyboard Macros

UserIO will load a set of keyboard macros at startup. These must be saved in the $SYS directory, in a file called macro_file. The first letter of each line is the key for the macro, and the rest of the line is the expanded macro. For example, if a line in macro_file contained

pPegaSoft

then typing control-A followed by "p" would put the word "PegaSoft" in the input queue as if the person had typed "PegaSoft".

15.2.11 Appearance and Keys

Most of the objects on the screen should be easily understood, the majority designed after their GUI counterparts. Here is a list:

- < > Text - A button. Press Return to activate. Type the hilighted letter to go immediately to this button.

- | > Text - An menu button. Enter Return to activate. Type the hilighted letter to immediately activate.
- ( ) Text - A radio button. Press Return to select this item and deselect the previous item in the group.
- [ ] Text - A check box. Press Return to switch on or off.
- -----#------- - A scroll bar.
- -----50%----- - A thermometer graph.

Buttons with hyphens in them are not selectable.

Basic Keyboard Shortcuts:

Movement Keys

Up/Down Arrow - move up or down to the next menu item

* in lists - move up or down one line in the list
* in scroll bars - adjust up or down by 10%

Left/Right Arrows - move left or right to the next menu item

* in lists - move up or down one line in the list
* in scroll bars - adjust up or down by 1

Page Up (or Control-P) - move up one page in a list

* in scroll bars - same as up and down arrows

Page Down (or Control-N) - move down one page in a list

* in scroll bars - same as up and down arrows

Home Key (or Control-Y) - move to the top of a list

* in scroll bars - go to the top

End Key (or Control-E) - move to the bottom of a list

* in scroll bars - go to the bottom

Tab Key - move to the next item in the window

Control-T - move to the previous item in the window

Return Key (or Spacebar) - activate a button

When inside of a list box, the movement keys move you around the list. If you are on the Linux console, pressing alt and the hilighted letter will always jump to the appropriate object, even if you're inside a list box or the notepad.

Editing Keys

Control-6 - mark text

* only works in edit lists

Control-X - clear text

* in lists, clear the current line (or lines, if control-6 used)

Control-B - copy text

* in lists, copy the current line (or lines, if control-6 used)

Control-V - paste text

* in notepad, paste the last line copied

Misc. Keys

ESC Key (or F1) - bring up the accessories menu

Control-L - redraw the screen
Control-A (or F2) - execute a keyboard macro

15.3 Using NCurses

Ncurses is a free toolset for drawing on text screens, such as the Linux console.

[not finished]

15.4 Using GTK+ Widgets

GTK+, the Gimp ToolKit, is a widget set (or sometimes called a control set). These are the same widgets used by the Gimp drawing program. Unlike Motif and QT widgets, GTK+ uses an LGPL license, making it very popular for new Linux software, include the GNOME desktop project.

GTK+ also contains 2D drawing operations.

GTK+ is available for download from the GTK web site at http://www.gtk.org.

GtkAda can be downloaded from the ALT web site, or from its home page at http://ada.eu.org/gtkada/.

GtkAda is an Ada95 binding of Gtk+ version 1.2.0. It allows you to develop graphical applications in Ada95 using Gtk+. General GTK+ documentation and a tutorial written with examples in C are available from the GTK web site.

[not finished]

15.5 Using Motif Widgets

Motif (pronounced "Moe-Teef") is an X Windows widget standard created by the Open Software Foundation (OSF), a group of several UNIX companies. Motif is built for the stanadard X Windows library Xt. With Motif, you can create windows and dialog boxes menus, buttons, scrolling lists and the like. Motif is a registered trademark of OSF.

LessTif (pronounced "Less-Teef") is an open source compatible version of Motif 1.2 with some extensions, licenced under LGPL. It is available for download from the LessTif web site at http://www.lesstif.org. This site also includes documentation on compiling and installing LessTif.

There are no Ada bindings for LessTif, but there are Ada bindings for Motif which should work equally well for Lesstif. The bindings are available from the Home for Brave Ada Programmers, http://www.adahome.com.

Motif (and LessTif) have not proven to be very popular. Motif programs tend to be very large, with widgets layouts that are difficult to design, and have a heavy reliance on Motif's cumbersome resource files. Even small Motif programs typically require contain several hundred lines of source code to set up their initial window. Toolsets such as Qt (used in KDE) and GTK+ (used in GNOME) have larger followings, and Motif support is primarily for older applications being ported to Linux.

However, Motif, as a standard, is continuing to evolve.

15.6 Using the TCL Binding

TASH (Tcl Ada SHell) is a binding to TCL/TK. It includes both a thin binding to the basic TCL/TK functions (as found in the C header file tcl.h), as well as versions of the functions made for easier calling from Ada. The binding supports TCL 8.0

http://tash.calspan.com/

TASH also comes with its own TCL shell interpreter which functions like tclsh but is written in Ada.

15.7 Using OpenGL Binding

GLOBE_3D is an OpenGL system for Windows, Linux and Mac OS X. It features resource management and binary space partition. It can be downloaded from GLOBE_3D Home Page.

NiEstu is a basic OpenGL binding for Ada.

An older project, Mesa is an OpenGL library for 3D graphics. It can create 3D objects, transform them, and supports accelerated drivers.

You can use Mesa under GTK+ by using a GTK+ "GL Area" widget and draw graphics inside using Mesa.

15.8 Engine_3D

Engine_3D is a real-time 3D drawing package written entirely in Ada. It includes a Physics child package for reactions between 3D objects. The Linux port is by Duncan Sands. The Engine_3D package is at http://www.mysunrise.ch/users/gdm/e3d.htm.

15.9 Using the APQ Postgres Binding

APQ is a comprehensive binding to the Postgres database libpq library. APQ is very easy to use and offers complete access to Postgres, including full support for BLOB, DATE and TIME data types. It is a thick binding that uses tagged records, exceptions for errors and Ada streams for BLOB data.

APQ is open source and available for download at http://home.cogeco.ca/~ve3wwg.

15.9 GNU.PDF - a PDF package

PDF refers to the Portable Document Format. The PDF library enables one to generate PDF files or data streams on the fly. The binding is in the style of an OpenGL programming interface. If you know how to handle context scopes, etc. you should feel at home using it. The binding is small but allows for fonts, rotations, skewing, filling, URL links, and other features.

GNU.PDF is available at http://umn.edu/~puk.

This is a binding against the http://www.pdflib.com/pdflib library. The static library is named libpdf.a.

15.11 Gwindows (for Win32) - GUI, DB, ActiveX

Gwindows is a set of open source Ada packages for Microsoft Windows programming, database integration and ActiveX control support. It is a thick binding which takes advantage of Ada's features, ideal for developing Ada applications under Windows.

GWindows is available at http://www.adapower.com/gwindows.

15.12 GNATCOM and DirectX (for Win32)

DirectX (which as a COM library as are most new Win32 APIs) is availble using GNATCOM - http://www.adapower.com/gnatcom. It is compatible with GWindows.

15.13 Graph - plotting library

Graph is a os-independent package for drawing scientific graphs. It uses floating-point coordinates and vector fonts. The package is loosely based on Borland Pascal graph unit.

Graph is available at http://www.mysunrise.ch/users/gdm/graph.htm.

15.14 Using QtAda Widgets

QtAda, a binding that for creating Qt 4.2 and 4.3 (that is, KDE) applications in Ada. QtAda use native thread safe signal/slot mechanism, provide access to more than 120 Qt classes, provide Ada-aware meta object compiler, support development of custom widgets and Qt Designer's custom widget plugins, support loading at runtime of GUI forms from Qt Designer's UI files and so on.

It is available at http://sourceforge.net/projects/qtada/.

15.15 Using Cairo

A binding to Cairo, a 2D graphics library that can produce PDF and Postscript output as well as X Windows, etc., is available fromhttp://damien.carbonne.free.fr/cairoada/index.html. The Cairo wiki is at http://www.cairographics.org/.

16 Advanced Linux Programming

16.1 Writing Your Own Bindings

Ada Package	*Description*	*C Equivalent*
`pragma import`	Import identifier from another language	extern?
`pragma export`	Export identifier to another language	extern?
`pragma import_function`	Like import, but extra options	
`pragma import_procedure`	Like import, but extra options	
`pragma import_valued_procedure`	Import a function that returns values as parameters	extern?
`pragma export_function`	Like export, but extra options	
`pragma export_procedure`	Like export, but extra options	
`pragma unchecked_union`	(Ada 2005) Variant record is like a C union	

Because gnat is tightly integrated with gcc, we can make certain assumptions that would otherwise be impossible.

- the basic Ada data types are equivalent to their C counterparts: an Ada integer array is a C integer array
- **in** parameters are the same as pass by copy parameters in C
- **in out** parameters are the same as passing a pointer as a parameter in C
- Ada string *parameters* ending in ASCII.NUL are the same as a C string
- Ada procedures are the same as C void functions

There are rare cases when these assumptions don't hold (e.g. certain cases when null pointer parameters are not allowed by Ada), but, generally speaking, these assumptions are valid under Linux. Gnat has general purpose interfacing pragmas and support for C types in the Interfaces.C package. Use these if you want maximum portability.

Because of these assumptions, most C library calls are easily represented in Ada. For example, we check the man page for gettime and discover it returns the current time as a long integer. To call this from Ada, we use

```
   function gettime return long_integer;
pragma Import( C, gettime );
```

Since there is no Ada body for the gettime function, we use pragma import to let gnat know gettime is a C function. When we link, we need to specify the C library that function is in. In the case for the GNU C library, this is unnecessary since it's automatically linked. We can now call the C function gettime as if we wrote it ourselves.In C, it's possible to call a function and discard the result by not assigning it to anything. You can call C functions from Ada this way by declaring them a procedure. For example:

```
   procedure gettime;
pragma Import( C, gettime );
```

In this case, it's not particularly useful to call gettime and throw away the time it gives you. In general, you should avoid discarding the result because you may find it useful at some time in the future. However, there are certain C function where the result is provided only for flexibility, such as functions that return a pointer in a parameter and return the same pointer as the function result as

well. These can safely be discarded by treating the function as a procedure.If we wanted to export an integer variable called TotalTimeEstimate to C, we'd use

```
    TotalTimeEstimate : integer;
  pragma Export( C, TotalTimeEstimate );
```

A C function that returns void corresponds to an Ada procedure.

When importing or exporting to C, gnat converts the variable to lower case because C is a case-sensitive language. TotalTimeEstimate would be called totaltimeestimate in a C program. You can override this by providing a specific C name to link to. For example,

```
    pragma Export( C, TotalTimeEstimate, "TotalTimeEstimate" );
```

Import and Export don't require the name be the same at all. However, using entirely different names in C and Ada will make your program hard to understand.

If you want to import functions from libraries other than the standard C library, you will have to explicitly link them in. For example, to use the C math library, libm.a, would have to be explicitly linked using -lm. In C, functions can have parameters that change value, while in Ada this kind of function is not allowed because functions can only have "in" parameters. To get around this problem, gnat defines an import_valued_procedure pragma. Suppose you have a C function like this:

```
    int SomeCFunction( char * param )
```

Normally, there is no way to represent this kind of function in an Ada program. However, we can import it by treating it as a procedure using the import_valued_procedure pragma:

```
procedure SomeCFunction ( result : out integer; param : in out integer );
pragma import( C, SomeCFunction);
pragma import_valued_procedure( SomeCFunction );
```

The **import_valued_procedure** pragma tells gnat that this procedure corresponds to a C function: the first parameter is the result of the C function, and the remaining parameters correspond to the parameters of the C function. The first import pragma is not strictly required, but ACT recommends using it.

You can't import identifiers created by the **#define** statement since they only exist before a C program is compiled. You also can't import types (except for C++ classes) since types have no address in memory. [KB-true?]

There is one case where these tricks fail: when the C function returns a pointer to a C variable that it declared. In this case, the function is returning a new C pointer. Luckily, Ada provides a package called **Address_To_Access_Conversions** to convert between C pointers and Ada access types. You instantiate the package with the type you want to convert between, and the package creates an access type that can be converted to and from an address (which is a C pointer).The following program demonstrates conversions to and from C pointer types.

```
with Ada.Text_IO, System.Address_To_Access_Conversions;
use Ada.Text_IO;
procedure pointers is

package IntPtrs is
   new System.Address_To_Access_Conversions( integer );
   -- Instantiate a package to convert access types to/from addresses.
   -- This creates an integer access type called Object_Pointer.

   five : aliased integer := 5;
   -- Five is aliased because we will be using access types on it
```

```
   int_pointer : IntPtrs.Object_Pointer;
   -- This is an Ada access all type

   int_address : System.Address;
   -- This is an address in memory, a C pointer

begin
   int_pointer := five'unchecked_access;
   -- Unchecked_access needed because five is local to main program.
   -- If it was global, we could use 'access.

   int_address := five'address;
   -- Addresses can be found with the 'address attribute.
   -- This is the equivalent of a C pointer.

   int_pointer := IntPtrs.To_Pointer( int_address );
   int_address := IntPtrs.To_Address( int_pointer);
   -- Convert between Ada and C pointer types.

end pointers;
```

For example, the standard C library function get_current_dir_name returns a pointer to a C string which it declares. To use get_current_dir_name, we have to instantiate Address_To_Access_Conversions for an array of characters (a C string), and convert the address to an access type using something like

```
CharArray_pointer := CharArrayPtrs.To_Pointer( get_current_dir_name );
```

There is no other way in Ada to access the array that get_current_dir_name points to.

To import a C union, you'll have to use pragma unchecked_union to disable Ada features that are incompatible with C (namely, that the discriminant is not saved).

KB-If your main program is a C program, you need to call adainit before any Ada code.

16.2 Linux Errors and Errno

Errno contains a number for the error returned by the last library function.

Most standard C library errors are returned in an integer variable called "errno". You can examine errno in your Ada programs by importing it.

```
   errno : integer;
pragma import( C, errno);
```

 In Multithreading programs, be aware that an integer errno may not be not "thread safe" because it can be shared between threads.

In some C libraries you cannot import errno directly. For example, in modern versions of the GNU C library on Linux, errno is a C preprocessor macro for a function instead of an integer variable. A marco is used to make errno safe for mulitthreaded applications. However, Ada cannot import a macro because a macro has no storage space. The best solution I've found is to create C wrapper functions to work with errno for you.

```
   #include <errno.h>

   /* C_errno -- wrapper functions to errno variable */
```

```
int C_errno() {
        return errno;
}
void C_reset_errno() {
        errno = 0;
}
```

And then import these C functions into Ada using:

```
-- C wrapper functions of errno

function C_errno return integer;
pragma import( C, C_errno, "C_errno" );

procedure C_reset_errno;
pragma import( C, C_reset_errno, "C_reset_errno" );
```

Linux provides two functions for working with errno error numbers.

```
type string255is new string(1..255);
type strptr is access string255;
-- error messages are no longer than 255 characters
```

procedure perror(message : string);
pragma import(C, perror);
Perror prints a standard error description with a leading message to standard error.

function strerror(error_number : integer) **return** strptr;
pragma import(C, strerror);
Retuns a C string standard error description.

Thread-safe version of strerror called stderror_r.

The following example program makes a deliberate error with the link function and prints the error message using perror and stderror.

```
with ada.text_io, ada.strings.fixed;
use ada.text_io, ada.strings.fixed;
procedure perr is
-- an example of perror and strerror error messages

    procedure perror( message : string );
    pragma import( C, perror );
    -- print a standard error description with a leading message

    type string255 is new string(1..255);
    type strptr is access string255;
    -- error messages are no longer than 255 characters

    function strerror( error_number : integer )  return strptr;
    pragma import( C, strerror);
    -- get a standard error description

    errno : integer;
    pragma import( C, errno );
    -- last error number

    function link( path1, path2 : string ) return integer;
    pragma import( C, link);
```

```
   -- we'll use the link function to create an error

   LinkResult      : integer; -- value returned by link
   ErrorMessagePtr : strptr;  -- pointer to stderror message
   NullLocation    : integer; -- location of NUL in stderror message

begin

   Put_Line( "This is an example of perror and strerror");
   New_Line;

   -- make a deliberate error and print it with perror

   Put_Line( "Trying to link a non-existent file to itself.." );
   LinkResult := Link( "blahblah", "blahblah" );
   if LinkResult = -1 then
      perror( "Link failed" );
   end if;
   New_Line;

   -- Retrieve the last error message with strerror.
   -- Because strerror returns a C string, only print the
   -- string up to the first NUL character.

   ErrorMessagePtr := StrError( Errno );
   NullLocation := Index( string( ErrorMessagePtr.all ), "" & ASCII.NUL );
   Put( "The last error message was '" );
   Put( Head( string( ErrorMessagePtr.all ), NullLocation-1 ) );
   Put_Line( "'." );

end perr;
```

```
This is an example of perror and strerror
Trying to link a non-existent file to itself.
Link failed: No such file or directory
The last error message was 'No such file or directory'.
```

A table of error numbers is in the appendix.

16.3 The Linux Clock

The Ada.Calendar package is the standard method of working with time in Ada programs. If you need to interface with C programs, you may need to use Linux's time features.

The Linux clock functions are either kernel calls or are a part of the standard C library, and they don't need to be linked in with the -lc option.

16.3.1 Basic time functions

The basic Linux time functions work with the number of seconds since January 1, 1970. This is referred to as the *epoch* in the Linux man pages. Because of the limits of a long integer value, the Linux clock will stop working properly around the year 2038.

The basic functions use a long_integer for the time:

```
type time_t is new long_integer;

procedure time ( time : in out time_t);
pragma import( C, time );
```
Returns the current time.

```
function difftime( time1, time2 : time_t ) return long_float;
pragma import( C, difftime );
```
Returns the number of seconds between two times (as a long_float).

16.3.2 Timeval Calls - Microsecond Accuracy

The timeval kernel calls return (or set) the current time with microsecond accuracy using a timeval record.

```
type timeval is record
  tv_sec  : time_t; -- number of seconds (since epoch)
  tv_usec : long_integer; -- number of microseconds
end record;

type timezone is record
  tz_minuteswest : integer; -- minutes west of Greenwich
  tz_dsttime     : integer  -- unsupported in Linux
end record;

procedure gettimeofday( result : out integer; tv : in out timeval, tz : in out timezone );
pragma import(C, gettimeofday );
pragma import_valued_procedure( gettimeofday );
```
Get the current time as the number of microseconds since January 1, 1970. Returns 0 for success. ftime() is an obsolete version of this function

```
procedure settimeofday( result : out integer; tv : in out timeval; tz : in out timezone );
pragma import( C, settimeofday );
pragma import_valued_procedure( settimeofday );
```
Set the current time as the number of microseconds since January 1, 1970. Returns 0 for success.

```
procedure tzset;
pragma import( C, tzset );
```
Create the TZ environment variable, if it doesn't exist, and sets it to the current timezone as specified in /etc/localtime or /usr/lib/zoneinfo/localtime. This is automatically invoked by the standard C library time functions whenever necessary.

```
procedure adjtimex( result : out integer; buf : inout timex );
pragma import( C, adjtimex );
```
Tunes the kernel's clock for specific time parameters

16.3.3 Functions using the tm record

Besides the number of seconds elapsed since 1970, Linux can also work with records containing the time broken down into common measurements. These functions use a tm record. These functions are all a part of the standard C library.

```
type tm is record
    sec   : integer; -- seconds on the clock (0-59)
```

```
    min   : integer; -- minutes on the clock (0-59)
    hour  : integer; -- hour on the clock (0-23)
    mday  : integer; -- day of the month (1-31)
    mon   : integer; -- month (0-11)
    year  : integer; -- year
    wday  : integer; -- day of the week (0-6)
    yday  : integer; -- day of the year (0-365)
    isdst : integer; -- >0 is daylight savings time, 0=not, <0 unknown
end record;
```

You will also need the Address_To_Access_Conversions package to convert C pointers to tm record into Ada access type pointers.

```
package TmPtrs is
   new System.Address_To_Access_Conversions( tm );
```

function localtime(time : in out time_t) **return** system.address;
pragma import(C, localtime);
Change the time into a tm record, making changes for the current time zone. time is the C pointer to the seconds since 1970.

function gmtime(time : in out time_t) **return** system.address;
pragma import(C, gmtime);
Change the time into a tm record for UTC (Coordinated Universal Time). time is the C pointer to the seconds since 1970.

function mktime(tm : system.address) **return** time_t;
pragma import(C, mktime);
Convert a tm record into the seconds since 1970.

To get the current time in tm format,

```
   seconds_since_1970 : long_integer;
   tm_rec : tm;
   ...
   time( seconds_since_1970 );
   tm = TmPtrs.To_Pointer( localtime( seconds_since_1970'address ) ).all;
```

16.3.4 Time as a String

function asctime(tm : system.address) **return** string;
pragma import(C, asctime);
Convert the tm into a standard UNIX time C string, such as you see with the ls -l shell command.

function ctime(time : in out time_t) **return** long_integer;
pragma import(C, ctime);
Get the current time as a standard UNIX time C string. It's equivalent to using asctime() on the localtime() of the current time().

procedure strftime(result: size_t; datestr : **in out** *stringtype*; max : size_t; format : string;
tm : **in out** tmrec);

pragma import(C, strftime);
Like asctime(), converts a tm time into text. strftime() uses formatting codes to determine the appearance of the text, similar to the shell date command. Returns the length of the date string (including the ending ASCII.NUL). See the man page for complete details.

Example:
```
datestring : string(1..80);
...
statftime( datestringsize, datestring, datestring'size/8, "%I:%M" & ASCII.NUL, tm );
Ada.Text_IO.Put_Line( "The time is " & datestring( 1..datestringsize-1 ) );
```

16.3.5 Timer Functions

Timer functions use the timeval structure

```
function timerclear( tv : timeval );
function timerisset( tv : timeval );
function timercmp( t0, t1 : timeval; operator : ? );
```

16.4 Process Information

The Linux process functions are part of the standard C library, and do not need to be linked in with -lc.

function getpid **return** integer;
Returns the Process Identification Number (PID) for your program.

16.4.1 Ownership

The owner of a program is referred to as the UID (user identification number). Under Linux, there are actually three owners to any given program: the effective UID, the real UID and the saved UID. Normally, these three are all the same login. The real and saved uids are provided for programs that must temporarily pretend to be somebody else, like a daemon that needs special login for a short period of time, or setuid/setgid programs that must temporarily switch between owners. These special functions are not covered here.

function getuid **return** integer;
pragma import(C, getuid);
Get the (real) UID of a process.
Example: Put_Line("My UID is " & getuid'img);

function setuid (uid : integer) **return** integer;
pragma import(C, setuid);
Change the effective (and saved and real) UID of a process to a new owner.

The GID (group identification number) is the group the program belongs to. In Linux, there's a main, effective group number, and any number of secondary groups that a program can belong to. There is also real and saved GIDs, just like UIDs.

procedure getgroups(result : out integer; num : integer; *gidlist*);
pragma import(C, getgroups);

pragma import_valued_procedure(getgroups);
Return a list of group numbers that a process belongs to. *Gidlist* is the address of a list of C strings.

function getgid **return** integer;
pragma import(C, getgid);
Get the (real) UID of the process.
Example: Put_Line("My GID is " & getgid'img);

function setguid(gid : integer) **return** integer;
pragma import(C, setgid);
Change the effective GID (and saved and real) of a process to a new group.

Linux also allows you to arrange processes into groups for easier management of multiple processes at once. Each process group as a, no surprise, a process group identification number (PGID).

function setpgid(pid, pgid : integer) **return** integer;
pragma import(C, setpgid);
Place a process into a new process group. PID 0 is the current process. PGID 0 creates a new process group.

function getpgid(pid : intger) **return** integer;
pragma import(C, getpgid);
Example: Put_Line("My PGID is " & getpgid'img);
Returns the process group number for a process. PID 0 is the current process.

Every program and process group also belongs to a **session** (as in a login session). When you log off the computer, Linux automatically stops all programs that were running in your session. The **session leader** is the top process in a session, such as your login shell. However, if you want to create a new session for some reason, you can use the following function:

function setsid **return** integer;
pragma import(C, setsid);
Start a new session and return a new session identification number (SID).
Example: NewSID := etsid;

16.4.2 Other Functions

function kill(uid, signal : integer) **return** integer;
pragma import(C, kill);
Stop a child process that your process has started, the same as using the kill command at the shell prompt. (More accuately, send a signal to a child process—some signals won't stop the child process.)
Example: Result := kill(MyRunawayChildUID, 15); -- send SIGTERM (terminate) signal

Signal handling, in general, is easier through Ada.Interrupts than through Linux kernel calls because of their heavy reliance on C macros.--KB

function alarm(seconds : Interfaces.C.unsigned) **return** Interfaces.C.unsigned;
pragma import(C, alarm);
After the specified number of seconds, cause a SIGALRM interrupt in the current process.

16.5 Environment Variables

Environment variables can easily be set and read with Ada.Command_Line.Environment package. You can also set them directly through the standard C library.

function putenv(str : string) **return** integer;
pragma import(C, putenv);
Define a Linux environment variable. putenv literally saves a pointer to the string; therefore the string must be global (or a literal).
Example: Result := putenv("TERM=vt102" & ASCII.NUL);

function getenv(str : string) **return** string;
pragma import(C, putenv);
Read the value of an environment value. Remember the string returned is a C string with an ending ASCII.NUL.

16.6 Multitasking

Multitasking creates child processes to do tasks for your main program. On multiprocessor machines, different processes can be assigned to different processors allowing work to be done simultaneously. On single processor machines, the processor switches several times a second between processes and does a little work on each.

The function to create a new child process is called **fork**. When Linux creates a new process, it doesn't start by creating a blank process. Instead, it makes a copy of the original process so there are effectively two copies of your program running. The fork function returns a value to tells your program if it is the original program or the new copy.

If you want to run a different program, you'll have to use one of the **exec** family of functions to load and run the new program. The exec functions destroy the old program and run a new program in its place. The above section, Using System and OSLib.Spawn, has an example C function called CrunIt that uses fork to start a new process and run a new program.

type pid_t **is new** integer;
function fork **return** pid_t;
pragma import(C, fork);
Create a new child process identical to the original process and return 0 if the program is running as the child process or the PID of the parent if the program is running as the original parent process.
Example: myPID := fork;

procedure wait(pid : out pid_t; status : in out integer);
pragma import(C, wait);
pragma import_valued_procedure(wait);
Wait until a child process have finished running. Pid is the PID of the child. status is an integer code indicating whether the child finished normally, and if it was stopped by a signal, which signal terminated the program. Status can be a null pointer if you don't want status information.
Example: wait(wait_pid, wait_status);

procedure waitpid(pid : out pid_t, pid_or_gid : in out pid_t; status : in out integer; options : integer);
pragma import(C, waitpid);
pragma import_valued_procedure(waitpid);
Wait for a specific child. If pid_or_gid is less than -1, waitpid waits for any child in the specified group id. If pid_or_gid is -1, it waits for any child (the same as wait). If pid_or_gid is 0, it waits for

any child in the same group id as the parent. If pid_or_gid is greater than zero, waits for the child with the specified pid. Status can be a null pointer if you don't want status information. Options can determine whether waitpid returns immediately or blocks indefintely.
Example: waitpid(child_pid, -children_gid, wait_status, 0);

Wait3 and Wait4 are BSD UNIX variations which perform the same functions as wait and waitpid but with slightly different parameters.

When multitasking, if a child process stops, it's retained in memory so that the parent can use wait to find out the reason it stopped running. These children are called "zombies". If the parent process doesn't use wait, the zombies will remain indefinitely, using up system resources. For any large multitasking program, make sure you handle SIGCHLD signals: these are created when a child stops running. The SIGCHLD handler only needs to call wait and Linux will then remove the child process from memory.

The following is a simple multitasking example that multitasks two Put_Line statements.

```
-- a simple example of multitasking that multitasks
-- two put_line statements

with ada.text_io;
use ada.text_io;
procedure multitask  is
type pid_t is new integer;

   function fork return pid_t;
   pragma import( C, fork);
   -- create a new process

   errno : integer;
   pragma import( C, errno);
   -- the last error code

   procedure wait( pid : out pid_t; status : in out integer);
   pragma import( C, wait);
   pragma import_valued_procedure( wait);
   -- wait until all child processes are finished

   myPID : pid_t;
   wait_pid : pid_t;
   wait_status : integer;

begin

   Put_Line( "Welcome to this multitasking example" );
   Put_Line( "This is the original process." );
   New_Line;

   -- the fork function duplicates this program into
   -- two identical processes.

   Put_Line( "Splitting into two identical processes..." );
   Put_Line( "----------------------------------------" );
   myPID := fork; -- split in two!

   -- This program is now the original process or the
   -- new child process. myPID tells you which process
   -- you are.
```

```
    if myPID < 0 then
        Put_Line( Standard_Error, "Fork has failed.  Error code " & errno'img );
    elsif myPID = 0 then
        Put_Line( "This is the child process" );
    else
        Put_Line( "This is the original process." );
        -- wait until child is finished
        wait( wait_pid, wait_status);
        if wait_pid < 0  then
            Put_Line( Standard_Error, "Wait error: wait returned PID " & wait_pid'img
                & " and error number " & errno'img);
        end if;
    end if;

end multitask;
```

```
Welcome to this multitasking example
This is the original process.
Splitting into two identical processes...
----------------------------------------
This is the original process.
This is the child process
```

16.7 Linux File Operations

The Linux file operations are part of the standard C library, and don't need to be linked in with the -lc option. The C calls are defined in the "fcntl.h" header file.

[Explain Linux files here]


Linux never shortens files. If your file gets smaller, you must shorten it yourself using `truncate`.

The following bindings assume these types have been defined.

```
type file_id is new integer;
-- file ID number are discussed below

type mode_t is new integer;
type gid_t is new integer;
type uid_t is new integer;
type size_t is new long_integer;
```

function unlink(pathname : string) **return** integer;
pragma import(C, unlink);
Delete a file.
Example: Result := unlink("/tmp/temp.txt" & ASCII.NUL);

function link(oldpath, newpath : string) **return** integer;
pragma import(C, link);
Make a shortcut (hard link) to a file.

Example: Result := link("/tmp/temp.txt" & ASCII.NUL, "/tmp/newtemp.txt" & ASCII.NUL);

procedure getcwd(buf1 : out StringPtr; buf2 : in out stringptr; size : integer);
pragma import(C, getcwd);
pragma import_valued_procedure(getcwd)
Return the current working directory.

function mkdir(pathname : stringPtr; mode : mode_t) **return** integer;
pragma import(C, mkdir);
Create a new directory and set default permissions.

function rmdir(pathname : string) **return** integer;
pragma import(C, rmdir);
Delete a directory.
Example: Result := rmdir("/tmp/tempdir" & ASCII.NUL);

function umask(mask : integer) **return** integer;
pragma import(c, umask);
Sets the default file permissions.

function stat(filename : stringPtr; buf : stat_struct) **return** integer;
pragma import(C, stat);
Get information about a file, such as size and when it was last opened.

function lstat(filename : stringPtr; buf : stat_struct) **return** integer
pragma import(C, lstat);
Same as stat function, but doesn't follow symbolic links.

function tmpnam(s : stringPtr) **return** stringPtr;
pragma import(C, tmpnam);
Create a random name for a temporary file.

function chown(path : string; owner : uid_t; group : gid_t) **return** integer;
pragma import(C, chown);
function fchown(file : file_id; owner : uid_t; group : gid_t **return** integer;
pragma import(C, fchown);
Change the ownership of a file to the specified owner and group.
Example: Result := chown("root.txt" & ASCII.NUL, 0, 0);

function chmod(path : string; mode : mode_t) **return** integer;
pragma import(C, chmod);
function fchmod(file : file_id; mode : mode_t) **return** integer;
pragma import(C, fchmod);
Change the read/write/execute permissions on a file.
Example: Result := chmod("secure.txt" & ASCII.NUL, #8#640);

Other low-level file operations are all done with the **fcntl** (file control) function. There are three variations to fcntl: it may have an operation code, an operation code and a long integer argument, or an operation code and a locking record argument.

The operation numbers are defined in /usr/src/linux/asm-i386/fnctl.h:

```
F_DUPFD  : constant integer :=  0;
F_GETFD  : constant integer :=  1;
```

```
F_SETFD  : constant integer :=  2;
F_GETFL  : constant integer :=  3;
F_SETFL  : constant integer :=  4;
F_GETLK  : constant integer :=  5;
F_SETLK  : constant integer :=  6;
F_SETLKW : constant integer :=  7;
F_SETOWN : constant integer :=  8;
F_GETOWN : constant integer :=  9;
F_SETSIG : constant integer := 10;
F_GETSIG : constant integer := 11;
```

function fcntl(fd : file_id; operation => F_DUPFD)
pragma import(C, fcntl);
Duplicates a file descriptor (same as dup2, but different errors returned). New descriptor shares everything except close-on-exec. New descriptor is returned.

function fcntl(fd : file_id; operation => F_GETFD)
pragma import(C, fcntl);
Get close-on-exec flag; low bit is zero, file will close on exec kernel call.

function fcntl(fd : file_id; operation => F_SETFD; arg : long_integer)
pragma import(C, fcntl);
Set the close-on-exec flag; low bit is 1 to make file close on exec kernel call.

function fcntl(fd : file_id; operation => F_GETFL)
pragma import(C, fcntl);
Get flags used on open kernel call used to open the file

function fcntl(fd : file_id; operation => F_SETFL; arg : long_integer)
pragma import(C, fcntl);
Set flags for open kernel call. Only async, nonblock and appending can be changed.

procedure fcntl(result : out integer; fd : file_id; operation => F_GETLK; lock : in out lockstruct)
return integer
pragma import(C, fcntl);
pragma import_valued_procedure(fcntl);
Return a copy of the lock that prevents the program from accessing the file, or else if there is nothing blocking, the type of lock

procedure fcntl(result : out integer; fd : file_id; operation => F_SETLK; lock : in out lockstruct)
return integer
pragma import(C, fcntl);
pragma import_valued_procedure(fcntl);
Place a lock on the file. If someone else has locked the file already, -1 is returned and errno contains the locking error.

procedure fcntl(result : out integer; fd : file_id; operation => F_SETLKW; lock : in out lockstruct)
return integer
pragma import(C, fcntl);
pragma import_valued_procedure(fcntl);
Place a read or write lock on the file, or to unlock it. If someone else has locked the file already, wait until the lock can be placed.

Additional information about locks are found in /usr/src/linux/Documentation/locks.txt

```
type aLock is new short_integer;
F_RDLCK : constant aLock := 0; -- read lock
F_WRLCK : constant aLock := 1; -- write lock
F_UNLCK : constant aLock := 2; -- unlock (remove a lock)
F_EXLCK : constant aLock := 3; -- exclusive lock
F_SHLCK : constant aLock := 4; -- shared lock

type aWhenceMode is new short_integer;
SEEK_SET : constant aWhenceMode := 0; -- absolute position
SEEK_CUR : constant aWhenceMode := 1; -- offset from current position
SEEK_END : constant aWhenceMode := 2; -- offset from end of file


type lockstruct is record
    l_type   : aLock;         -- type of lock
    l_whence : short_integer; -- how to interpret l_start
    l_start  : integer;       -- offset or position
    l_len    : integer;       -- number of bytes to lock (0 for all)
    l_pid    : integer;       -- with GETLK, process ID owning lock
end record;
```

To lock a file, create a lockstruct record and fill in the details about the kind of lock you want.

A read lock (F_RDLCK) makes the part of the file you specify read-only. No one can write to that part of the file.

A write lock prevents any other program from reading or writing to the part of the file you specify. Your program may change that part of the file without being concerned that another process will try to read it before you're finished.

If your program stops prematurely, the locks will be released.

Example: Get exclusive right to write to the file, waiting until it's possible:

```
    -- lock file
    myLockStruct : lockStruct;
    result : integer;
    ...
    myLockStruct.l_type := F_WRLCK;
    myLockStruct.l_whence := 0;
    myLockStruct.l_start := 0;
    myLockStruct.l_end := 0;
    fcntl( result, fd, F_SETLKW, myLockStruct );
    if result = -1 then
        put_line( standard_error, "fcntl failed" );
    end if;
    -- file is now locked
    ...
    -- unlock file
    myLockStruct.l_type := F_UNLCK;
    myLockStruct.l_whence := 0;
    fcntl( result, fd, F_SETLKW, myLockStruct );
    if result = -1 then
        put_line( standard_error, "fcntl failed" );
    end if;
```

[Double check off_t size for l_start, l_len--KB]

```
function fcntl( fd : file_id; operation => F_GETOWN )
pragma import( C, fcntl );
```

Get the process (or process group) id of owner of file. The owner is the process that handles SIGIO and SIGURG signals for that file.

function fcntl(fd : file_id; operation => F_SETOWN, arg : long_integer)
pragma import(C, fcntl);
Set the process (or process group) id of owner of file. The owner is the process that handles SIGIO and SIGURG signals for that file. This affects async files and sockets.

function fcntl(fd : file_id; operation => F_GETSIG)
pragma import(C, fcntl);
Get the signal number of the signal sent when input or output becomes possible on a file (usually SIGIO or zero). (This is a Linux-specific function.)

function fcntl(fd : file_id; operation => F_SETSIG, arg : long_integer)
pragma import(C, fcntl);
Set the signal number of the signal sent when input or output becomes possible on a file (zero being the default SIGIO). Use this to set up a signal handler alternative to the kernel calls select and poll. See the man page for more information. (This is a Linux-specifc function.)

16.8 Opening and Closing Files

The standard Ada packages Text_IO, Sequential_IO and Direct_IO are suitable for simple projects, but they were never intended as a complete solution for large-scale applications. If you want to do efficient file manipulation, you'll have to write your own routines based on kernel calls or the standard C library.

gnat's OSLIB package contains low-level commands to work with UNIX files. However, you can always create your own.

The following bindings assume these types have been defined.

```
type file_id is new integer;
type mode_t is new integer;
type off_t is new long_integer;
type size_t is new long_integer;
subype ssize_t is size_t;
```

function open(path : string; flags : integer; mode : mode_t) **return** file_id;
pragma import(c, open);
Open a file and return and file identification number. flags indicates how the file should be opened and what kind of access the file should allow (defined in /usr/include/fcntlbits.h). Mode defines the access permissions you want on the file.

The flags are a set of bits with different meanings:

```
O_RDONLY   : constant integer :=      8#00#; -- open for reading only
O_WRONLY   : constant integer :=      8#01#; -- open for writing only
O_RDWR     : constant integer :=     8#02#; -- open for reading and writing
O_CREAT    : constant integer :=    8#0100#; -- no file? create it
O_EXCL     : constant integer :=    8#0200#; -- lock file (see below)
O_NOCTTY   : constant integer :=    8#0400#; -- if tty, don't acquire it
O_TRUNC    : constant integer :=   8#01000#; -- file exists? truncate it
O_APPEND   : constant integer :=   8#02000#; -- file exists? move to end
```

```
O_NONBLOCK : constant integer :=   8#04000#; -- if pipe, don't wait for data
O_SYNC     : constant integer :=  8#010000#; -- don't cache writes
O_ASYNC    : constant integer :=  8#020000#; -- async. IO via SIGIO
O_DIRECT   : constant integer :=  8#040000#; -- direct disk access
O_LARGEFILE: constant integer := 8#0100000#; -- not implemented in Linux (yet)
O_DIRECTORY: constant integer := 8#0200000#; -- error if file isn't a dir
O_NOFOLLOW : constant integer := 8#0400000#; -- if sym link, open link itself
```

Flags may be added together.
O_EXCL is somewhat obsolete and has limitations on certain file systems. Use fcntl to lock files instead.
O_SYNC only works on the ext2 file system or on block devices.

function creat(path : string, mode : mode_t) **return** file_id;
pragma import(c, creat);
Creat is a short form for open(path, create + writeonly + truncate, mode)

function close(file : file_id) **return** integer;
pragma import(C, close);
Closes a file.

function truncate(path : string; length : size_t) **return** integer;
pragma import(C, truncate);
function ftruncate(file : file_id; length : size_t) **return** integer;
pragma import(C, ftruncate);
Shorten a file to a specific length. Despite its name, ftruncate is a kernel call, not a standard C library call like fopen.

function read(file : file_id; b : **in out** *buffer*; length : size_t) **return** ssize_t;
pragma import(C, read);
Read bytes from the specified file into a buffer. *Buffer* is any type of destination for the bytes read, with length being the size of the buffer in bytes. The number of bytes read is returned, or -1 on an error.

function write(file : file_id; b : **in out** *buffer*; length : size_t) **return** ssize_t;
pragma import(C, write);
Write bytes from a buffer into the specified file. *Buffer* is any type of destination for the bytes read, with length being the size of the buffer in bytes. The number of bytes written is returned, or -1 on an error.

function lseek(file : file_id; offset : off_t; whence : integer) **return** integer;
pragma import(C, lseek);
Move to a particular position in the specified file. Whence is a code representing where your starting position is. Offset is how many bytes to move.

There are three possible "whence" values:

```
SEEK_SET : constant integer := 0; -- from start of file
SEEK_CUR : constant integer := 1; -- offset from current position
SEEK_END : constant integer := 2; -- from end of file
```

File input/output is naturally suited to generic packages. You can use the generic package to hide the low-level details of the standard C library. In following example, SeqIO is a generic package for reading and writing a sequential file of some type, using the kernel calls mentioned above.

```
-- SeqIO
--
-- A simple sequential IO package using standard C functions

generic
    type AFileElement is private;

package SeqIO is

    type AFileID is new short_integer;
    seqio_error : exception;

    function Open( path : string; read : boolean := true ) return AFileID;
    -- open a new file for read or write

    procedure Close( fid : AFileID );
    -- close a file

    procedure Read( fid : AFileID; data : in out AFileElement);
    -- read one data item from the file. seqio_error is raised
    -- if the data couldn't be read

    procedure Write( fid : AFileID; data : AFileElement );
    -- write one data item to the file.  seqio_error is raised
    -- if the data couldn't be written

end SeqIO;

package body SeqIO is
pragma optimize( space);

    -- Import C file handling functions
    type mode_t is new integer; -- C mode_t type
    type size_t is new integer; -- C size_t type
    subtype ssize_t  is size_t; -- C ssize_t type

    -- The C file functions we'll be using
    -- (denoted with a C_ prefix for clarity )

    function C_Open( path : string; flags : integer; mode : mode_t)
        return AFileID;
    pragma import( C, C_Open, "open");

    function C_Close( file : AFileID )  return integer;
    pragma import( C, C_Close, "close" );

    procedure C_Read( size : out ssize_t;
        file : AFileID;
        data : in out AFileElement;
        count: size_t);
    pragma import( C, C_Read, "read");
    pragma import_valued_procedure( C_Read);
    -- Using an "in out" parameter is the easiest way to pass
    -- the address of the data element. Because Ada doesn't
    -- allow in out parameters in functions, we'll use gnat's
    -- valued procedure pragma to pretend read is a procedure

    procedure C_Write( size : out ssize_t;
        file : AFileID;
        data : in out AFileElement;
```

```
        count: size_t);
    pragma import( C, C_Write, "write");
    pragma import_valued_procedure( C_Write);
    -- Using an "in out" parameter is the easiest way to pass
    -- the address of the data element. Because Ada doesn't
    -- allow in out parameters in functions, we'll use gnat's
    -- valued procedure pragma to pretend write is a procedure
    -- Our Ada subprograms

function Open( path : string; read : boolean := true ) return AFileID  is
    -- open a new file for read or write
    flags : integer;
begin

    -- the flag values are listed in fcntlbits.h and man 2 open
    if read then
        flags := 0; -- read only, existing file
    else
        flags := 1000 + 100 + 1; -- write only, create or truncate
    end if;
    -- octal 640 => usr=read/write, group=read, others=no access

    return C_Open( path & ASCII.NUL, flags, 8#640# );
end Open;

procedure Close( fid : AFileID ) is
    -- close a file
    Result : integer; -- we'll ignore it
begin
    Result := C_Close( fid);
end Close;

procedure Read( fid : AFileID; data : in out AFileElement ) is
    -- read one data item from the file
    BytesRead : ssize_t;
begin
    -- 'size returns the size of the type in bits, so we
    -- divide by 8 for number of bytes to read
    C_Read( BytesRead, fid, data, AFileElement'size / 8 );
    if BytesRead /= AFileElement'size / 8  then
        raise seqio_error;
    end if;
end Read;

procedure Write( fid : AFileID; data : AFileElement ) is
    -- write one data item to the file
    BytesWritten : ssize_t;
    data2write : AFileElement;
begin
    -- can't use data directly because it's an "in" parameter
    data2write := data;
    -- 'size returns the size of the type in bits, so we
    -- divide by 8 for number of bytes to write
    C_Write( BytesWritten, fid, data2write, AFileElement'size / 8);
    if BytesWritten /= AFileElement'size / 8 then
        raise seqio_error;
    end if;
end Write;

end SeqIO;
```

You can test SeqIO with the following program:

```
with SeqIO;
with Ada.Text_IO;
use Ada.Text_IO;

procedure SeqIOtest is
-- program to test SeqIO

package IntIO is new SeqIO( integer);
-- IntIO is a SeqIO for integer numbers

   id : IntIO.AFileID;
   int2read : integer;

begin

   Put_Line( "Testing SeqIO package..." );
   New_Line;

   -- Part #1: Write numbers to a file

   Put_Line( "Writing numbers 1 to 10 to a file...");
   id := IntIO.Open( "int_list.txt", read => false );
   for i in 1..10 loop
      IntIO.Write( id, i)
   end loop;
   IntIO.Close( id);

   -- Part #2: Read the numbers back from the same file

   Put_Line( "Reading numbers back..." );
   id := IntIO.Open( "int_list.txt", read => true);
   for i in 1..10  loop
      IntIO.Read( id, int2read);
      Put_Line( "Number" & i'img & " =" & int2read'img );
   end loop;
   IntIO.Close( id );

exception when IntIO.seqio_error =>
   Put_Line( "Oh, oh! seqio_error!");
end SeqIOtest;
```

Note: This should be rewritten because a failure to write all the bytes is not necessarily an error--Linux has a buffer limit on how much it writes at one time--KB

```
Writing numbers 1 to 10 to a file...
Reading numbers back...
Number 1 = 1
Number 2 = 2
Number 3 = 3
Number 4 = 4
Number 5 = 5
Number 6 = 6
Number 7 = 7
Number 8 = 8
Number 9 = 9
Number 10 = 10
```

File Multiplexing Operations

These kernel calls help programs that have to monitor several file descriptors at once for activity.

procedure select(result : **out** integer; topplusone : integer; readset : **in out** fdset; writeset : **in out** fd_set; errorset : **in out** fd_set; timeout : **in out** timeval);
pragma import(C, select);
pragma import_valued_procedure(select);
Select checks one or more file descriptors to see if they are ready for reading, writing, or if there is an error. It will wait up to timeout microseconds before timing out (0 wll return immediately). topplusone is the numerically highest file descriptor to wait on, plue one. The result is 0 for a timeout, -1 for failure, or the number of file discriptors that are ready and the file discriptor sets indicate which ones.
Unlike most UNIX's, Linux leaves the time remaining in the timeout record so that you can use select in a timing loop--to repeatedly select file descriptors until the timeout counts down to zero. Other UNIX's leave the timeout unchanged.

```
type pollfd is record
    fd : integer;
    events : short_integer;
    revents : short_integer;
end record;
```

Poll Events
POLLIN := 16#1#;
POLLPRI := 16#2#;
POLLOUT := 16#4#;
POLLERR := 16#8#;
POLLHUP := 16#10#;
POLLNVAL := 16#20#;

These are defined in asm/poll.h.

procedure poll(result : **out** integer; ufds : **in out** pollfd; nfds : integer; timeout_milliseconds : integer);
pragma import(C, poll);
pragma import_valued_procedure(poll);
The name of this kernel call is misleading: poll is a form of select(). timeout_milliseconds is a timeout in milliseconds, -1 for no timeout. ufds is an array of pollfd records for files that poll() should monitor. Poll returns the number of pollfd array elements that have something to report, 0 in a timeout, or -1 for an error. Each bit in events, when set, indicates a particular event that the program is waiting for. Revents represents the events which occurred.

16.9 Directories

Directories are "folders" containing collections of files and other directories. In Linux, a directory is a special kind of file. Some of the standard file operations work on directories and some other file operations are specific to directories.

The top-most directory is /, or the root directory. All files on a system are located under the root directory. Disk drives do not have separate designations as in MS-DOS.

A period (.) represents the current directory, and a double period (..) represents the parent directory of the current directory. All directories have . and .. defined. The root directory, of course, doesn't have a parent: it's .. entry points to itself.

Many of the kernel calls and standard C library functions dealing with directories use functions that return C pointers. As mentioned in the bindings section, the only way to convert these kind of functions to Ada is by declaring the C pointers as a System.Address type and changing the C pointers to Ada access types using the Address_To_Access_Conversions package.

procedure getcwd(buffer : out string; maxsize : size_t);
pragma import(C, getcwd);
Returns the name of the current working directory as a C string in buffer. Maxsize is the size of the buffer. All symbolic links are dereferenced.

function get_current_dir_name **return** System.Address;
pragma import(C, get_current_dir_name);
Like getcwd, returns the current working directory name as a pointer to a C string. Unlike getcwd, symbolic links aren't dereferenced. Use this function to show the current directory to a user.

procedure chdir(path : string);
pragma import(C, chdir);
Change the current working directory to the specified path.
Example: chdir("/home/bob/stuff" & ASCII.NUL);

function mkdir(path : string; mode : size_t) **return** integer;
pragma import(C, mkdir);
Create a new directory with permission bits as specified by mode.

function rmdir(path : string) **return** integer;
pragma import(C, rmdir);
Remove a directory.

function opendir(path : string) **return** System.Address; **pragma** import(C, opendir);
Open a directory in order to read its contents with readdir.

function closedir(info : System.Address) **return** integer;
pragma import(C, closedir);
Close a directory openned with opendir. Info is the C pointer returned by opendir.

function readdir(info : System.Address) **return** DirEntCPtr;
pragma import(C, readdir);
Read the next entry in the directory. A null C pointer is returned if there is no more entries. Info is the C pointer returned by opendir.

function rewinddir(info : System.Address) **return** integer;
pragma import(C, rewinddir);
Begin reading from the top of the directory. Info is the C pointer returned by opendir.

function telldir(info : System.Address) **return** integer;
pragma import(C, telldir);
Mark the current position in the directory, to return to it later using the seekdir function. Info is the C pointer returned by opendir.

function seekdir(info : System.Address; position : integer) **return** integer;
pragma import(C, seekdir);
Return to a position in the directory marked by telldir. Info is the C pointer returned by opendir.

function chroot(newroot : string) **return** int;
pragma import(C, chroot);
Make Linux think that a different directory is the root directory (for your program). This is used by programs such as FTP servers to prevent uses from trying to access files outside of a designated FTP directory.
Example: Result := chroot("/home/server/stay-in-this-directory" & ASCII.NUL);

There is also a **scandir** function that reads a directory and sorts the entries, but this is difficult to use directly from Ada.

The following program demonstrates some of the directory subprograms in Linux.

```
with Ada.Text_IO, Interfaces.C, Ada.Strings.Fixed;
use Ada.Text_IO, Interfaces.C, Ada.Strings.Fixed;
with System.Address_To_Access_Conversions;

procedure direct is

      -- Working with directories

   subtype size_t is Interfaces.C.size_t;
   -- renaming size_t to save some typing

   package CStringPtrs is  new
      System.Address_To_Access_Conversions( string );
   use CStringPtrs;
-- Convert between C and Ada pointers to a string

   subtype DirInfoCPtr  is System.Address;
   subtype DirEntCPtr is System.Address;
   -- two C pointers (System.Address types), renamed for
   -- clarity below

   type DirEnt is record
      inode : long_integer; -- inode number
      offset : integer; -- system dependent
      offset2: unsigned_char; -- system dependent
      reclen : unsigned_short; -- system dependent
      name : string( 1..257 ); -- name of file
   end record;
   pragma pack( dirent);
   -- dirent is defined in /usr/src/linux../linux/dirent.h

   package DirEntPtrs is new
      System.Address_To_Access_Conversions( DirEnt );
   use DirEntPtrs;
      -- Convert between C and Ada pointers to a directory entry

   procedure getcwd( buffer : out string; maxsize : size_t );
   pragma import( C, getcwd );

   function get_current_dir_name return System.Address;
   pragma import( C, get_current_dir_name);

   function mkdir( path : string; mode : size_t ) return integer;
   pragma import( C, mkdir );

   function rmdir( path : string ) return integer;
   pragma import( C, rmdir );
```

```
    function opendir( path : string ) return DirInfoCPtr;
    pragma import( C, opendir );

    function closedir( info : DirInfoCPtr ) return integer;
    pragma import( C, closedir );

    function readdir( info : DirInfoCPtr ) return DirEntCPtr;
    pragma import( C, readdir );

    function rewinddir( info : DirInfoCPtr ) return integer;
    pragma import( C, rewinddir );

    function telldir( info : DirInfoCPtr ) return integer;
    pragma import( C, telldir );

    function seekdir( info : DirInfoCPtr; position : integer ) return integer;
    pragma import( C, seekdir );

    -- scandir hard to use from Ada

    s: string(1..80);
    csop: CStringPtrs.Object_Pointer;
    Result: integer;
    DirInfo: DirInfoCPtr;
    direntop : DirEntPtrs.Object_Pointer;
    Position : integer;
    LastPosition : integer;

begin

    Put_Line( "This program demonstrates Linux's directory functions" );
    New_Line;

    -- getcwd example

    getcwd( s, s'length );
    Put( "The current path (simplified) is " );
    Put_Line( Head( s, Index( s, ASCII.NUL & "" )-1 ));

    -- Index for fixed strings takes a string as the second parameter
    -- We'll make a string containing an ASCII.NUL with &

    -- get_current_dir_name example

    csop := To_Pointer( get_current_dir_name );
    Put( "The current path is " );
    Put_Line( Head( csop.all, Index( csop.all, ASCII.NUL & "" )-1 ) );

    -- mkdir example: create a directory named "temp"

    Result := mkdir( "temp" & ASCII.NUL, 755 );
    if Result /= 0 then
        Put_Line( "mkdir error" );
    else
        Put_Line( "temp directory created" );
    end if;

    -- rmdir example: remove the directory we just made

    Result := rmdir( "temp" & ASCII.NUL );
```

```
    if Result /= 0 then
        Put_Line( "rmdir error" );
    else
        Put_Line( "temp directory removed" );
    end if;
    New_Line;

    -- directory reading

    DirInfo := OpenDir( "/home/ken/ada" & ASCII.NUL);
    Put_Line( "Directory /home/ken/ada contains these files:");
    loop
        direntop := To_Pointer( ReadDir( DirInfo ) );
    exit when direntop = null;

        -- TellDir returns the position in the directory
        -- LastPosition will hold the position of the last entry read

        LastPosition := Position;
        Position := TellDir( DirInfo );
        Put_Line( Head( Direntop.name, Index( Direntop.name, ASCII.NUL & "" )-1 ) );
    end loop;
    New_Line;

    -- SeekDir: move to last position in directory
    Result := SeekDir( DirInfo, LastPosition );
    Put( "The last position is " );
    direntop := To_Pointer( ReadDir( DirInfo ) );
    Put_Line( Head( Direntop.name, Index( Direntop.name, ASCII.NUL & "" )-1 ) );
    New_Line;

    -- RewindDir: Start reading again

    Result := RewindDir( DirInfo );
    Put( "The first position is " );
    direntop := To_Pointer( ReadDir( DirInfo ) );
    Put_Line( Head( Direntop.name, Index( Direntop.name, ASCII.NUL & "" )-1 ) );
    New_Line;

    -- close the directory

    Result := CloseDir( DirInfo );
end direct;
```

```
This program demonstrates Linux's directory functions
The current path (simplified) is /home/ken/ada/trials
The current path is /home/ken/ada/trial
temp directory created
temp directory removed
Directory /home/ken/ada contains these files:
.
..
temp
all.zip
README
posix.zip
sm
posix
```

cgi
tia
x
rcsinfo.txt
text_only
original
lintel
texttools
smbeta2.zip
trials
plugins
texttools.zip

The last position is texttools.zip

The first position is .

16.10 Stdio Files

C has a library called stdio, or STanDard IO, which contains standard operations for text files. Loosely, stdio is the C equivalent of Ada's Text_IO package.The standard gnat package cstreams(?) provides a thin binding to many of the stdio functions. In this section, we'll looking at using stdio directly.

Some of the stdio functions can't be used from Ada because of differences in the languages. For example, printf, the standard command for writing to the screen, has a variable number of parameters. Because there's no way to express a variable number of parameters in Ada, printf can't be imported into Ada.

```
with System;
type AStdioFileID is new System.Address;

function fputc( c : integer; fid : AStdioFileID ) return integer;
pragma import( C, fputc, "fputc" );
```
Part of standard C library. Writes one charcter to a file.

```
function fputs( s : string; fid : AStdioFileID ) return integer;
pragma import( C, fputs, "fputs" );
```
Writes a C string to a file.

```
function ferror( fid : AStdioFileID ) return integer;
pragma import( C, ferror);
```
Return error from last file operation, if any.

```
procedure clearerr( fid : AStdioFileID );
pragma import( C, clearerr);
```
Clear the error and end of file information.

```
function feof( fid : AStdioFileID ) return integer;
pragma import( C, feof);
```
Return non-zero if you are at the end of the file.

```
function fileno( fid : AStdioFileID ) return integer;
pragma import( C, fileno);
```

Return the file number for use with Linux file kernel calls.

function flock(fd, operation : integer) **return** integer;
pragma import(C, flock);
Locks or unlocks a file (or a portion of a file).
This is for compatibility with other UNIXes--use fcntl instead.

Operation: LOCK_SH (1) - shared lock
LOCK_EX (2) - exclusive lock
LOCK_NB (4) - no block flag (may be added to others)
LOCK_UN (8) - unlock

16.11 Stdio Pipes

Pipes are the equivalent of shell command pipes formed by the '|' character. You can open a pipe to or from a shell command, depending if the pipe is for writing or reading respectively.

These single direction pipe commands are a part of the standard C library.

function popen(command, mode : string) **return** AStdioFileID;
pragma import(C, popen, "popen");
Opens a pipe to a Linux shell command.Mode can be "w" for write or "r" for read.

procedure pclose(result : **out** integer; fid : AStdioFileID);
pragma import(C, pclose, "pclose");
pragma import_valued_procedure(pclose);
Closes a pipe.

The following program prints to a printer by opening a pipe to the lpr command.

```
with Ada.Text_IO, System, SeqIO;
use Ada.Text_IO;

procedure printer2 is
   -- a program for simple printing

   ---> Pipe Stuff -------------------------------------

   type AStdioFileID is new System.Address;
   -- a pointer to a C standard IO (stdio) file id

   function popen( command, mode : string )  return AStdioFileID;
   pragma import( C, popen, "popen" );
   -- opens a pipe to command

   procedure pclose( result : out integer; fid : AStdioFileID );
   pragma import( C, pclose, "pclose" );
   pragma import_valued_procedure( pclose);
   -- closes a pipe

   function fputc( c : integer; fid : AStdioFileID ) return integer;
   pragma import( C, fputc, "fputc" );
   -- part of standard C library.Writes one charctera to a file.

   function fputs( s : string; fid : AStdioFileID ) return integer;
   pragma import( C, fputs, "fputs" );
   -- part of standard C library.Writes a string to a file.
```

```
    PipeID : AStdioFileID; -- File ID for lpr pipe

    procedure BeginPrinting is
       -- open a pipe to lpr
    begin
       Put_Line( "Opening pipe to lpr ..." );
       PipeID := popen( "lpr" & ASCII.NUL, "w"& ASCII.NUL);
    end BeginPrinting;

    procedure EndPrinting is
       -- close the pipe.Result doesn't matter.
       -- Linux normally will not eject a page when
       -- printing is done, so we'll use a form feed.
       Result : integer;
    begin
       Result := fputc( character'pos( ASCII.FF ), PipeID);
       pclose( Result, PipeID );
    end EndPrinting;

    --> Input/Output Stuff --------------------------------

    procedure Print( s : string ) is
       -- print a string to the pipe, with a carriage
       -- return and line feed.
       Result : integer;
    begin
       Result := fputs( s & ASCII.CR & ASCII.LF & ASCII.NUL, PipeID );
    end Print;

begin

    -- Open the pipe to the lpr command


    Put_Line( "Starting to print..." );
    BeginPrinting;
    Print( "Sales Report" );
    Print( "------------" );
    Print( "" );

    Print( "Sales were good" );

    -- Now, close the pipe.

    EndPrinting;

    Put_Line( "Program done...check the printer" );

end printer2;
```

16.12 Memory Management

The amount of virtual memory for a process depends on the processor. For Intel x86 processors, your program and data must be 3 Gigabytes or less. (An additional 1 Gigabyte per process is reserved for the kernel, accounting for the full 32-bits of addressing space.)

[not finished--KB]

16.13 The Virtual Consoles

The virtual consoles are controlled by the ioctl() function.

[not finished--KB]

The following example catches SIGWINCH signals and reports the new window size.

```
with Ada.Interrupts.Names;
useAda.Interrupts;

package sigwinch is

protected SignalHandler is

    procedure SizeChangeHandler;
    pragma Attach_Handler( SizeChangeHandler, Names.SIGWINCH );
    -- this handler will catch SIGWINCH signals, a window size
    -- change

end SignalHandler;

end sigwinch;
```

```
with Ada.Text_IO;
useAda.Text_IO;

package body sigwinch  is

    -- imported C functions

    TIOGWINSIZ : constant integer := 16#5413#;
    -- get window size ioctl request

    type WindowSizeInfo is record
        row, column, unused1, unused2 : short_integer;
    end record;
    pragma pack( WindowSizeInfo );

    -- the window size information returned by ioctl

    type AFileID is new integer;
    -- a file descriptor, a new integer for safety

    procedure ioctl_winsz( Result : out integer; fid : AFileID; request : integer;
    info : in out WindowSizeInfo );
    pragma import( C, ioctl_winsz, "ioctl");
    pragma import_valued_procedure( ioctl_winsz, "ioctl" );
    -- get the size of the window

    function open( path : string; flags : integer ) return AFileID;
    pragma import( C, open, "open" );
    -- open a file (in this case, the tty)

    procedure close( fid : AFileID );
    pragma import( C, close, "close" );
    -- close a file
```

```
    -- The Signal Handler

    protected body SignalHandler is

        procedure SizeChangeHandler is
        -- handle a window size change, SIGWINCH

            fid: AFileID;-- open's file ID

            Result : integer; -- function result of ioctl

            Info: WindowSizeInfo; -- window size returned by ioctl

        begin

            fid := Open( "/dev/tty" & ASCII.NUL, 0 );

            ioctl_winsz( Result, fid, TIOGWINSIZ, Info );

            if Result = 0 then
                Put_Line( "Window is now " & info.column'img & " x " & info.row'img);
            else
                Put_Line( "ioctl reports an error" );
            end if;

            Close( fid );

        end SizeChangeHandler;

    end SignalHandler;

end sigwinch;
```

```
with Ada.Text_IO, sigwinch;
useAda.Text_IO, sigwinch;

procedure winch is
begin
    Put_Line( "This program catches SIGWINCH signals");
    New_Line;

    Put_Line( "It will stop running is 60 seconds. If you are using" );
    Put_Line( "X Windows, move the window to send signals.");
    New_Line;

    delay 60.0; -- run for 60 seconds

end winch;
```

16.14. Making Database Queries

16.14.1 mySQL

mySQL (pronounced "my ess que ell") is a free, high-performance database from T.c.X. It's available for a number of platform, including Linux. The mySQL home page is http://www.mysql.org.

mySQL comes with a C library called "mysqlclient". If an Ada program links in this library with "-lmysqlclient", it can access mySQL databases. You issue commands to the database called queries using the database language SQL (pronounced "sequel").

Connecting to a mySQL database is a six step process:

1. Open a new connect using **mysql_init**.
2. Login using **mysql_real_connect**.
3. Perform database queries with **mysql_query** or **mysql_real_query**. real_query allows binary data in the query.
4. Retrieve the results using **mysql_store_result** or **mysql_use_result**.
5. Free any memory using **mysql_free_result**.
6. Close your connection with **mysql_close**.

Usually, a null point or non-zero integer result indicates an error. **mysql_errono** returns the error.

Complete documentation is available from the mySQL web site.

16.14.2 PostgreSQL

Not finished--KB

16.15 Dynamic Loading

Not finished--KB

16.16 Writing Linux Modules

If you are writing kernel components, you'll have to use "GNORT" (**pragma no_run_time**). Without the run time components, you won't be able to use some Ada packages and features, including exceptions and the Text_IO library (which uses exceptions). You'll have to import the appropriate C functions to do I/O. Without the run time library, you're source code will also be much smaller, comparable with C.

If you have the time, you can always copy some of the standard Ada packages to a separate directory and compile them into your GNORT project, effectively creating your own small, custom run-time system.

The Gnat run-time system is described at http://www.iuma.ulpgc.es/users/jmiranda/.

For details on how to program for the Linux kernel, read Linux Kernel Module Programming Guide. To create a Linux kernel module in Ada, you will also need to register a *license*. In C, this is accomplised with the MODULE_LICENSE macro. The Linux kernel will expect *init_module* and *cleanup_module*subprograms to exist.

```
with Interfaces.C;
```

```
package SomeModule is

    -- Module Variables

    type Aliased_String is array (Positive range <>)
    of aliased Character;
    pragma Convention (C, Aliased_String);

    Kernel_Version: constant Aliased_String:="2.4.18-5" & Character'Val(0);
    pragma Export (C, Kernel_Version, "kernel_version");

    Mod_Use_Count: Integer;
    Pragma Export (C, Mod_Use_Count, "mod_use_count_");

    -- Kernel Calls

    procedure Printk( s : string );
    pragma import( C, printk, "printk" );

    -- Our Module Functions

    function Init_Module return Interfaces.C.int;
    pragma Export (C, Init_Module, "init_module");

    procedure Cleanup_Module;
    pragma Export (C, Cleanup_Module, "cleanup_module");
end SomeModule;

package body SomeModule is
    -- sample module layout

    function Init_Module return Interfaces.C.int is
    begin
        Printk("Hello,World!" & Character'val(10) & character'val(0));
        return 0;
    end Init_Module;

    procedure Cleanup_Module is
    begin
        Printk("Goodbye , World!" & Character'val(10) & character'val(0));
    end Cleanup_Module;
  end SomeModule;
```

Multiple object files must be combined into a single loadable module object file using the *ld* command. Never use -fPIC when working with the kernel. You may need the '-a' flag and '-s' flags so that Gnat will recompile the system package and related low-level Gnat files, or copy them by hand and compile them yourself (use the -gnatg switch).

Use *insmod*, *lsmod* and *rmmod* to install, test and uninstall your module.

```
$ insmod hello.o
      Hello, World!
```

For embedded systems, there is RTEMS, an embedded run time, and supports a more complete Ada runtime (with tasking) on some targets. Seehttp://www.rtems.com/.

```
with Ada.Text_IO;
```

```
use Ada.Text_IO;

procedure nrt2 is
   -- Simple Program
begin
   put_line( "Hello World" );
end nrt2;
```

```
pragma no_run_time;

procedure nrt is
   -- Same as nrt2 but using no run time

   type file_id is new integer;

   -- No Ada.Text_IO possible, so we'll write our own
   -- that talks directly to the Linux kernel

   procedure write_char( amount_written : out long_integer;
                id : file_id;
                buffer : in out character;
                amount2write : long_integer );
   pragma import( C, write_char, "write" );
   pragma import_valued_procedure( write_char, "write" );

   procedure put( c : character ) is
      result : long_integer;
      buf    : character := c;
   begin
      write_char( result, 1, buf, 1 );
   end put;

   procedure new_line is
   begin
      put( character'val( 10 ) );
   end new_line;

   procedure put_line( s : string ) is
      pragma suppress( index_check, s );
      -- s(i) won't throw a range error, but Gnat checks for it
      -- by default.  Exceptions are a part of the run time.
   begin
      for i in s'range loop
            put( s(i) );
      end loop;
      new_line;
   end put_line;

begin
   put_line( "Hello World" );
end nrt;
```

16.17 Linux Sound

The Linux sound capabilities, called OSS, were developed by 4front technologies.You can find more advanced documentation at their websitehttp://www.opensound.com. This section describes only the basic functions.

The newest Linux sound standard is ALSA.

Most distributions have OSS in the kernel by default, but there's no reason that OSS must be present--it can always be turned off for computers without a sound card.

16.17.1 Detecting a Sound Card

Open the file /dev/sndstatus. If there is no error, the computer has a sound card.

16.17.2 Playing Sound Samples

There are no C libraries or kernel calls to play sound samples. Instead, there is a device file called /dev/audio which plays sound samples in the .au sound format.

The .au sound format consists of a header describing the sound followed by the actual sound data. The header looks like this:

```
type AAUHeader is record
    Magic : integer;        -- a unique number denoting a .au file,
                                    -- as used with the magic file, SND_MAGIC
                                    -- Hex 646E732E (bytes 2E, 73, 6E, 64)
    dataLocation : integer; -- offset or pointer to the sound data
    dataSize: integer;      -- number of bytes of sound data
    dataFormat: integer;    -- the data format code
    samplingRate : integer; -- the sampling rate
    channelCount : integer; -- the number of channels
    info1, info2, info3, info4 : character;-- name of sound
end record;
```

dataLocation is an offset to the first byte of the sound data. If there's no sound name, it's 28, the size of the header. It can a pointer to the data, depending on the dataFormat code, but that doesn't apply if you're playing a .au file.

dataSize is the size of the sound data in bytes, not including the header.

dataFormat describes how the sound data is to be interpreted. Here is a table of some common values.

Value	Code	Format
0	SND_FORMAT_UNSPECIFIED	unspecified format
1	SND_FORMAT_MULAW_8	8-bit mu-law samples
2	SND_FORMAT_LINEAR_8	8-bit linear samples
3	SND_FORMAT_LINEAR_16	16-bit linear samples
4	SND_FORMAT_LINEAR_24	24-bit linear samples
5	SND_FORMAT_LINEAR_32	32-bit linear samples
6	SND_FORMAT_FLOAT	floating-point samples
7	SND_FORMAT_DOUBLE	double-precision float samples
8	SND_FORMAT_INDIRECT	fragmented sampled data
10	SND_FORMAT_DSP_CORE	DSP program
11	SND_FORMAT_DSP_DATA_8	8-bit fixed-point samples
12	SND_FORMAT_DSP_DATA_16	16-bit fixed-point samples

13	SND_FORMAT_DSP_DATA_24	24-bit fixed-point samples
14	SND_FORMAT_DSP_DATA_32	32-bit fixed-point samples
16	SND_FORMAT_DISPLAY	non-audio display data
18	SND_FORMAT_EMPHASIZED	16-bit linear with emphasis
19	SND_FORMAT_COMPRESSED	16-bit linear with compression
20	SND_FORMAT_COMPRESSED_EMPHASIZED	Combo of the two above
21	SND_FORMAT_DSP_COMMANDS	Music Kit DSP commands

SamplingRate is the playback rate in hertz.CD quality samples are 44100.

channelCount is 1 for mono, 2 for stereo.

The *info* characters are a C null-terminated string giving a name for the sound. It's always at least 4 characters long, even if unused.

In order to play a sound, treat /dev/audio as if it were a device attached to your computer for playing .au sounds.Write a program to open /dev/audio for writing and write the .au sound to it.

Playing sounds is a natural candidate for multithreading because you don't want your entire program to stop while a sound is being played.

The following program uses the seqio generic package we developed above to play an .au sound through /dev/audio.

```
with seqio;
with Ada.Text_IO;
use Ada.Text_IO;

procedure playsnd is

   -- simple program to play an .au sound file

   package byteio is new seqio( short_short_integer );
   -- sequential files of bytes

   au_filename : constant string := "beep.au";
   -- sound file to play.  supply the name of the .au file to play

   au_file: byteio.AFileID; -- the sound file
   dev_audio: byteio.AFileID; -- /dev/audio device

   soundbyte : short_short_integer;

begin

   Put_Line( "Playing " & au_filename & "...");

   -- open the files

   au_file := byteio.Open( au_filename, read => true);
   dev_audio := byteio.Open( "/dev/audio", read => false);

   -- read until we run out of bytes, send all bytes to
   -- /dev/audio.The end of file will cause a seqio_error

   begin
      -- nested block to catch the exception

      loop
```

```
            byteio.Read( au_file, soundbyte );
            byteio.Write( dev_audio, soundbyte );
        end loop;

    exception when byteio.seqio_error =>
        null; -- just leave block
    end;

    -- close files

    byteio.Close( au_file );
    byteio.Close( dev_audio );

    Put_Line( "All done" );

exception when others =>
    Put_Line( "Oh, oh! An exception occurred!" );
    byteio.Close( au_file );
    byteio.Close( dev_audio );
    raise;

end playsnd;
```

16.17.3 Using the Mixer

You control the mixer chip, if your sound card has one, by using the ioctl() kernel call. If there is no mixer, the ioctl() function returns -1. **Mixer Functions Table**

SOUND_MIXER_NRDEVICES	17	Number of mixer functions on this computer
SOUND_MIXER_VOLUME	0	The master volume setting
SOUND_MIXER_BASS	1	Bass setting
SOUND_MIXER_TREBLE	2	Treble setting
SOUND_MIXER_SYNTH	3	FM synthesizer volume
SOUND_MIXER_PCM	4	/dev/dsp volume
SOUND_MIXER_SPEAKER	5	internal speaker volume, if attached to sound card
SOUND_MIXER_LINE	6	"line in" jack volume
SOUND_MIXER_MIC	7	microphone jack volume
SOUND_MIXER_CD	8	CD input volume
SOUND_MIXER_IMIX	9	Recording monitor volume
SOUND_MIXER_ALTPCM	10	volume of alternate codec, on some cards
SOUND_MIXER_RECLEV	11	Recording level volume
SOUND_MIXER_IGAIN	12	Input gain
SOUND_MIXER_OGAIN	13	Output gain
SOUND_MIXER_LINE1	14	Input source 1 (aux1)
SOUND_MIXER_LINE2	15	Input source 2 (aux2)
SOUND_MIXER_LINE3	16	Input source 3 (line)

Reading or writing values have a special bit set [Ken check using soundcard.h].

Ioctl calls return an integer value. Volume levels can be 0 to 100, but many sound cards do not offer 100 levels of volume. /dev/mixer will set the volume to setting nearest to your requested volume.

[Not complete--KB]

```
Sound_mixer_volume : constant integer := 0;
Sound_Mixer_Read : constant integer := ?;
Sound_Mixer_Write : constant integer := ?;

oldVolume : integer;

ioctlResult := Ioctl( mixer_file_id, sound_mixer_read + sound_mixer_volume, oldVolume );

-- master volume now in oldVolume

if ioctlResult = -1  then
    Put_Line( "No mixer installed " );
end if;

newVolume := 75;

ioctlResult := ioctl( mixer_file_id, sound_mixer_write + sound_mixer_volume, newVolume );

-- master volume is 75%
```

16.17.4 Recording Sound Samples

Recording sounds works is the reverse process of playing sounds. Open /dev/dsp for reading, and it returns sound data. However, before you can begin recording from /dev/dsp, you need to describe how you want the recording done. You need to select the input source, sample format (sometimes called as number of bits), number of channels (mono/stereo) , and the sampling rate (speed). These are set by using the ioctl function on the /dev/dsp file.

To select the input source, you'll need to use /dev/mixer.

[Not finished--KB

```
Sound_Mixer_Recsrc : constant integer := ?;
Sound_Mixer_Read : constant integer := ?;
Sound_Mixer_Write : constant integer := ?;

newInputSource := Sound_Mixer_Mic;

ioctlResult := ioctl( mixer_file_id, sound_mixer_write + sound_mixer_recsrc, newInputSource
);
```

Common formats

/* Audio data formats (Note! U8=8 and S16_LE=16 for compatibility) */

AFMT_QUERY	16#00000000#	Returns current format
AFMT_IMA_ADPCM	16#00000004#	ADPCM compressed data
AFMT_U8	16#00000008#	Unsigned bytes
AFMT_S16_LE	16#00000010#	Little endian signed 16 bits
AFMT_S16_BE	16#00000020#	Big endian signed 16 bits
AFMT_S8	16#00000040#	Signed bytes

AFMT_U16_LE	16#00000080#	Little endian U16 bits
AFMT_U16_BE	16#00000100#	Big endian U16 bits
AFMT_MPEG	16#00000200#	MPEG (2) audio

```
sndctl_dsp_setfmt : constant integer := ?;

newFormat : integer;

newFormat := 16#0000010#;

ioctlResult := ioctl( dsp_id, sndctl_dsp_setfmt, newFormat );
-- recording format now 16 bit signed little endian

if newFormat /= 16#00000010  then
    Put_Line( "Sound card doesn't support recording format" );
end if;
```

Selecting mono or stereo recording is a matter of 0 or 1 respectively.

```
sndctl_dsp_stereo : constant integer := ?;

stereo : integer;

stereo := 1;

...

ioctlResult := ioctl( dsp_id, sndctl_dsp_stereo, stereo );
-- recording format now stereo

if stereo /= 1 then
    Put_Line( "Sound card doesn't support stereo" );
end if;
```

Finally, select a sampling rate.

```
sndctl_dsp_speed : constant integer := ?;

newSpeed : integer;

newSpeed:= 44100;

ioctlResult := ioctl( dsp_id, sndctl_dsp_speed, newFormat );
-- recording now CD quality sampling speed

if newSpeed /= 44100  then
    Put_Line( "Sound card doesn't support sampling speed" );
end if;
```

Now read /dev/dsp for the raw sound data. If you want to save the sound as an .au file, you'll have to create the .au header information to attach to the sound data.

16.18 Audio CDs

16.19 Kernel Pipes

16.20 Shared Memory

Shared Memory Flags

IPC_CREAT		Create new shared memory block
IPC_EXCL		plus read, write and execute bits.

IPC_PRIVATE indicates no key is supplied.

function shmget(key : key_t; bytes : integer; shmflag : integer) **return** integer;
pragma import(C, shmget);
Key is an id you supply to identify the memory (or IPC_PRIVATE for no key). bytes is the minimum amount of memory that you need. shmflag indicates options for this call. Returns -1 on error, or an id for the memory.
Example: shmid := shmget(mykey, 4096, IPC_CREAT+IPC_EXCL+8#0660#);

function shmat(result : **out** system.address; shmid : integer; shmaddr : system.address; shmflag : integer) **return**system.address;
pragma import(C, shmat);
Shared memory attach. Makes shared memory accessible by returning a pointer to it. shmid is the id returned by shmget. if shmaddr isn't zero, the kernel will the address you give instead of chosing one itself. shmflags are additional options. Returns the address of the shared memory, or an address of -1 for an error.
Example: shmat(ShmCPtr, myID, To_Address(null), 0);
ShmPtr := To_Address(ShmCPtr);

SHM_RDONLY - this memory is read-only (that is, as if it was constant).
SHM_RND - allows your shmaddr to be truncated to a virtual memory page boundary.

function shmdt(shmaddr : system.address) **return** integer;
pragma import(C, shmdt);
Shared memory detach. Removes the association of the shared memory to the pointer. Returns 0 if the memory was detached, -1 for failure.
Example: Result := shmdt(To_Address(ShmPtr));

function shmctl(shmid : integer; cmd : integer; info : system.address) return integer;
pragma import(C, shmctl);
Performs miscellaneous shared memory functions, including deallocating shared memory allocated with shmget. Returns 0 if the function was successful, or -1 for a failure.
Example: Result := shmctl(myID, IPC_RMID, To_Address(null));

IPC_RMID - deallocate shared memory

16.21 Message Queues

What are they?

Message queues are one of three IPC (Inter Process Communication) ways of System V. The others are shared memory and semaphores. Message queues are linked lists of messages maintained by the kernel. A process can set one up, disappear, and the queue still remains.

Are they usefull?
Yes, they provide a fairly simple way of passing messages between processes. They are also very fast.

A way of using them

We'll look at a simple case where two processes will pass a message between each other.

First we'll need a System V IPC key. Ftok generates a, almost always uniqe, key by gathering some info from a user provided file. This key is needed when we create a queue.

type key_t **is new** integer;

function ftok(path : string; proj_id : integer) **return** key_t;
Convert a project id number and a pathname of an accessible file to a key that can be used by Linux's System V interprocess communication features (that is, message queues.)
Example: my_queue := ftok("./queue_file.que" & ASCII.NUL, my_proj);

Ftok proberbly means "File To Key".

Although Ada integers and C "int" types are identical, we'll use the Interfaces.C package for maximum portability.

```
-- C is key_t ftok(const char *pathname, int proj_id);

package C renames Interfaces.C;
type Key_t is new C.Int;
pragma Convention (C, Key_t);
function C_Ftok(Pathname : in String; Proj : in C.Int) return Key_t;
pragma Import(C,C_Ftok,"ftok");
```

By calling C_Ftok, with Proj greater than 0, we get a key, or -1 for error. We now wants to create a message queue, with this key.

function msgget(key : key_t; flags : integer) **return** integer;
Return the message queue id number associated with the key. A new message queue will be created if the key has value IPC_PRIVATE or of no queue exists. The flags indicate indicate the permissions for the message queue.
Example: qid := msgget(key, IPC_CREAT+8#660#;

Mesget can be called with a lot of options, but we'll go for getting an id for a queue, and if it does not exists, we create it.

```
int msgget(key_t key, int msgflg);

IPC_CREAT      : constant C.Int :=  512 ;
IPC_PERMISSIONS : constant C.Int := 8#660#;
function C_Msgget(Key : Key_t; Msgflg : C.Int) return C.Int;
pragma Import (C, C_Msgget, "msgget");
```

By calling msgget with IPC_CREAT + IPC_PERMISSIONS, and the generated key, we get the identity of a message queue, that either exists, or is newly created with the corresponding permissons. The execute flag has no meaning for message queues.

msgsnd msgsnd(2)

Now we want to do something, but first, have a look at 'ipcs -q'. This command lists message queues in the system.

We send a record that looks like this

```
struct msgbuf {
    long    mtype;   /* message type, must be > 0 */
    char    mtext[1];       /* message data */
  };
```

This translates in Ada into a record with an C.long member + another member of arbitrary kind, ie a record.

```
type Message_Type is record
   M_Type        : C.Long := 100;
   An_Integer    : Integer;
   Anther_Integer : Integer;
end record;

function C_Send(Queue_Identity : in C.Int;
                                Message       : in Message_Type) return C.Int is

type Message_Pointer_Type is access all Message_Type;

Tmp_Msg     : aliased Message_Type := Message;
Tmp_Msg_Ptr : Message_Pointer_Type := Tmp_Msg'Access;
-- All 'size are in bits. There are System.Storage_Unit bits in a byte
Local_Size : C.Int := C.Int((C.Long'Size + 2 * Integer'Size)/System.Storage_Unit);

function C_Msgsnd(Msqid : C.Int; Msgp   : Message_Pointer_Type;
                                Msgsz : C.Int; Msgflg : C.Int) return C.Int;
pragma Import (C, C_Msgsnd, "msgsnd");
begin
   return C_Msgsnd(Queue_Identity,Tmp_Msg_Ptr,Local_Size,0);
end C_Send;
```

This will send a record containing 2 integers, with message type set to 100. The type can be used in receiving messages.

msgrec

msgrcv(2)

When receiving from a queue, we can use fifo-order or just look at messages of a certain kind. This is determined by the Msg_Type parameter. 0 means fifo, greater than 0 means first message of that type, less than 0 means message with lowest type less than or equal to the absolute value of Msg_Type.

```
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);

function C_Receive(Queue_Identity : in C.Int;
                                Msg_Type      : in C.Long;
                                Msg_Flag      : in C.Int := 0) return Message_Type is

   type Message_Pointer_Type is access all Message_Type;
   Receive_Failure : exception;
   function C_Msgrcv(Msqid   : C.Int; Msgpointer : Message_Pointer_Type;
                                Msgsize : C.Int; Msgtype    : C.Long;
```

```
                                    Msgflag : C.Int)              return C.Int;
      pragma Import (C, C_Msgrcv, "msgrcv");

      Tmp_Msg     : aliased Message_Type;
      Tmp_Msg_Ptr : Message_Pointer_Type := Tmp_Msg'Access;
      Result      : C.Int               := C.Int'First;
   begin
      Result := C_Msgrcv(Queue_Identity, Tmp_Msg_Ptr,
                                    C.Int((Message_type'Size + C.Long'Size)/System.Storage_Unit),
                                    Msg_Type, Msg_Flag);
      if (Result >= 0) then
         return Tmp_Msg_Ptr.all;
      else
         raise Receive_Failure;
      end if;
   end C_Receive;
```

msgctl

msgctl(2)

With msgctl, you can examine and remove existing message queues.

Putting it all together

We'll do 2 simple programs, the first sends a message to a message queue and creates the queue if it does not exists. Then the program exits. The second program retrieves a message from a queue. If the queue does not exists, it creates the queue, and blocks until a message arrives. It then prints out the sum of the fields in the record. The programs was tested with Mandrake 9 and Gnat 3.15p. 4 files are involved:

```
-- test_def.ads
--
-- a package to provide a simple message queue binding

with Interfaces.C;
package Test_Def is
   package C renames Interfaces.C;

   type Key_t is new C.Int;
   pragma Convention (C, Key_t);

   IPC_CREAT      : constant C.Int :=  512 ;
   IPC_PERMISSIONS : constant C.Int := 8#660#;

   type Message_Type is record
      M_Type        : C.Long := 100;
      An_Integer    : Integer;
      Another_Integer : Integer;
   end record;

   Receive_Failure : exception;

   function C_Ftok(Pathname : in String; Proj : in C.Int) return Key_t;
   pragma Import(C,C_Ftok,"ftok");

   function C_Msgget(Key : Key_t; Msgflg : C.Int) return C.Int;
   pragma Import (C, C_Msgget, "msgget");
```

```
   function C_Send(Queue_Identity : in C.Int;
                                Message       : in Message_Type) return C.Int ;
   function C_Receive(Queue_Identity : in C.Int;
                                   Msg_Type      : in C.Long;
                                   Msg_Flag      : in C.Int := 0) return Message_Type;
end Test_Def;
```

```
-- test_def.adb package body
--
-- a package to provide a simple message queue binding

with Ada.Text_IO;
with System;
   package body Test_Def is
   use C;

   function C_Send(Queue_Identity : in C.Int;
                                Message       : in Message_Type) return C.Int is
      -- Send a message through the message queue.  Wrapper function for msgsnd.

      type Message_Pointer_Type is new System.Address;

      Tmp_Msg     : aliased Message_Type := Message;
      Tmp_Msg_Ptr : Message_Pointer_Type := Tmp_Msg'Address;
      -- All 'size are in bits. There are System.Storage_Unit bits in a byte
      Local_Size : C.Int := C.Int((C.Long'Size + 2 * Integer'Size)/System.Storage_Unit);

      function C_Msgsnd(Msqid : C.Int; Msgp   : Message_Pointer_Type;
                                   Msgsz : C.Int; Msgflg : C.Int) return C.Int;
      pragma Import (C, C_Msgsnd, "msgsnd");

   begin
      return C_Msgsnd(Queue_Identity,Tmp_Msg_Ptr,Local_Size,0);
   end C_Send;

   function C_Receive(Queue_Identity : in C.Int;
                                   Msg_Type      : in C.Long;
                                   Msg_Flag      : in C.Int := 0) return Message_Type is
      -- Receive a message from a message queue.  Wrapper function for msgrcv.

      type Message_Pointer_Type is new System.Address;

      function C_Msgrcv(Msqid   : C.Int; Msgpointer : Message_Pointer_Type;
                                   Msgsize : C.Int; Msgtype    : C.Long;
                                   Msgflag : C.Int)              return C.Int;
      pragma Import (C, C_Msgrcv, "msgrcv");

      Tmp_Msg     : aliased Message_Type;
      Tmp_Msg_Ptr : Message_Pointer_Type := Tmp_Msg'Address;
      Result      : C.Int               := C.Int'First;

   begin
      Result := C_Msgrcv(Queue_Identity, Tmp_Msg_Ptr,
                                   C.Int((Message_type'Size + C.Long'Size)/System.Storage_Unit),
                                   Msg_Type, Msg_Flag);
      Ada.Text_Io.Put_Line("Lenght of message:" & C.Int'Image(Result));
```

```
        if Result >= 0 then
            return Tmp_Msg_Ptr.all;
        else
            raise Receive_Failure;
        end if;
    end C_Receive;

end Test_Def;
```

```
-- test_send.adb
--
-- a program to test our message queue package

with Ada.Text_IO;
with Test_Def; use Test_Def;
procedure Test_Send is
    use C;
    Key  : Key_T ;
    Q_Id : C.Int ;
    Message : Message_Type;
    Result : C.Int;
begin
    Key  := C_Ftok("/etc/profile" & Ascii.NUL,1);
    Q_Id := C_Msgget(Key, IPC_CREAT + IPC_PERMISSIONS);

    Message.An_Integer := 40;
    Message.Another_Integer := 2;
    Result := C_Send(Q_Id,Message);
    Ada.Text_Io.Put_Line("Created/got hold of Key:" & Key_T'Image(Key) &
    " Q-id:" & C.Int'Image(Q_Id) & "Result sending:" & C.Int'Image(Result));

    Ada.Text_Io.Put_Line("Check with 'ipcs -q'");
end Test_Send;
```

```
-- test_receive.adb
--
-- another program to test our message queue package

with Ada.Text_IO;
with Test_Def; use Test_Def;
procedure Test_Receive is
    use C;
    Key  : Key_T ;
    Q_Id : C.Int ;
    Message : Message_Type;
begin
    Key  := C_Ftok("/etc/profile" & Ascii.NUL,1);
    Ada.Text_Io.Put_Line("Received with  Key:" & Key_T'Image(Key) );

    Q_Id := C_Msgget(Key, IPC_CREAT + IPC_PERMISSIONS);
    Ada.Text_Io.Put_Line("Q-id:" & C.Int'Image(Q_Id));

    Message := C_Receive(Q_Id, Message.M_Type);
    Ada.Text_Io.Put_Line("Received with  Key:" & Key_T'Image(Key) &
```

```
      " Q-id:" & C.Int'Image(Q_Id) & " The sum of the two fields are:" &
      Integer'Image(Message.An_Integer + Message.Another_Integer));

   Ada.Text_Io.Put_Line("Check with 'ipcs -q'");
end Test_Receive;
```

Compile with gnatmake test_send.adb and gnatmake test_receive.adb. Run test_send first, check with 'ipcs -q' and then run test_receive. remove the queue with 'ipcrm'

16.22 Semaphores

16.23 Sockets

Send (sentto and sendmsg) are supersets of write. When you use write on a socket, it's actually implemented using the send family.

Write will not work on UDP because it's connectionless. Use send to specify an address everytime.

Protocol Families
PF_INET Internet (IPv4)
PF_INET6 Internet (IPv6)
PF_IPX Novell
PF_NETLINK Kernel user interface device
PF_X25 ITU-T X.25 / ISO-8208
PF_AX25 Amateur radio AX.25
PF_ATMPVC Access to raw ATM PVCs
PF_APPLETALK Appletalk
PF_PACKET Low-level packet interface

Socket Types
SOCK_STREAM Two-way reliable connection, with possible out-of-band transmission (eg. TCP/IP)
SOCK_DGRAM (Datagram) Connectionless, unreliable messages (eg. UDP/IP)
SOCK_SEQPACKET Sequenced, reliable datagram connection.
SOCK_RAQ Raw network protocol access.
SOCK_RDM Reliable, unordered datagrams.

function socket(domain, soctype, protocol : integer) **return** integer;
pragma import(C, socket);
Creates a network socket for protocol family *domain*, connection type *soctype*, and a protocol (0 uses the default protocol). Returns -1 on an error, or else a kernel file descriptor for the socket.
Example: mySocket := socket(PF_INET, SOCK_STREAM, 0); -- open a standard Internet socket

procedure connect(result : **out** integer; socket : integer; addr : **in out** socketaddr; addrlen : integer);
pragma import(C, connect);
pragma import_valued_procedure(connect);
Connects to a server on the network. socket is the socket to use; addr is the machine and service to connect to; addrlen is the length of the addr record. Returns -1 on an error, 0 for success.
Example: connect(result, webserver, webserver'size/8); -- connect to the web server described by the webserver record

function shutdown(socket, how : integer) **return** integer;
pragma import(C, shutdown);
Shuts down one or both directions of a socket. This is used, for example, by web browsers to let the

server know there are no more HTTP requests being sent. Returns 0 on sucess, -1 on failure.
Example: result := shutdown(mysocket, 1);

procedure bind(result : **out** integer; myaddr : **in out** sockaddr, addrlen : integer);
pragma import(C, bind);
pragma import_valued_procedure(bind);
Registers your server on a particular port number with the Linux kernel. addrlen is the length of myaddr. Returns 0 on success, -1 on failure.
Example: bind(result, myservice, myservice'size/8);

function listen(socket : integer; backlog : integer) **return** integer;
pragma import(C, listen); **pragma** import_valued_procedure(listen);
Prepares a socket for your server to listen for incoming connections. Backlog is the maximum number of established connections that can be queued. Returns 0 on success, -1 on failure.
Example: result := listen(mysocket, 10);

procedure accept(result : **out** integer; socket : integer; clientaddr : **in out** sockaddr;
addrlen : **in out** addrlen);
pragma import(C, accept);
pragma import_valued_procuedre(accept);
Returns the next connection to your server. If there are no connections, it waits for a new connection (unless you disabled blocking on the socket.) myaddr is the address of the incoming connection, and addrlen is the size of the address is bytes. addrlen should be initialized to the size of your sockaddr record. You must use listen before accept. Returns -1 on failure, or a new socket with the connection on success. You have to close the new socket when you are finished handling the connection.
Example: len := clientaddr'size/8;
accept(newsocket, listensocket, clientaddr, len);

This section ends with a demonstration of how to get a web page off the Internet.

```
with Ada.Text_IO, Interfaces.C, System.Address_To_Access_Conversions;
use  Ada.Text_IO, Interfaces.C;

procedure websocket is

   -- A program to fetch a web page from a server


   -- Socket related definitions
   --
   -- These are the kernel calls and types we need to create
   -- and use a basic Internet socket.

   type aSocketFD is new int;

   -- a socket file descriptor is an integer -- man socket
   -- make this a new integer for strong typing purposes

   type aProtocolFamily is new unsigned_short;
   AF_INET : constant aProtocolFamily := 2;

   -- Internet protocol PF_Net defined as 2 in
   -- /usr/src/linux/include/linux/socket.h
   -- Make this a new integer for strong typing purposes

   type aSocketType is new int;
```

```
SOCK_STREAM : constant aSocketType := 1;

-- this is for a steady connection.  Defined as 1 in
-- /usr/src/linux/include/linux/socket.h
-- Make this a new integer for strong typing purposes

type aNetProtocol is new int;
IPPROTO_TCP : constant aNetProtocol := 6;

-- The number of the TCP/IP protocol
-- TCP protocol defined as 6 in /etc/protocols
-- See man 5 protocols
-- Make this a new integer for strong typing purposes

type aNetDomain is new integer;
PF_INET : constant aNetDomain := 2;

-- The number of the Internet domain
-- Make this a new integer for strong typing purposes

type aInAddr is record
        addr : unsigned := 0;
end record;
for aInAddr'size use 96;
-- A sockaddr_in record is defined as 16 bytes long (or 96 bits)
-- Request Ada to use 16 bytes to represent this record

type aSocketAddr is record
        family : aProtocolFamily := AF_INET; -- protocol (AF_INET for TCP/IP)
        port   : unsigned_short := 0;  -- the port number (eg 80 for web)
        ip     : aInAddr;              -- IP number
end record;
-- an Internet socket address
-- defined in /usr/src/linux/include/linux/socket.h
-- and /usr/src/linux/include/linux/in.h

function socket( domain   : aNetDomain;
                 stype    : aSocketType;
                 protocol : aNetProtocol )
   return aSocketFD;
pragma import( C, socket );
-- initialize a communication socket.  -1 if error

procedure bind( result : out int; sockfd : aSocketFD;
   sa : in out aSocketAddr; addrlen : int );
pragma import( C, bind );
pragma import_valued_procedure( bind );
-- give socket a name. 0 if successful

procedure Connect( result : out int; socket : aSocketFD;
   sa : in out aSocketAddr; addrlen : int );
pragma import( C, connect );
pragma import_valued_procedure( connect );
-- connect to a (Internet) server.  0 if successful

procedure Close( fd : aSocketFD );
pragma import( C, close );
-- close the socket, discard the integer result
```

```
procedure Read( result : out integer; from : aSocketFD; buffer : in out string;
    buffersize : integer );
pragma import( C, read );
pragma import_valued_procedure( read );
-- read from a socket

procedure Write( result : out integer; from : aSocketFD;
    buffer : system.address; buffersize : integer );
pragma import( C, write );
pragma import_valued_procedure( write );
-- write to a socket

package addrListPtrs is new System.Address_To_Access_Conversions( System.Address );
-- We need to use C pointers with the address list because this is
-- a pointer to a pointer in C.  This will allow us to dereference
-- the C pointers in Ada.

subtype addrListPtr is System.Address;
-- easier to read than System.Address

type aHostEnt is record
        h_name      : System.Address;    -- pointer to offical name of host
        h_aliases   : System.Address;    -- pointer to alias list
        h_addrtype  : int     := 0;      -- host address type (PF_INET)
        h_length    : int     := 0;      -- length of address
        h_addr_list : addrListPtr;       -- pointer to list IP addresses
                                                        -- we only want first one
end record;
-- defined in man gethostbyname

package HEptrs is new System.Address_To_Access_Conversions( aHostEnt );
-- Again, we need to work with C pointers here
subtype aHEptr is System.Address;
-- and this is easier to read
use HEptrs;
-- use makes = (equals) visible

function getHostByName( cname : string ) return aHEptr;
pragma import( C, getHostByName );
-- look up a host by it's name, returning the IP number

function htons( s : unsigned_short ) return unsigned_short;
pragma import( C, htons );
-- acronym: host to network short -- on Intel x86 platforms,
-- switches the byte order on a short integer to the network
-- Most Significant Byte first standard of the Internet

procedure memcpy( dest, src : System.Address; numbytes : int );
pragma import( C, memcpy);
-- Copies bytes from one C pointer to another.  We could probably
-- use unchecked_conversion, but the C examples use this.

errno : int;
pragma import( C, errno );
-- last error number

procedure perror( s : string );
pragma import( C, perror );
-- print the last kernel error and a leading C string
```

```
procedure PutIPNum( ia : aInAddr ) is
    -- divide an IP number into bytes and display it
    IP : unsigned := ia.addr;
    Byte1, Byte2, Byte3, Byte4 : unsigned;
begin
    Byte4 := IP mod 256;
    IP := IP / 256;
    Byte3 := IP mod 256;
    IP := IP / 256;
    Byte2 := IP mod 256;
    Byte1 := IP / 256;
    Put( Byte4'img );
    Put( "." );
    Put( Byte3'img );
    Put( "." );
    Put( Byte2'img );
    Put( "." );
    Put( Byte1'img );
end PutIPNum;

procedure SendHTTPCommand( soc : aSocketFD; cmd : string ) is
-- Write a HTTP command string to the socket
    amountWritten : integer := 0;
    totalWritten  : integer := 0;
    position      : integer := cmd'first;
begin
    loop
            Write( amountWritten, soc, cmd( position )'address,
                cmd'length - integer( totalWritten ) );
            if amountWritten < 0 then
                Put_Line( Standard_Error, "Write to socket failed" );
                return;
            end if;
            Put_Line( "Sent" & amountWritten'img & " bytes" );
            totalWritten := totalWritten + amountWritten;
            position := position + amountWritten;
    exit when totalWritten = cmd'length;
    end loop;
end SendHTTPCommand;

procedure ShowWebPage( soc : aSocketFD ) is
-- Read the web server's response and display it to the screen
    amountRead : integer := 1;
    buffer : string( 1..80 );
begin
    -- continue reading until an error or no more data read
    -- up to 80 bytes at a time
    while amountRead > 0 loop
        Read( amountRead, soc, buffer, buffer'length );
        if amountRead > 0 then
            Put( buffer( 1..amountRead ) );
        end if;
    end loop;
end ShowWebPage;

ServerName: string := "www.adapower.com";

mySocket    : aSocketFD;     -- the socket
```

```
    myAddress   : aSocketAddr;   -- where it goes
    myServer    : aHEptr;        -- IP number of server
    myServerPtr : HEptrs.Object_Pointer;
    addrList    : addrListPtrs.Object_Pointer;
    Result      : int;

begin

    Put_Line( "Socket Demonstration" );
    New_Line;
    Put_Line( "This program opens a socket to a web server" );
    Put_Line( "and retrieves the server's home page" );
    New_Line;

    -- initialize a new TCP/IP socket
    -- 0 for the third param lets the kernel decide

    Put_Line( "Initializing a TCP/IP socket" );
    Put_Line( "Socket( " & PF_INET'img & ',' & SOCK_STREAM'img &
          ", 0 );" );

    mySocket := Socket( PF_INET, SOCK_STREAM, 0 );
    if mySocket = -1 then
          perror( "Error making socket" & ASCII.NUL );
          return;
    end if;
    New_Line;

    -- Lookup the IP number for the server

    Put_Line( "Looking for information on " & ServerName );
    Put_Line( "GetHostByName( " & ServerName & ");" );

    myServer := GetHostByName( ServerName & ASCII.NUL );
    myServerPtr := HEptrs.To_Pointer( myServer );
    if myServerPtr = null then
          Put_Line( Standard_Error, "Error looking up server" );
          return;
    end if;

    Put_Line( "IP number is" & myServerPtr.h_length'img & " bytes long" );
    addrList := addrlistPtrs.To_Pointer( myServerPtr.h_addr_list );
    New_Line;

    -- Create the IP, port and protocol information

    Put_Line( "Preparing connection destination information" );
    myAddress.family := AF_INET;
    myAddress.port   := htons( 80 );
    memcpy( myAddress.ip'address, addrlist.all, myServerPtr.h_length );
    New_Line;

    -- Open a connection to the server

    Put_Line( "Connect( Result, Socket, Family/Address rec, F/A rec size )" );

    Connect( Result, mySocket, myAddress, myAddress'size/8 );

    Put( "Connect( " & Result'img & "," );
    Put(  myAddress.family'img & "/" );
```

```
    PutIPNum( myAddress.ip );
    Put(  "," & integer'image( myAddress'size / 8 ) & ")" );
    if Result /= 0 then
        perror( "Error connecting to server" & ASCII.NUL );
        return;
    end if;
    New_Line;

    -- Write the request
    --
    -- "GET" returns a web page from a web server
    -- Also send minimal HTTP header using User-Agent
    -- Followed with a blank line to indicate end of command

    Put_Line( "Transmitting HTTP command..." );
    SendHTTPCommand( mySocket,
        "GET /index.html HTTP/1.0" & ASCII.CR & ASCII.LF &
        "User-Agent: WebSocket/1.0 (BigBookLinuxAda Example)" & ASCII.CR & ASCII.LF
        & ASCII.CR & ASCII.LF );
    New_Line;

    -- read web page back

    Put_Line( "---Web Page / Server Results-------------------------" );
    ShowWebPage( mySocket );
    Put_Line( "-----------------------------------------------------" );

    -- close the connection

    Put_Line( "Closing the connection" );
    close( mySocket );

    Put_Line( "Demonstration finished - have a nice day" );

end websocket;
```

```
Socket Demonstration

This program opens a socket to a web server
and retrieves the server's home page

Initializing a TCP/IP socket
Socket(  2, 1, 0 );

Looking for information on www.adapower.com
GetHostByName( www.adapower.com);
IP number is 4 bytes long

Preparing connection destination information

Connect( Result, Socket, Family/Address rec, F/A rec size )
Connect(  0, 2/ 216. 92. 66. 46, 16)
Transmitting HTTP command...
Sent 81 bytes

---Web Page / Server Results-------------------------
HTTP/1.0 200 OK
Date: Wed, 29 Mar 2000 02:32:56 GMT
Server: Apache/1.3.3
```

```
Last-Modified: Thu, 11 Nov 1999 02:03:14 GMT
Etag: "1f31-406-382a23e2"
Accept-Ranges: Bytes
Content-Length: 1030
Content-Type: text/html
Age: 39
Via: HTTP/1.0 csmc2 (Traffic-Server/3.0.3 [uScHs f p eN:t cCHi p s ])

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
    <META NAME="author" CONTENT="David Botton">>
    <META NAME="keywords" CONTENT="Ada AdaPower power source code free treasury repository">
    <META NAME="description" CONTENT="The Ada Source Code Treasurey contains components, procedures,
algorithms and articles for Ada developers.">
    <META http-equiv="Page-Enter" CONTENT="revealtrans(duration=2.0, transition=3)">
    <TITLE>AdaPower.com</TITLE>
    <LINK href="mailto:David@Botton.com" rev="made">
</HEAD>
<FRAMESET COLS="120,*" FRAMEBORDER=0 FRAMESPACING=0 BORDER=0>
        <FRAME SRC="buttons.html" name="menu" frameborder=0 marginheight=0 marginwidth=0 noresize
scrolling=auto border=0>
        <FRAME SRC="http://www.adapower.com/body.html" name="body" frameborder=0 marginheight=5
marginwidth=0 noresize scrolling=auto border=0>
<NOFRAMES>
<meta HTTP-EQUIV="REFRESH" CONTENT="0; URL="body.html">
<body bgcolor="#ffffff" text="#000000">
<a href="body.html">Click here</a>
</BODY>
</BODY>
</NOFRAMES>
</FRAMESET>

</HTML>
-------------------------------------------------------
Closing the connection
Demonstration finished - have a nice day
```

16.24 Memory Management

type aProtection **is new** integer;
type aMapFlag **is new** integer;

function getpagesize **return** long_integer;
pragma import(C, getpagesize);
Return the size of a Linux memory page (that is, the size of the memory blocks that your program and data are broken up into when loaded into memory).

function mmap(start : system.address; size : long_integer; prot : aProtection; flags : aMapFlag; fd : integer; offset : long_integer) **return** system.address;
pragma import(C, mmap);
Allocates size bytes of memory and returns a C pointer. If it failed, -1 is returned. If MAP_FIXED and start are used, the memory will be at the specified address. The protection flags indicate how the memory will be accessed: a signal will be raised on an illegal access. If MAP_ANON is used, fd should be -1 and no file will be associated with the memory, otherwise fd is a file that will be copied into the block of memory and offset indicates how many bytes into the file the copying should take place.

function munmap(start : system.address; size : long_integer) **return** integer;
pramga import(C, munmap);
Deallocate memory allocated by mmap. Returns -1 on an error.

function mremap(old_start : system.address; old_size : long_integer;
new_size : long_integer; flags : aMapFlag) **return** system.address;
pragma import(C, mremap);
Changes the size of a block of memory allocated by mmap, possibly moving it.

function mprotect(start : system.address; size : long_integer; new_prot :
aProtection) **return** integer;
pragma import(C, mprotect);
Change the protection settings on a block of memory allocated by mmap. Returns -1 on an error.

Other mmap flags:

```
   PROT_NONE     : constant aProtection := 0;     -- shorthand for no access
 PROT_READ     : constant aProtection := 1;     -- read access allowed
 PROT_WRITE    : constant aProtection := 2;     -- write access allowed
 PROT_EXEC     : constant aProtection := 4;     -- execute access allowed

   MAP_SHARED    : constant aMapFlag := 16#01#;   -- share changes with child processes
                                  -- (write changes to the file, if any)
 MAP_PRIVATE   : constant aMapFlag := 16#02#;   -- separate copy for child processes
                                  -- (changes kept in memory, if any)
 MAP_FIXED     : constant aMapFlag := 16#10#;   -- use specified address
 MAP_ANON      : constant aMapFlag := 16#20#;   -- just alloc memory, no related file
 MAP_ANONYMOUS : constant aMapFlag := MAP_ANON; -- another name for MAP_ANON
 MAP_GROWSDOWN : constant aMapFlag := 16#0100#; -- stack-line usage
 MAP_DENYWRITE : constant aMapFlag := 16#0800#; -- write lock the file
 MAP_EXECUTABLE: constant aMapFlag := 16#1000#; -- mark as executable
 MAP_LOCKED    : constant aMapFlag := 16#2000#; -- don't swap out memory
 MAP_NORESERVE : constant aMapFlag := 16#4000#; -- don't check for reservations
```

function msync(start : system.address; size : length;
flags : aMSyncFlag) **return** integer;
pragma import(C, msync);
Updates the file associated with the memory allocated by mmap. Returns -1 on an error.

```
   MS_ASYNC     : constant aSyncFlag := 1;       -- request memory to be saved soon
 MS_INVALIDATE: constant aSyncFlag := 2;       -- mark cache as needing updating
 MS_SYNC      : constant aSyncFlag := 4;       -- save memory to file immediately
```

function mlock(start : system.address; size : long_integer) **return** integer;
pragma import(C, mlock);
Deny page swapping on this block of memory allocated by mmap. Only a superuser process may lock pages. Returns -1 on an error.

function munlock(start : system.address; size : long_integer) **return** integer;
pragma import(C, munlock);
Allow page swapping on this block of memory allocated by mmap. Returns -1 on an error.

function mlockall(flags : aLockFlag) **return** integer;

pragma import(C, mlockall);
Deny swapping on all memory for this process. Only a superuser process can lock memory. Returns -1 on an error.

function munlockall **return** integer;
pragma import(C, mlockall);
Allow swapping on all memory for this process. Returns -1 on an error.

```
   MCL_CURRENT : constant aLockFlag := 1; -- lock current blocks
 MCL_FUTURE  : constant aLockFlag := 2; -- lock subsequent blocks
```

function brk(end_data_segment : system.address) **return** integer;
pragma import(C, brk);
Resize the (Intel) data segment to the specified ending address. Returns -1 on an error.

procedure sbrk(increment : long_integer);
pragma import(C, sbrk);
Increase the (Intel) data segment by the specified number of bytes.

16.25 Exit Procedures

procedure C_exit;
pragma import(C, C_exit, "exit");
pragma import_valued_procedure(C_exit);
exit is a C a standard C library function that closes all your standard C library files and stops your program. *This procedure is meant to be used by C. It is not recommended in an Ada program.*

procedure K_exit;
pragma import(C, K_exit, "_exit");
pragma import_valued_procedure(K_exit);
_exit is a kernel call to stop your program. It leaves any open file open. *Not recommended in an Ada program: there are more effective ways to stop your program.*

16.26 Example: An Ada Daemon

Example by Petr Holub.

```
pragma License (GPL);

with Ada.Exceptions; use Ada.Exceptions;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Command_Line; use Ada.Command_Line;

with Sys_Interface; use Sys_Interface;
with Sys_Interface.Daemon; use Sys_Interface.Daemon;

procedure Daemon is

begin
   Put_Line ("Starting Ada daemon");
   Daemonize;
   Syslog (LOG_NOTICE, "Ada daemon runs as daemon now.");
   delay 120.0;
   Syslog (LOG_NOTICE, "Ada daemon terminates.");
end Daemon;
```

-- Settings provides basic thin intefaces to POSIX/UN*X functions that are needed. For Configurator it supports Linux/FreeBSD only.

```
pragma License (GPL);

with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

package Sys_Interface is

   type OS_Type is (FreeBSD, Linux, Windows);
   Running_OS : OS_Type := FreeBSD;


   -- errno : Integer;
   -- pragma Import (C, errno);
   -- XXX: doesn't work on Linux properly; why? how to do it? GNAT.Os_Lib?

   procedure Perror (Descr : String);

   procedure System (Command : String; Return_Val : out Integer; Signal : out Integer);
   function System (Command : String) return Integer;
   procedure System (Command : in String);

   type Log_Prioriry_Type is (LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR,
      LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG);
   procedure Syslog (Log_Priority : in Log_Prioriry_Type; Text : in String);

   type pid_t is new Integer;
   function Fork return pid_t;
   procedure SetPGID (PID : pid_t; PGRP : pid_t);
   function GetPID return pid_t;
   procedure Wait (PID : out pid_t; Status : out Integer);
   procedure Wait_Pid (PID : out pid_t; PID_or_GID : in pid_t; Status : out Integer; Options : Integer);

   type Signal_Type is (SIGHUP, SIGINT, SIGQUIT, SIGKILL, SIGPIPE, SIGALRM,
      SIGTERM, SIGSTOP, SIGTSTP, SIGCONT, SIGINFO, SIGUSR1, SIGUSR2);
   function Kill (PID : pid_t; Signal : Signal_Type) return Integer;
   function Kill_PG (PID : pid_t; Signal : Signal_Type) return Integer;

private

   for Log_Prioriry_Type use (LOG_EMERG => 0, LOG_ALERT => 1, LOG_CRIT => 2,
      LOG_ERR => 3, LOG_WARNING => 4, LOG_NOTICE => 5, LOG_INFO => 6,
      LOG_DEBUG => 7);
   for Log_Prioriry_Type'Size use Integer'Size;

   for Signal_Type use (SIGHUP => 1, SIGINT => 2, SIGQUIT => 3, SIGKILL => 9,
      SIGPIPE => 13, SIGALRM => 14, SIGTERM => 15, SIGSTOP => 17,
      SIGTSTP => 18, SIGCONT => 19, SIGINFO => 29, SIGUSR1 => 30,
      SIGUSR2 =7gt; 31);
   for Signal_Type'Size use Integer'Size;

end Sys_Interface;
```

```
with Ada.Unchecked_Conversion;
with Interfaces.C.Strings; use Interfaces.C.Strings;
with Interfaces.C; use Interfaces.C;
```

```
with Interfaces; use Interfaces;
with Interfaces.C_Streams; use Interfaces.C_Streams;

package body Sys_Interface is

-- Imports of system functions

   procedure C_perror (descr : chars_ptr);
   pragma Import (C, C_perror, "perror");

   function C_system (command : chars_ptr) return Integer;
   pragma Import (C, C_system, "system");

   procedure C_syslog (Log_Priority : in Integer; Format : in chars_ptr; Text : in chars_ptr);
   pragma Import (C, C_syslog, "syslog");

   function C_fork return pid_t;
   pragma Import (C, C_fork, "fork");

   procedure C_wait (pid : out pid_t; status : out Integer);
   pragma Import (C, C_wait, "wait");
   pragma Import_Valued_Procedure (C_wait);

   procedure C_waitpid (pid : out pid_t; pid_or_gid : in pid_t; status : out Integer; options : Integer);
   pragma Import (C, C_waitpid, "waitpid");
   pragma Import_Valued_Procedure (C_waitpid);

   procedure C_getpid (PID : out pid_t);
   pragma Import (C, C_getpid, "getpid");
   pragma Import_Valued_Procedure (C_getpid);

   function C_kill (PID : pid_t; Signal : Integer) return Integer;
   pragma Import (C, C_kill, "kill");

   function C_killpg (PID : pid_t; Signal : Integer) return Integer;
   pragma Import (C, C_killpg, "killpg");

   procedure C_setpgid (PID : pid_t; PGRP : pid_t);
   pragma Import (C, C_setpgid, "setpgid");

-- Sysinterface body

   function LPT_To_Int is new Ada.Unchecked_Conversion
      (Source => Log_Prioriry_Type, Target => Integer);

   function Signal_Type_To_Int is new Ada.Unchecked_Conversion
      (Source => Signal_Type, Target => Integer);

   procedure Perror (Descr : in String) is
      C_descr : chars_ptr;
   begin
      C_descr := New_String (Descr);
      C_perror (C_descr);
      Free (C_descr);
   end Perror;

   procedure System (Command : String; Return_Val : out Integer; Signal : out Integer) is
      C_command : chars_ptr;
      C_Ret_Val : Integer;
   begin
```

```
   C_command := New_String (Command);
   C_Ret_Val := C_system (C_command);
   Return_Val := Integer (Shift_Right (Unsigned_32 (C_Ret_Val), 8));
   --  Return_Val = -1 .... fork or waitpid failed
   --  Return_Val = 127 ... missing binary to be executed
   Signal := Integer (Unsigned_32 (C_Ret_Val) and 255);
   Free (C_command);
   return;
end System;

function System (Command : String) return Integer is
   Ret_Val : Integer;
   Signal : Integer;
begin
   System (Command, Ret_Val, Signal);
   return Ret_Val;
end System;

procedure System (Command : in String) is
   Ret_Val : Integer;
begin
   Ret_Val := System (Command);
   pragma Unreferenced (Ret_Val);
end System;

procedure Syslog (Log_Priority : in Log_Prioriry_Type; Text : in String) is
   Format_String : constant String := "%s";
   C_Format : chars_ptr;
   C_Text : chars_ptr;
begin
   C_Format := New_String (Format_String);
   C_Text := New_String (Text);
   C_syslog (LPT_To_Int (Log_Priority), C_Format, C_Text);
   Free (C_Format);
   Free (C_Text);
end Syslog;

function Fork return pid_t is
begin
   return C_fork;
end Fork;

procedure SetPGID (PID : pid_t; PGRP : pid_t) is
begin
   C_setpgid (PID, PGRP);
end SetPGID;

procedure Wait (PID : out pid_t; Status : out Integer) is
begin
   C_wait (PID, Status);
end Wait;

procedure Wait_Pid (PID : out pid_t; PID_or_GID : in pid_t; Status : out Integer; Options : Integer) is
begin
   C_waitpid (PID, PID_or_GID, Status, Options);
end Wait_Pid;

function GetPID return pid_t is
   PID : pid_t;
```

```
   begin
      C_getpid (PID);
      return PID;
   end GetPID;

   function Kill (PID : pid_t; Signal : Signal_Type) return Integer is
   begin
      return C_kill (PID, Signal_Type_To_Int (Signal));
   end Kill;

   function Kill_PG (PID : pid_t; Signal : Signal_Type) return Integer is
   begin
      return C_killpg (PID, Signal_Type_To_Int (Signal));
   end Kill_PG;

   function C_popen (Command, Mode : String)  return FILEs;
   pragma Import (C, C_popen, "popen");

   procedure C_pclose (Result : out Integer; FID : FILEs);
   pragma Import (C, C_pclose, "pclose");
   pragma Import_Valued_Procedure (C_pclose);

end Sys_Interface;
pragma License (GPL);
```

```
package Sys_Interface.Daemon is

   procedure Daemonize;

end Sys_Interface.Daemon;
```

```
package body Sys_Interface.Daemon is

   procedure C_daemonize;
   pragma Import (C, C_daemonize, "daemonize");

   procedure Daemonize is
   begin
      C_daemonize;
   end Daemonize;

end Sys_Interface.Daemon;
```

```
#include
#include
#include

void daemonize(void)
{
   int i,n = getdtablesize();

   if (chdir("/")) {
      perror("chdir to /");
      exit(EXIT_FAILURE);
```

```
    }
    for (i=0; i<n; i++)
        (void) close(i);

        switch (fork()) {
            case -1:
                perror("daemonize fork");
                exit(EXIT_FAILURE);
            case 0:
                if(setsid() == -1) {
                        perror("setsid");
                        exit(EXIT_FAILURE);
                }
                break;
            default:
                exit(EXIT_SUCCESS);
        }
}
```

17 Translating Programs To Ada

17.1 c2ada: Translating Your Programs

c2ada is a C to Ada translator built for Linux. It was created by Intermetrics and was released into the Ada community. The Linux version is available fromhttp://www.12000.org/my_notes/ada/index.htm.

17.2 Interfaces.C package

Ada Package	*Description*	*C Equivalent*
int	C integer	int
unsigned	C unsigned integer	unsigned
char_array(n)	C character array	char [n]
long_long	C long long	long long
etc		

The **Interfaces.C** and **Interfaces.C.Extensions** packages provide basic type definitions and conversions functions for C programs.

Gnat 3.12 introduces a new boolean type, C_bool, which behaves as a proper C boolean value: 0 is false and any other value is true.

One thing to remember about this package is that C strings are defined as an array of characters, and Ada will raise a CONSTRANT_ERROR exception if two arrays of characters are not exactly equal length, even if a smaller array is being assigned to a larger one. For example,

```
s : char_array( 1..80 ) := To_C( "Fred Smith" ); -- bad
```

This example will raise the exception because the string is 11 characters long (10 characters plus a null character), but the array being assigned to is 80 characters. You can get around these kind of errors with dynamic allocation.

The following program demonstrates some of the types and functions in the Interfaces.C packages.

```
with text_io, unchecked_deallocation,
Interfaces.C.Extensions;


use text_io, Interfaces.C, interfaces.C.Extensions;

procedure ctest is

-- my types
--
-- pointer to C string and deallocation procedure for same

type stringptr is access all char_array;

procedure free is new
unchecked_deallocation( char_array, stringptr );

-- types from standard Ada package Interfaces.C
```

```
i  : int;          -- integer
u  : unsigned;     -- unsigned integer
l  : long;         -- long
ul : unsigned_long; -- unsigned long
c  : char;         -- a character
sp : stringptr;    -- ptr to a string (array of characters)
f  : C_float;      -- a float
d  : double;       -- a double
wc : wchar_t;      -- 16-bit wide character

-- additional types from Gnat package Interfaces.C.Extensions

ll : long_long;    -- long long
ull: unsigned_long_long; -- unsigned long long
vp : void_ptr;     -- void pointer

begin

   Put_Line( "This is an example of Interfaces.C" );
   New_Line;

   sp := new char_array'( To_C( "This is a string" ) );
   Put_Line( "The C string s is '" & To_Ada( sp.all ) &
      "'." );
   Free( sp );

end ctest;
```

```
This is an example of Interfaces.C

The C string s is 'This is a string'.
```

17.3 Interfaces.C.Pointers Package

One C feature that Ada programs lack is pointer arithmetic. In C, you can move pointers forward and backwards through an array by using simple arithmetic operations. For example, adding two to an character pointer move the pointer two characters forward in a string. Decrementing an integer pointer moves the pointer back one index position in an integer array.

Since pointer arithmetic is important in many C programs, especially sorts, Ada 95 provides a standard generic package called Interfaces.C.Pointers which implements access types that can use pointer arithmetic.

To instantiate the package, you need to specify the elements that will be in your arrays, an unbounded array that will contain the elements, the range of index values, and a default terminator value used by some of the package's subprograms.

For example, to create C-style pointers for the unbounded Char_Array (C string) type in Interfaces.C, you could instantiate the package with

```
package StringPtrs is new Interfaces.C.Pointers(
      Index => size_t, -- the index range is size_t
      Element => char, -- the array contains chars
      Element_Array => char_array, -- the unbounded type
      Default_Terminator => char'val( 0 ) -- the terminator value, ASCII.NUL
);
use StringPtrs; -- need this to make + and - visible
```

```
strptr : StringPtrs.Pointer;
```

The use clause is very important. Without it, the arithmetic operators would not be directly available because they would be hidden inside the StringPtrs package.

Pointers created using Interfaces.C.Pointers are access types, and can be used like any other access type.

```
Put_Line( "strptr is pointing to the character " & strptr.all );
```

However, unlike other access types, they have new pointer arithmetic features. Addition and subtraction is performed the same way as in C by specifying how many positions in the array to move. To move strptr ahead 2 index positions in a string, add 2 to it:

```
strptr := strptr + 2;
```

Since Ada has no increment or decrement operators, two procedures are provided to move a pointer forward or backward by one array position:

```
Increment( strptr );  -- forward one position
Decrement( strptr );  -- back one position
```

The package also makes four additional subprograms available: The **Virtual_Length** function returns the length of an array, up to end of the array or until a terminator is found. [If no terminator, do you get a storage error?--KB] The **Value** function returns a slice from the array, from the position of the pointer to the end of the array or until a terminator is found. It can also slice a specific number of elements from an array.

Copy_Array copies a slice of a specific number of elements from one pointer to another. **Copy_Terminated Array** copies a slice from the pointer position until a terminator is found.

The following program demonstrates C pointers to integer arrays.

```
with Ada.Text_IO, Interfaces.C.Pointers;

use Ada.Text_IO;

procedure point is

-- To use Interfaces.C.Pointers, you need to define an unbounded
-- array type. In this case, we'll create an unbounded array
-- called IntegerArrays with a maximum index range of 1 to 9.
-- BiggestArray is the largest IntegerArrays array possible,
-- with an index range of 1 to 9.  IntegerArrays must have
-- aliased elements because we will be accessing them with an
-- access type.

subtype PointerRange is integer range 1..9;

type IntegerArrays is array PointerRange range <> )
   of aliased integer;

type BiggestArray is new IntegerArrays( PointerRange );

package IntPtrs is new Interfaces.C.Pointers(
   Index => PointerRange, -- the index range
   Element => Integer, -- what the array contains
   Element_Array => IntegerArrays, -- the unbounded type
```

```
    Default_Terminator => 0 ); -- the terminator value

use IntPtrs; -- need this to make + and - visible

procedure ShowArray( ia : IntegerArrays ) is

-- show the contents of any IntegerArrays array

begin

    for i in ia'first..ia'last-1 loop
        Put( i'img );
        Put( " =>" );
        Put( ia( i )'img );
        Put( "," );

    end loop;

    Put( ia'last'img );
    Put( " => " );
    Put_Line( ia( ia'last )'img );
end ShowArray;

ia, ia2 : BiggestArray;  -- two integer arrays
ip, ip2 : IntPtrs.Pointer; -- two pointers to integer arrays

begin

    Put_Line( "This program demonstrates C-style pointers provided" );
    Put_Line( "by Interfaces.C.Pointers" );
    New_Line;

    -- initialize and display the contents of the array

    for i in PointerRange'first..PointerRange'last-1 loop
        ia( i ) := i*2;
    end loop;

    ia( PointerRange'last ) := 0;
    Put_Line( "The array is: " );
    ShowArray( IntegerArrays( ia ) );
    -- must typecast ia because ShowArray is expecting an IntegerArrays

    Put_Line( "Zero is our terminator in this example" );
    New_Line;

    -- set the pointers to the first elements in the arrays

    ip := ia( ia'first )'access;
    ip2 := ia2( ia'first )'access;

    -- ip works like a normal access type

    Put_Line( "Our pointer is set to first position in the array");
    Put_Line( "The element is " & ip.all'img );
    New_Line;

    -- increment example

    Increment( ip );
    Put_Line( "Incrementing the pointer, it now points at " &
```

```
    ip.all'img );
New_Line;

-- decrement example

Decrement( ip );
Put_Line( "Decrementing the pointer, it now points at" &
    ip.all'img );
New_Line;

-- addition example

ip := ip + 3;
Put_Line( "Addition moves the pointer forward." );
Put_Line( "Moving forward three elements, it now points at"
    & ip.all'img );
New_Line;

-- subtraction example

ip := ip - 2;
Put_Line( "Subtraction moves the pointer backwards." );
Put_Line( "Moving backwards two elements, it now points at"
    & ip.all'img );
New_Line;

-- Virtual_Length examples

Put_Line( "Virtual_Length gives the length from the pointer to the" );

Put_Line( "default terminator.  The length from this position is" &
Virtual_Length( ip )'img & " positions" );
Put_Line( "Virtual_Length can also use an arbitrary terminator." );
Put_Line( "The length from the pointer to the first 14 is" &
Virtual_Length( ip, 14 )'img & " positions" );
New_Line;

-- Value examples

Put_Line( "Value returns the array slice from the pointer position to" );
Put_Line( "the terminator.  The array value from this position is" );
ShowArray( Value( ip ) );

Put_Line( "Value can also return a slice of a specific length." );
Put_Line( "The next four elements are" );
ShowArray( Value( ip, Length => 4 ) );
New_Line;

-- Copy_Terminated_Array example

Put_Line( "Our second array contains" );
ShowArray( IntegerArrays( ia2 ) ); -- must typecast here
New_Line;

Put_Line( "Copy_Terminated_Array copies elements from one pointer to" );
Put_Line( "another, up to and including the terminator.  Copying to" );
Put_Line( "the second array " );
Copy_Terminated_Array( ip, ip2 );
ShowArray( IntegerArrays ( ia2 ) ); -- must typecast here
New_Line;
```

```
    -- Copy_Array example

    Put_Line( "Copy_Array copies a specific number of elements.");
    Put_Line( "Copying 4 elements from 3 positions ahead, the new");
    Put_Line( "array contains" );
    Copy_Array( ip+3, ip2, 4 );
    ShowArray( IntegerArrays( ia2 ) ); -- must typecast here
    New_Line;

end point;
```

```
This program demonstrates C-style pointers provided
by Interfaces.C.Pointers

The array is:

1 => 2, 2 => 4, 3 => 6, 4 => 8, 5 => 10, 6 => 12, 7 => 14, 8 => 16, 9 => 0

Zero is our terminator in this example

Our pointer is set to first position in the array

The element is 2

Incrementing the pointer, it now points at 4

Decrementing the pointer, it now points at 2

Addition moves the pointer forward.
Moving forward three elements, it now points at 8

Subtraction moves the pointer backwards.
Moving backwards two elements, it now points at 4

Virtual_Length gives the length from the pointer to the
default terminator.  The length from this position is 7 positions

Virtual_Length can also use an arbitrary terminator.
The length from the pointer to the first 14 is 5 positions

Value returns the array slice from the pointer position to
the terminator.  The array value from this position is

1 => 4, 2 => 6, 3 => 8, 4 => 10, 5 => 12, 6 => 14, 7 => 16, 8 => 0

Value can also return a slice of a specific length.

The next four elements are
1 => 4, 2 => 6, 3 => 8, 4 => 10

Our second array contains

1 => 12, 2 => 134531961, 3 => 12, 4 => 1, 5 => 134560412, 6 => 134891560, 7 => 134575980, 8 => 0, 9 =>
134560432

Copy_Terminated_Array copies elements from one pointer to another, up to
and including the terminator.
Copying to the second array
```

1 => 4, 2 => 6, 3 => 8, 4 => 10, 5 => 12, 6 => 14, 7 => 16, 8 => 0, 9 => 134560432

Copy_Array copies a specific number of elements.

Copying 4 elements from 3 positions ahead, the new array contains
1 => 10, 2 => 12, 3 => 14, 4 => 16, 5 => 12, 6 => 14, 7 => 16, 8 => 0, 9 => 134560432

17.4 Interfaces.C_Streams package

Ada Package	*Description*	*C Equivalent*
fopen	Open a text file (C stream)	fopen
fclose	Close a text file (C stream)	fclose
fread	Read bytes from a text file (C stream)	fread
etc.		

Although the basic Ada types are identical to their C counterparts, the IO libraries are not guaranteed to write data in a format that is readable from other languages. Text files are fine, but to write binary files that can be accessed by C, you'll need to read and write the files using C file handing libraries.

The **Interfaces.C_Streams** package provides a thin binding to the C stdio library. This is comparable to the gnat.os_lib library, but the binding is "thinner" and covers all C stream operations. Some stdio library functions aren't covered because they can't be represented by Ada. Gnat guarantees these functions will be available, no matter what platform gnat is running under, even if it isn't UNIX-based.

It is also possible to call stdio directly. See the discussion above.

c_streams uses "stream" to refer to a Linux text file.

procedure clearerr(stream : FILEs);
Clear any error associated with the stream.

function fclose(stream : FILEs) **return** int;
Close a stream.

function fdopen(handle : int; mode : chars) **return** FILEs;
Open a stream by a handle (UNIX file descriptor).

function feof(stream : FILEs) **return** int;
Check for the end of stream.

function ferror(stream : FILEs) **return** int;
Return any error associated with the last stream operation.

functionfflush(stream : FILEs) **return** int;
Finish writing any outstanding data to the stream.

function fgetc(stream : FILEs) **return** int;
Read one character from the stream. Characters will be ASCII values between 0 and 255, and can be converted to a character with character'val.

function fgets(strng : chars; n : int; stream : FILEs) **return** chars;
Read a string from the stream. Note this is not an Ada string.

function fileno(stream : FILEs) **return** int;
Return the fine number associated with a stream for use with standard Linux file operations.

function fopen(filename : chars; Mode : chars) **return** FILEs;
Open a stream.

function fputc(C : int; stream : FILEs) **return** int;
Write one character to a stream. Convert the character to an integer using character'val.

function fputs(Strng : chars; Stream : FILEs) **return** int;
Write a string of characters to the stream.

function fread(buffer : voids; size : size_t; count : size_t; stream : FILEs) **return** size_t;
Read count bytes into a buffer of length size and return number of bytes actually read.

function freopen(filename : chars; mode : chars; stream : FILEs) **return** FILEs;
Reopen the stream with a new mode.

function fseek(stream : FILEs; offset : long; origin : int) **return** int;
Move offset bytes from the specified origin point.

function ftell(stream : FILEs) **return** long;
Get stream offset for fseek.

function fwrite(buffer : voids; size : size_t; count : size_t; stream : FILEs) **return** size_t;
Write count bytes from a buffer of count length and return the number of bytes actually written.

function isatty(handle : int) **return** int;
[NQS--determine if stream is a TTY device?--KB]

procedure mktemp(template : chars);
Create a random name for a temporary file.

procedure rewind(stream : FILEs);
Move to the start of the stream.

function setvbuf(stream : FILEs; buffer : chars; mode : int; size : size_t) **return** int;
[NQS--used to know what this did--KB]

procedure tmpnam(string : chars);
[Difference with tmpnam?--KB]
The parameter must be a pointer to a string buffer of at least L_tmpnam bytes (the call with a null parameter is not supported).

function tmpfile **return** FILEs;
[NQS--KB]

function ungetc(c : int; stream : FILEs) **return** int;
Back up one character in the stream.

function unlink(filename : chars) **return** int;
Delete a stream file.

The following are related utility functions added by ACT. They are not standard UNIX functions like the above.

function file_exists(name : chars) **return** int;
Returns 0 if a file doesn't exist, 1 if it does.

function is_regular_file(handle : int) **return** int;
Return 1 if given handle is for a regular file, or 0 for some other kind of file.

procedure set_binary_mode(handle : int);
Read text without translation. Only works if compiled with text_translation_required.

procedure set_text_mode(handle : int);
Translate text. Only works if compiled with text_translation_required.

procedure full_name(nam : chars; buffer : chars);
Return the full path of a file as a C string.

The following program demonstrates some of the c_stream functions.

```
with text_io, unchecked_deallocation, Interfaces.C_Streams;
use text_io, Interfaces.C_Streams;

procedure cstreamtest is

   fd : FILEs;
   line2write : constant string := "This is a test";
   cline2write: constant string := "This is a test" &
      ASCII.NUL;
   path : constant string := "testfile.xxx";
   cpath : constant string := path & ASCII.NUL;
   fileMode : constant string := "w";

   amountWritten : size_t;
   result : int;

begin

   Put_Line( "This is an example of Interfaces.C_Streams" );
   New_Line;

   fd := fopen( cpath'address, fileMode'address );
   if ferror( fd ) = 0 then
      Put_Line( "Opened " & path & " with fopen");
      Put_Line( "Writing '" & line2write & "' with fwrite" );
      amountWritten := fwrite( line2write'address, -- what to write
                               1, -- size of elements
                               line2write'length, -- how many to write
                               fd ); -- to which file

      Put_Line( "Wrote" & amountWritten'img & " characters" );
      New_Line;
      Put_Line( "Writing with fputs" );
      result := fputs( cline2write'address, fd );
      Put_Line( "Result was" & result'img );
      New_Line;

      result := fclose( fd );
      Put_Line( "Closed " & path & " with fclose");
      Put_Line( "Result was" & result'img );
      New_Line;

      result := unlink( cpath'address );
      Put_Line( "Deleted " & path & " with unlink");
      Put_Line( "Result was" & result'img );
   end if;

end cstreamtest;
```

```
This is an example of Interfaces.C_Streams

Opened testfile.xxx with fopen
Writing 'This is a test' with fwrite
Wrote 14 characters
```

```
Writing with fputs
Result was 1

Closed testfile.xxx with fclose
Result was 0

Deleted testfile.xxx with unlink
Result was 0
```

17.5 Ada and C Files

Gnat provides several interfacing packages to allow Ada to read and write C files. These are a "thicker binding" than Interfaces.C_Streams.

- Ada.Direct_IO.C_Streams - a variation of Ada.Direct_IO of reading and writing C direct files.
- Ada.Sequential_IO.C_Streams - a variation of Ada.Sequential_IO for reading and writing C sequential files
- Ada.Streams_Stream_IO.C_Streams - a package for reading and writing C streams
- Ada.Text_IO.C_Streams - a package for reading and writing C text files
- Ada.Wide_Text_IO.C_Streams - a package for reading and writing 16-bit character C text files

17.6 A Word on Interfaces.Fortran

Gnat provides interfacing packages for languages besides C. *Interfaces.Fortran* contains types and subprograms to link Fortran language programs to your Ada programs. The GCC Fortran 77 compiler is g77.

As with gcc, most of the Fortran data types correspond identically with an Ada type. A Fortran real variable, for example, is the same as an Ada float, and adouble precision variable is an Ada long_float. Other Ada compilers may not do this: if portability is an issue, always use the types of Interfaces.Fortran.

Gnat 3.12 introduces a proper Fortran logical type that behaves according to Fortran semantics.

Fortran subprograms may be imported into Ada using pragma import:

```
procedure MyFortranSubroutine;
pragma import( Fortran, MyFortranSubroutine );
```

Variables may be likewise imported.

```
RealVar : float;
pragma import( Fortran, RealVar );
```

g77 adds an undescore to subroutine names, so ifyou are importing from g77 you'll need to include the name of the subroutine with a trailing underscore in pragma import.

```
pragma import (Fortran, MyFortranSubroutine, "MyFortranSubroutine_");
```

17.7 P2Ada - Pascal to Ada

There is a Pascal to Ada translator called P2Ada. It supports Turbo/Borland/Delphi, CodeWarrior, ISO Pascal. It's available fromhttp://homepage.sunrise.ch/mysunrise/gdm/gsoft.htm

18 Data Structures

Good programmers write good programs. Great programmers write good programs and good data structures. Organizing your data is as important as the program that crunches the data and produces a result.

Unfortunately, my experiences in the corporate world have taught me that that the only data structure used is the single dimensional array. When results are the only goal and more processing power is the cure for bad software design, arrays are easy to implement (they are built into Ada). Even the worst programmer knows how to use an array. And arrays are easy to understand. Try to use a linked list, and a programmer can get into trouble with his boss for using risky, "advanced" technology.

Alternatively, programmers will sometimes rely on the complexity and overhead of databases when a simplier solution using the correct data structure would be faster and easier to implement.

If you are lucky enough to work for a company that uses more than arrays, this chapter will discuss how to use other kinds of data structures in Ada.

18.1 Using the Booch Components

Like Ada, C++ has no advanced data structures built into the language. To provide a standard set of data structures, what is now called the Standard Template Library was developed to provide the tools necessary to organize most types of data.

Perhaps because of an oversight, Ada 95 with all its annexes has no equivalent to the C++ Standard Template Library. (Ada 2005 has a data strucutre library.) There are no standard packages providing common data structures. The Gnat compiler fills part of this void with packages for creating simple tables and hash tables.

The Booch components are a set of C++ objects created by Grady Booch. These were later ported to Ada 95. The components contain sets of general purpose data structures. The Booch components are available from AdaPower.Net or in RPM format from the Ada Linux Team. This is one popular choice for Ada's unofficial "Standard Template Library".

The components are organized into three main categories: tools, support and structs. The tools cover many items already implemented in the standard Ada or Gnat packages, such as searching, sorting and pattern recognition. Support refers to components that implement the tools and structs.

The structs (data structures) are the primary interest of Ada programmers. These are further subcategorized by the user's requirements: bounded (where the size is known at compile-time or there's no heap allocation), unbounded (using dynamic allocation and item caching), or the dynamic (a compromize between bounded and unbounded). The default if no others are available is unbounded.

Dynamic and unbounded types can specify a storage manager to use. The storage manager is a program that allocates memory. Use Global_Heap package if you have no preference.

Unbounded structures allocate memory whenever a new item is added to the structure.

Dynamic structures allocate memory in fixed-sized chunks. Each chunk is large enough for several items. The chunk size is set when the dynamic data structure is first created, but it can be altered at any time. When a chunk is full, the structure is grows by the size of another chunk. This reduces the number of memory allocations to improve performance.

Each dynamic structure includes these subprograms:

- **Create** - Change the chunk size for the collection
- **Set_Chunk_Size** - Change the chunk size for the collection
- **Preallocate** - Increase the size of the collection immediately
- **Chunk_Size** - Returns the current chunk size

The Booch components are organzied in a hierarchy of packages. The BC package is the top-most package. BC defines the basic execptions that can be raised by the various components:

```
Container_Error : exception;
Duplicate : exception;
Illegal_Pattern : exception;
Is_Null : exception;
Lexical_Error : exception;
Math_Error : exception;
Not_Found : exception;
Not_Null : exception;
Not_Root : exception;
Overflow : exception;
Range_Error : exception;
Storage_Error : exception;
Synchronization_Error : exception;
Underflow : exception;
Should_Have_Been_Overridden : exception;
Not_Yet_Implemented : exception;
```

The data structure components are:

Data Structure	*Booch Packages*	*Description*
Bags	`bc-containers-bags-bounded bc-containers-bags-dynamic bc-containers-bags-unbounded`	Unordered collection of items. Duplicates are counted but not actually stored.
Collections	`bc-containers-collections-bounded bc-containers-collections-dynamic bc-containers-collections-unbounded`	Ordered collection of items. Duplicates are allowed and stored.
Deques	`bc-containers-deques-bounded bc-containers-deques-dynamic bc-containers-deques-unbounded`	Double-ended queues
Single linked Lists	`bc-containers-lists-single`	A sequence of 0 or more items with a head and a pointer to each successive item.

Double linked Lists	`bc-containers-lists-double`	A sequence of 0 or more items with a head and a pointer to both successive and previous items.

Maps	`bc-containers-maps-bounded`
	`bc-containers-maps-dynamic`
	`bc-containers-maps-unbounded`
		A set with relationships between pairs of items.

Queues	`bc-containers-queues-bounded`
	`bc-containers-queues-dynamic`
	`bc-containers-queues-unbounded`
		First in, first out list.

Ordered (Priority) Queues	`bc-containers-queues-ordered-bounded`
	`bc-containers-queues-ordered-dynamic`
	`bc-containers-queues-ordered-unbounded`
		A sorted list, items removed from the front.

Rings	`bc-containers-rings-bounded`
	`bc-containers-rings-dynamic`
	`bc-containers-rings-unbounded`
		A deque with only one endpoint.

Sets	`bc-containers-sets-bounded`
	`bc-containers-sets-dynamic`
	`bc-containers-sets-unbounded`
		Unordered collection of items. Duplicates are not allowed.

Stacks	`bc-containers-stacks-bounded`
	`bc-containers-stacks-dynamic`
	`bc-containers-stacks-unbounded`
		Last in, first out list.

AVL Trees	`bc-containers-trees-avl`	Balanced binary trees

Binary Trees	`bc-containers-trees-binary-in_order`
	`bc-containers-trees-binary-post_order`
	`bc-containers-trees-binary-pre_order`
		A list with two successors per item.

Multiway Trees	`bc-containers-trees-multiway-post_order` `bc-containers-trees-multiway-pre_order`	Tree with an arbitrary number of children.
Directed Graphs	`bc-graphs-directed`	Groups of items with one-way relationships
Undirected Graphs	`bc-graphs-undirected`	Groups of items with two-way relationships
Smart Pointers	`bc-smart`	Access types that

		automatically deallocate themselves

A definition of common data structures can be found at the National Institute of Standards and Technology.

The components are generic packages and must be instantiated for a particular type. They are arranged in hierarchies of generic packages. Each parent package must be instantiated before its child. For example, to use single linked lists (bc.containers.lists.single), bc.containers, bc.containers.lists, and bc.containers.lists.single must all be be created for the item type.

As with many component libraries, the Booch components represent all structures in memory, not in long-term storage. They cannot be used to create disk files, although the data could be saved to disk and reloaded later.

18.1.1 Containers

Containers form the cornerstone of the Booch components.

Containers are a controlled tagged record that encloses an item. The Booch components are composed of items stored in containers that are arranged in different ways.

To use any of the Booch components, a container must be instantiated to hold your item. For example, to create a new package to manage character in containers, use

```
package charContainers is new BC.Containers (Item => Character);
```

18.1.2 Iterators

The Container package also manages the iterators used by the Booch components. An iterator is a variable that keeps track of the position in a data structure during a traversal.

Iterators are created by New_Iterator in a data structure's package, but the subprograms that work with the iterator are defined in the Container package.

- **Reset** - start a new traversal at the first item
- **Next** - continue to another item in the component
- **Is_Done** - True if there are no more items
- **Current_Item** - return the current item

The Is_Done function indicates when all items have been traversed. When Is_Done is true, Current_Item is undefined. In other words, the program must loop through all items in the list, plus 1, before Is_Done is true.

Because an Iterator is a class-wide type, it must be assigned a new value when it is declared to avoid a compiler error.

```
i : charContainers.Iterator'class := charList.New_Iterator( customers );
```

18.1.3 Single linked Lists

Creating a single linked list requires the instantiation of 3 separate generic packages: BC.Containers, BC.Containers.Lists, and BC.Containers.Lists.Single. To avoid problems with access types, these should be declared globally (that is, in a package spec).

First, a container must be defined to hold the item you want to store in your linked list.

```
package Containers is new BC.Containers (Item => Character);
```

Second, basic operations on lists must be instantiated.

```
package Lists is new Containers.Lists;
```

Finally, the single linked list package must be instantiated. For an unbounded package, you chose a storage pool to use. Single linked lists are always unbounded. Use Global_Heap if you have no preference.

```
package LS is new Lists.Single (Storage_Manager => Global_Heap.Pool,
                                Storage => Global_Heap.Storage);
```

The single linked list package provides the following subprograms:

- **Clear** - destroy the list
- **Insert** - add an item to the list
- **Append** - add an item to the end of the list
- **Remove** - remove an item from the list
- **Purge** - remove consecutive items
- **Preserve** - inverse of Purge, keep consecutive items, removing the rest
- **Share/_Head/_Foot** - make the list an alias for a sublist in another list
- **Tail** - discard everything but the last item in the list
- **Length** - return the number of items in the list
- **Is_Null** - true if the list has no items
- **Is_Shared** - true if there an alias to a sublist in the list
- **Head** - return the top-most item
- **Foot** - return the last item in the list
- **Item_at** - return an item at a given position
- **New_Iterator** - return an iterator for traversing the list
- **Process_Head/Tail** - generic procedure to return an item with processing
- **Swap_Tail** - NQS-KB

Notice that the term "Foot" refers to the last item in the list. The Ada string packages uses the term "Tail".

Here's an example:

```
with BC.Containers.Lists.Single;
with Global_Heap;

package customers is

   type aCustomer is record
         customerID     : integer;
         accountBalance : float;
   end record;
   -- this is the item to put in the list

   package customerContainers is new BC.Containers (Item => aCustomer);
   -- create a new controlled tagged record container for customers

   package customerLists is new customerContainers.Lists;
   -- create a new list support package for our using container type

   package customerList is new customerLists.Single (Storage_Manager => Global_Heap.Pool, Storage =>
Global_Heap.Storage);
   -- create a single linked list package using the lists support
   -- customized for our container type
```

```
end customers;
```

```
with ada.text_io, BC, customers;
use ada.text_io, BC, customers;

procedure list_demo is
   customers : customerList.Single_List;
   c         : aCustomer;
   i         : customerContainers.Iterator'class := customerList.New_Iterator( customers );
begin
   Put_Line( "This is a demo of the Booch components: single-linked lists" );
   New_Line;

   -- The Newly Declared List

   Put_Line( "The list is newly declared." );
   Put_Line( "The list is empty? " & customerList.Is_Null( customers )'img );
   Put_Line( "The list is shared? " & customerList.Is_Shared( customers )'img );
   Put_Line( "The list length is" & customerList.Length( customers )'img );
   New_Line;

   -- Inserting a customer

   c.customerID := 7456;
   c.accountBalance := 56.74;
   customerList.Insert( customers, c );

   Put_Line( "Added customer" & c.customerID'img );
   Put_Line( "The list is empty? " & customerList.Is_Null( customers )'img );
   Put_Line( "The list is shared? " & customerList.Is_Shared( customers )'img );
   Put_Line( "The list length is" & customerList.Length( customers )'img );

   c := customerList.Head( customers );
   Put_Line( "The head item is customer id" & c.customerID'img );
   c := customerList.Foot( customers );
   Put_Line( "The foot item is customer id" & c.customerID'img );
   New_Line;

   -- Apending a customer

   c.customerID := 9362;
   c.accountBalance := 88.92;
   customerList.Append( customers, c );

   Put_Line( "Appended customer" & c.customerID'img );
   Put_Line( "The list length is" & customerList.Length( customers )'img );
   c := customerList.Head( customers );
   Put_Line( "The head item is customer id" & c.customerID'img );
   c := customerList.Foot( customers );
   Put_Line( "The foot item is customer id" & c.customerID'img );
   New_Line;

   -- Iterator example

   Put_Line( "Resetting the iterator.." );
   customerContainers.Reset( i );
   c := customerContainers.Current_item ( i );
   Put_Line( "The current item is customer id" & c.customerID'img );
```

```
    Put_Line( "Are we done? " & customerContainers.Is_Done( i )'img );

    Put_Line( "Advancing to the next item..." );
    customerContainers.Next( i );
    c := customerContainers.Current_item ( i );
    Put_Line( "The current item is customer id" & c.customerID'img );
    Put_Line( "Are we done? " & customerContainers.Is_Done( i )'img );

    Put_Line( "Advancing to the next item..." );
    customerContainers.Next( i );
    Put_Line( "Are we done? " & customerContainers.Is_Done( i )'img );
    begin
       c := customerContainers.Current_item ( i );
    exception when BC.NOT_FOUND =>
       put_line( "BC.NOT_FOUND exception: no item at this position in the list" );
    end;

end list_demo;
```

```
This is a demo of the Booch components: single-linked lists

The list is newly declared.
The list is empty? TRUE
The list is shared? FALSE
The list length is 0

Added customer 7456
The list is empty? FALSE
The list is shared? FALSE
The list length is 1
The head item is customer id 7456
The foot item is customer id 7456

Appended customer 9362
The list length is 2
The head item is customer id 7456
The foot item is customer id 9362

Resetting the iterator..
The current item is customer id 7456
Are we done? FALSE
Advancing to the next item...
The current item is customer id 9362
Are we done? FALSE
Advancing to the next item...
Are we done? TRUE
BC.NOT_FOUND exception: no item at this position in the list
```

Single linked lists should not be Guarded.

18.1.4 Double linked Lists

Double linked lists are implemented exactly the same as single-linked lists except that the word "Double" is substituted for the word "Single".

Double linked lists are useful for lists that must be browsed backwards and forwards continuously.

Double linked lists should not be Guarded.

18.1.5 Bags

Bags, like linked lists, are collections of items. However, there is no attempt to order the items. Duplicate items can be stored, but the bag keeps a count of duplications to save memory instead of storing copies of the duplicates.

The bags package provides the following subprograms:

- **Are_Equal** - True if two bags have the same contents
- **Clear** - Removes all items from the bag
- **Add** - Adds an item to the bag
- **Remove** - Removes an item from the bag
- **Union** - Add one bag to another
- **Intersection** - Remove all items not common between two bags. Where there are duplicates, keep the lower duplicate count
- **Difference** - Remove all items not common between two bags. Where there are duplicates, subtract from the original and discard the item if the total is >= 0
- **Extent** - Return the number of distinct items
- **Total_Size** - Return the total number of items (including duplicates)
- **Count** - Return the number of occurences of an item in the bag
- **Is_Empty** - True if the bag is empty
- **Is_Member** - True if one or more copies of an item is in the bag
- **Is_Subset** - True if the contents of one bag is completely contained in another
- **Is_Proper_Subset** - Same as Is_Subset, but the bags must not be equal

Bags can be bounded, dynamic or unbounded.

Bags are implemented using a hash table. To declare a bag, a program must provide a hash function for storing items in the bag, and must indicate the size of the hash table.

Here's an example. Notice that some of the subprograms are in the Bags instantiation, and some in the Bags.Unbounded instantiation. Also notice the iterator moves over the items, but not the duplications:

```
with BC.Containers.Bags.Unbounded;
with Global_Heap;

package customers is

   type aCustomerID is new integer range 1_000..9_999;

   function IDHash( id : aCustomerID ) return Positive;
   -- our hash function

   package customerContainers is new BC.Containers (Item => aCustomerID);
   -- create a new controlled tagged record container for customers

   package customerBags is new customerContainers.Bags;
   -- create a new bag support for our using container type

   package customerBag is new customerBags.Unbounded(
                  Hash => IDHash,
                  Buckets => 99,
                  Storage_Manager => Global_Heap.Pool,
                  Storage => Global_Heap.Storage);
   -- create an unbounded bag package holding customer numbers
```

```
end customers;
```

```
package body customers is

   function IDHash( id : aCustomerID ) return Positive is
   -- our hash function
   begin
      return Positive( id ); -- in this case, using the id is good enough
   end IDHash;


end customers;
```

```
with ada.text_io, BC, customers;
use ada.text_io, BC, customers;

procedure bag_demo is
   customers : customerBag.Unbounded_Bag;
   c         : aCustomerID;
   i         : customerContainers.Iterator'class := customerBag.New_Iterator( customers );
begin
   Put_Line( "This is a demo of the Booch components: bags" );
   New_Line;

   -- The Newly Declared Bag

   Put_Line( "The bag is newly declared." );
   Put_Line( "The bag is empty? " & customerBag.Is_Empty( customers )'img );
   Put_Line( "The bag extent is" & customerBag.Extent( customers )'img );
   Put_Line( "The bag total size is" & customerBags.Total_Size( customers )'img );
   New_Line;

   -- Inserting a customer

   c := 7456;
   customerBags.Add( customers, c );

   Put_Line( "Added customer" & c'img );
   Put_Line( "The bag is empty? " & customerBag.Is_Empty( customers )'img );
   Put_Line( "The bag extent is" & customerBag.Extent( customers )'img );
   New_Line;

   -- Inserting another customer

   c := 9362;
   customerBags.Add( customers, c );

   Put_Line( "Added customer" & c'img );
   Put_Line( "The bag is empty? " & customerBag.Is_Empty( customers )'img );
   Put_Line( "The bag extent is" & customerBag.Extent( customers )'img );
   Put_Line( "The bag total size is" & customerBags.Total_Size( customers )'img );
   New_Line;

   -- Inserting duplicate customer

   c := 9362;
```

```
    customerBags.Add( customers, c );

    Put_Line( "Added customer" & c'img );
    Put_Line( "The bag is empty? " & customerBag.Is_Empty( customers )'img );
    Put_Line( "The bag extent is" & customerBag.Extent( customers )'img );
    Put_Line( "The bag total size is" & customerBags.Total_Size( customers )'img );
    New_Line;

    -- Iterator example

    Put_Line( "Resetting the iterator.." );
    customerContainers.Reset( i );
    c := customerContainers.Current_item ( i );
    Put_Line( "The current item is customer id" & c'img );
    Put_Line( "Are we done? " & customerContainers.Is_Done( i )'img );

    Put_Line( "Advancing to the next item..." );
    customerContainers.Next( i );
    c := customerContainers.Current_item ( i );
    Put_Line( "The current item is customer id" & c'img );
    Put_Line( "Are we done? " & customerContainers.Is_Done( i )'img );

    Put_Line( "Advancing to the next item..." );
    customerContainers.Next( i );
    Put_Line( "Are we done? " & customerContainers.Is_Done( i )'img );
    begin
        c := customerContainers.Current_item ( i );
    exception when BC.NOT_FOUND =>
        put_line( "BC.NOT_FOUND exception: no item at this position in the bag" );
    end;

end bag_demo;
```

```
This is a demo of the Booch components: bags

The bag is newly declared.
The bag is empty? TRUE
The bag extent is 0
The bag total size is 0

Added customer 7456
The bag is empty? FALSE
The bag extent is 1

Added customer 9362
The bag is empty? FALSE
The bag extent is 2
The bag total size is 2

Added customer 9362
The bag is empty? FALSE
The bag extent is 2
The bag total size is 3

Resetting the iterator..
The current item is customer id 7456
Are we done? FALSE
Advancing to the next item...
The current item is customer id 9362
```

```
Are we done? FALSE
Advancing to the next item...
Are we done? TRUE
BC.NOT_FOUND exception: no item at this position in the bag
```

Bags are useful for counting the occurrences of an item in a large amount of data.

18.1.6 Sets

Sets are essentially the same as bags but may not contain duplicates. The are useful for detecting the presence/absence of an item, or representing flags or conditions.

```
with BC.Containers.Sets.Bounded;
with Global_Heap;

package fruit_sets is

   -- my grandfather owned one of the largest fruit companies in the world

   type aFruit    is ( Apples, Grapes, Peaches, Cherries, Pears, Plums, Other );

   function FruitHash( f : aFruit ) return Positive;
   -- our hash function for the set

   package fruitContainers is new BC.Containers( item=> aFruit );
   -- basic fruit container

   package fruitSets is new fruitContainers.Sets;
   -- basic set support

   package fruitBoundedSets is new fruitSets.Bounded( fruitHash,
         Buckets => 10,
         Size => 20 );
   -- our actual set is an unbounded set

end fruit_sets;
```

```
package body fruit_sets is

   function FruitHash( f : aFruit ) return Positive is
   begin
      return aFruit'pos( f )+1; -- good enough for this example
   end FruitHash;

end fruit_sets;
```

```
with ada.text_io, kb_sets;
use ada.text_io, kb_sets;

procedure set_demo is
   use fruitSets;
   use fruitBoundedSets;
   s1 : Bounded_Set;
   s2 : Bounded_Set;
   s3 : Bounded_Set;
begin
```

```
    Put_Line( "This is a demo of the Booch components: sets" );
    New_Line;

    Add( s1, apples );
    Add( s1, peaches );
    Add( s2, apples );
    Add( s2, peaches );
    Add( s2, pears );

    Put_Line( "Set 1 has apples and peaches." );
    Put_Line( "Set 2 has apples, peaches and pears." );
    New_Line;
    Put_Line( "Extent of set 1? " & Extent( s1 )'img );
    Put_Line( "Extent of set 2? " & Extent( s2 )'img );
    Put_Line( "Peaches in set 1? " & Is_Member( s1, peaches )'img );
    Put_Line( "Pears   in set 1? " & Is_Member( s1, pears )'img );
    Put_Line( "Set 1 a subset of set 2? " & Is_Subset( s1, s2 )'img );
    Put_Line( "Set 2 a subset of set 1? " & Is_Subset( s2, s1 )'img );
    Put_Line( "Set 1 a subset of set 1? " & Is_Subset( s1, s1 )'img );
    Put_Line( "Set 1 a proper subset of set 1? " & Is_Proper_Subset( s1, s1 )'img );
    New_Line;

    s3 := s1;
    Union( s3, s2 );
    Put_Line( "Set 3 is the union of set 1 and set 2" );
    Put_Line( "Extent of set 3? " & Extent( s3 )'img );
end set_demo;
```

```
This is a demo of the Booch components: sets

Set 1 has apples and peaches.
Set 2 has apples, peaches and pears.

Extent of set 1?  2
Extent of set 2?  3
Peaches in set 1? TRUE
Pears   in set 1? FALSE
Set 1 a subset of set 2? TRUE
Set 2 a subset of set 1? FALSE
Set 1 a subset of set 1? TRUE
Set 1 a proper subset of set 1? FALSE

Set 3 is the union of set 1 and set 2
Extent of set 3?  3
```

18.1.7 Collections

Collections are a (conceptually) combination of lists and bags. Duplicates actually exist as copies in the collection, not simply counted. Collections are also indexed, like a list, so that items can be referenced in the collection.

The Collections package provides the following subprograms:

- **Create** - Create a new collection and its initial chunk
- **Clear** - Remove all items from a collection
- **Insert** - Add an item in front of another
- **Append** - Add an item to the end of the collection

- **Remove** - Remove an item at an index
- **Replace** - Replace an item at an index
- **Length** - Return the number of items in the collection
- **Is_Empty** - True if there are no items in the collection
- **First** - Return the item at the front of the collection
- **Last** - Return the item at the end of the collection
- **Item_At** - Return the item at a particular index
- **Location** - Return the first index where an item is found (0 if

Collections are implemented as dynamically allocated arrays.

```
with BC.Containers.Collections.Dynamic;
with Global_Heap;

package products is

   type aProduct is record
            id : integer;
            weight : float;
   end record;

   package productContainers is new BC.Containers (Item => aProduct);
   -- this is the basic container

   package productCollections is new productContainers.Collections;
   -- create a new collection support for our using container type

   package productCollection is new productCollections.dynamic(
                  Storage_Manager => Global_Heap.Pool,
                  Storage => Global_Heap.Storage);
   -- create a dynamic collection holding products

end products;
```

```
with ada.text_io, BC, products;
use ada.text_io, BC, products;

procedure collection_demo is
   products : productCollection.Dynamic_Collection;
   p        : aProduct;
   i        : productContainers.Iterator'class := productCollection.New_Iterator( products );
begin
   Put_Line( "This is a demo of the Booch components: collections" );
   New_Line;

   products := productCollection.Create( 100 );

   -- The Newly Declared Collection

   Put_Line( "The collection is newly declared with a chunk size of 100..." );
   Put_Line( "The collection is empty? " & productCollection.Is_Empty( products )'img );
   Put_Line( "The collection length is" & productCollection.Length( products )'img );
   Put_Line( "The collection chunk size is" & productCollection.Chunk_Size( products )'img );
   New_Line;

   -- Adding an Item
```

```
p.id := 8301;
p.weight := 17.0;
productCollection.Append( products, p );

Put_Line( "Product id" & p.id'img & " was added..." );
Put_Line( "The collection is empty? " & productCollection.Is_Empty( products )'img );
Put_Line( "The collection length is" & productCollection.Length( products )'img );
Put_Line( "The collection chunk size is" & productCollection.Chunk_Size( products )'img );
p := productCollection.First( products );
Put_Line( "The first item is" & p.id'img );
p := productCollection.Last( products );
Put_Line( "The last item is" & p.id'img );
New_Line;

-- Adding another Item
p.id := 1732;
p.weight := 27.0;
productCollection.Append( products, p );

Put_Line( "Product id" & p.id'img & " was added..." );
Put_Line( "The collection is empty? " & productCollection.Is_Empty( products )'img );
Put_Line( "The collection length is" & productCollection.Length( products )'img );
Put_Line( "The collection chunk size is" & productCollection.Chunk_Size( products )'img );
p := productCollection.First( products );
Put_Line( "The first item is" & p.id'img );
p := productCollection.Last( products );
Put_Line( "The last item is" & p.id'img );
New_Line;

-- Changing the Chunk Size

productCollection.Set_Chunk_Size( products, Size => 1 );

Put_Line( "The chunk size was reduced to only 1..." );
Put_Line( "The collection is empty? " & productCollection.Is_Empty( products )'img );
Put_Line( "The collection length is" & productCollection.Length( products )'img );
Put_Line( "The collection chunk size is" & productCollection.Chunk_Size( products )'img );
p := productCollection.First( products );
Put_Line( "The first item is" & p.id'img );
p := productCollection.Last( products );
Put_Line( "The last item is" & p.id'img );
New_Line;

-- Iterator example

Put_Line( "Resetting the iterator.." );
productContainers.Reset( i );
p := productContainers.Current_item ( i );
Put_Line( "The current item is customer id" & p.id'img );
Put_Line( "Are we done? " & productContainers.Is_Done( i )'img );

Put_Line( "Advancing to the next item..." );
productContainers.Next( i );
p := productContainers.Current_item ( i );
Put_Line( "The current item is customer id" & p.id'img );
Put_Line( "Are we done? " & productContainers.Is_Done( i )'img );

Put_Line( "Advancing to the next item..." );
productContainers.Next( i );
```

```
   Put_Line( "Are we done? " & productContainers.Is_Done( i )'img );
   begin
      p := productContainers.Current_item ( i );
   exception when BC.NOT_FOUND =>
      put_line( "BC.NOT_FOUND exception: no item at this position in the collection" );
   end;
```

Collections are suitable for small lists or lists where the upper bound is known or rarely exceeded.

```
This is a demo of the Booch components: collections

The collection is newly declared with a chunk size of 100...
The collection is empty? TRUE
The collection length is 0
The collection chunk size is 100

Product id 8301 was added...
The collection is empty? FALSE
The collection length is 1
The collection chunk size is 100
The first item is 8301
The last item is 8301

Product id 1732 was added...
The collection is empty? FALSE
The collection length is 2
The collection chunk size is 100
The first item is 8301
The last item is 1732

The chunk size was reduced to only 1...
The collection is empty? FALSE
The collection length is 2
The collection chunk size is 1
The first item is 8301
The last item is 1732

Resetting the iterator..
The current item is customer id 8301
Are we done? FALSE
Advancing to the next item...
The current item is customer id 1732
Are we done? FALSE
Advancing to the next item...
Are we done? TRUE
BC.NOT_FOUND exception: no item at this position in the collection
```

18.1.8 Queues

Queues are a list in which items are removed in the same order they are added. Items are added at the end of the queue and removed at the front.

An ordered (or "priority") queue is a queue in which added items are sorted.

The queues package provides the following subprograms:

- **Clear** - Remove all items from the queue
- **Append** - Add an item to the back of the queue
- **Pop** - Remove an item from the front of the queue and return it

- **Remove** - Remove an item at a particular index
- **Length** - Return the number of items in the queue
- **Is_Empty** - True if there are no items in the queue
- **Front** - Return the item at the front of the queue without removing it
- **Process** - generic procedure to return an item with processing
- **Location** - Return the first index where an item appears else 0
- **Are_Equal** - True if two queues have the same items and length
- **Copy** - Copy one queue to another

An ordered queue is identical except that append adds an item in sorted order.

Queues can be bounded, dynamic or unbounded.

Queues provide "fair" processing and reduce starvation.

18.1.9 Stacks

Stacks are lists in which the last item placed in the list is the first item removed.

The Stacks package provides the following subprograms:

- **Clear** - Remove all items from the stack
- **Push** - Add an item to the top of the queue
- **Pop** - Remove an item from the top of the stack and return it
- **Depth** - Return the number of items in the stack
- **Is_Empty** - True if there are no items in the stack
- **Top** - Return the item at the top of the stack without removing it
- **Process_Top** - generic procedure to return an item with processing
- **Are_Equal** - True if two stacks have the same items and length
- **Copy** - Copy one stack to another

Stacks can be bounded, dynamic or unbounded.

Stacks are used for temporary storage, compact representation and fast data access.

18.1.10 Deques

Deques (double-ended queues, pronounced "deck") are a combination of a stack and queue where items can be placed at the front or the back and removed from either the front or the back.

The Deques package provides the following subprograms:

- **Clear** - Remove all items from the deque
- **Append** - Add an item to the deque
- **Pop** - Remove an item from the deque and return it
- **Remove** - Remove an item at a particular index
- **Length** - Return the number of items in the deque
- **Is_Empty** - True if there are no items in the deque
- **Front** - Return the item at the front of the deque without removing
- **Back** - Return the item at the back of the deque without removing it
- **Process_Front/_Back** - generic procedure to return an item with processing
- **Location** - Return the first index where an item appears else 0
- **Are_Equal** - True if two deques have the same items and length
- **Copy** - Copy one deque to another

Deques can be bounded, dynamic or unbounded.

18.1.11 Rings

Rings are similar to deques, but rings have no "front" or "back", only a moving point of reference called "top".

In addition to the deque subprograms, rings include "Mark" to mark a point in the ring, "Rotate_To_Mark" to move the ring to the marked position, and "At_Mark" to test to see if the top of the ring is at the mark.

Rings can be bounded or dynamic.

18.1.12 Maps

Maps are ordered pairs of related items. Each item is related to a "value" which may or may not be the same type. Maps relate items to values by "binding" them.

The Maps package provides the following subprograms:

- **Clear** - destroy a map
- **Bind** - relate an item to a value
- **Rebind** - relate an item to a different value
- **Unbind** - remove the relationship between an item and its value
- **Extent** - return the number of relationships
- **Is_Empty** - true if there are no relationships
- **Is_Bound** - true if the item is related to a value
- **Value_Of** - the value an item is related to
- **Visit** - a "read-only" procedure to traverse the map
- **Modify** - a procedure that traverses the map making changes

Maps are implemented with a hash table and caching.

Maps can be bounded, dynamic, unbounded or synchronized.

Maps are useful as translation tables.

18.1.13 Binary Trees

Binary trees are lists with two successors instead of 1, named "left" and "right". The items in the tree are not sorted by the Booch component. The program has full control on how items are added to the tree.

Programs "walk" the tree by moving the root of the tree up and down the links to the items. Left_Child follows the left child link. Right_Child follows the right child link. Parent follows the parent link. Each of these subprograms can be used as a procedure (to move the root of the tree) or as a function (to examine the item the link connects to).

```
item := Item_At( tree );
Put( "Left child of " & item ) ;
item := Item_At( Left_Child( tree ) );
Put_Line( " is " & item ) ;
```

When the root of the tree is moved, any items above the new root that aren't referenced anymore are destroyed. To move around the tree without destroying nodes (which is typically what you want to do), create an "alias" to the root of the tree with Create prior to moving.

```
root := Create( tree ); -- create a reference to the root
Left_Child( tree );     -- safe: old root is not destroyed
```

Moving into an empty (null) position in the tree is allowed, but any attempt to look at the item there will raise an exception. The leaves and the parent of the root are empty.

The Trees.Binary package provides the following subprograms:

- **Clear** - Destroy the tree
- **Insert** - Insert an item at the tree's root
- **Append** - Add an item in the place of a particular item, moving the old item to a new position
- **Remove** - Remove an item from the tree
- **Share** - Create an alias to a subtree of the tree
- **Child/Left_Child/Right_Child** - Move to a child item
- **Parent** - Move towards the root
- **Set_Item** - Make an item the root of the tree
- **Has_Children** - True if the tree has any children items
- **Is_Null** - True if the tree has no items
- **Is_Shared** - True if any subtree has an alias to it
- **Is_Root** - True if the tree is at the root of tree
- **Item_At** - Return the item at the root of the tree

In addition, the tree may have an in_order, pre_order or post_order generic procedure. This procedure traverses the tree and executes processes each item. Pre_order processes an item before its children. Post_order processes an item after its children. In_order processes a node in the sort order of the tree--after all the left children but before all the right.

```
with BC.Containers.Trees.Binary.In_Order;
with BC.Containers.Trees.Binary.Pre_Order;
with BC.Containers.Trees.Binary.Post_Order;
with Global_Heap;

package shipment_binary is

   -- grandfather would be proud

   type aQuantity is ( Unknown, Basket_6Quart, Basket_11Quart, Bushel, Skid, Boxcar );
   type aFruit    is ( Apples, Grapes, Peaches, Cherries, Pears, Plums, Other );

   type aShipment is record
           number   : Positive;   -- number of containers
           quantity : aQuantity;  -- the containers
           contents : aFruit;     -- type of fruit
   end record;


   procedure visitShipment( s : aShipment; OK : out boolean );
   -- our tree traversal function

   package shipmentContainers is new BC.Containers( item=> aShipment );
   -- basic fruit container

   package shipmentTrees is new shipmentContainers.Trees;
   -- basic tree support

   package shipmentBinaryTrees is new shipmentTrees.Binary(
            Storage_Manager => Global_Heap.Pool,
            Storage => Global_Heap.Storage );
   -- our binary tree support
```

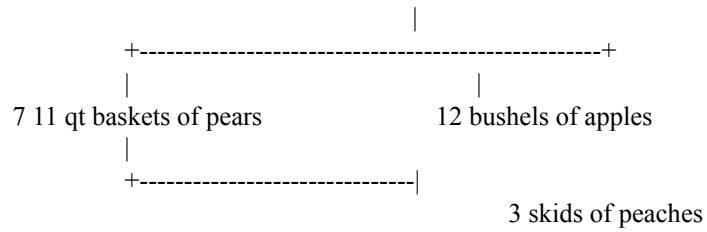
```
    procedure inOrdershipmentTraversal is new shipmentBinaryTrees.In_Order( visitShipment );
    -- an in-order traversal

    procedure preOrdershipmentTraversal is new shipmentBinaryTrees.Pre_Order( visitShipment );
    -- a pre-order traversal

    procedure postOrdershipmentTraversal is new shipmentBinaryTrees.Post_Order( visitShipment );
    -- a post-order traversal

end shipment_binary;
```

```
with ada.text_io;
use ada.text_io;

package body shipment_binary is

    procedure visitShipment( s : aShipment; OK : out boolean ) is
    -- our tree traversal function
    begin
        Put( "Shipment of" );
        Put( s.number'img );
        Put( " " );
        Put( s.quantity'img );
        Put( "(S) of " );
        Put_Line( s.contents'img );
        OK := true;
    end visitShipment;

end shipment_binary;
```

```
with ada.text_io, shipment_binary;
use ada.text_io, shipment_binary;

procedure bintree_demo is
    use shipmentBinaryTrees;
    root : Binary_Tree;
    t    : Binary_Tree;
    s    : aShipment;
    OK   : boolean;
begin
    Put_Line( "This is a demo of the Booch components: binary trees" );
    New_Line;

    -- this is the root item

    s.number := 5;
    s.quantity := basket_6quart;
    s.contents := cherries;
    Insert( t, s, Child => Left );
    -- child doesn't really matter because there's no prior item at the root

    root := Create( t ); -- remember where the root is

    -- add to left of root

    s.number := 7;
```

```
    s.quantity := basket_11quart;
    s.contents := pears;
    Append( t, s, Child => Left, After => Left );
    -- child doesn't really matter here

    -- add to right of root

    s.number := 12;
    s.quantity := bushel;
    s.contents := apples;
    Append( t, s, Child => Left, After => Right );
    -- child doesn't really matter here

    Left_Child( t );  -- move "t" down left branch

    s.number := 3;
    s.quantity := skid;
    s.contents := peaches;
    Append( t, s, Child => Left, After => Right );
    -- child doesn't really matter here

    Put_Line( "Our tree is: ");
    Put_Line( "          5 6 qt baskets of cherries" );
    Put_Line( "                          |" );
    Put_Line( "       +-----------------------------------------------------+" );
    Put_Line( "       |                                                     |" );
    Put_Line( "7 11 qt baskets of pears                        12 bushels of apples" );
    Put_Line( "       |" );
    Put_Line( "       +--------------------------------|" );
    Put_Line( "                               3 skids of peaches" );
    New_Line;

    Put_Line( "In-order traversal:" );
    inOrderShipmentTraversal( root, OK );
    if not OK then
        Put_Line( "The traversal was interrupted" );
    end if;
    New_Line;

    Put_Line( "Pre-order traversal:" );
    preOrderShipmentTraversal( root, OK );
    if not OK then
        Put_Line( "The traversal was interrupted" );
    end if;
    New_Line;

    Put_Line( "Post-order traversal:" );
    postOrderShipmentTraversal( root, OK );
    if not OK then
        Put_Line( "The traversal was interrupted" );
    end if;

end bintree_demo;
```

```
This is a demo of the Booch components: binary trees

Our tree is:
                    5 6 qt baskets of cherries
```



In-order traversal:
Shipment of 7 BASKET_11QUART(S) of PEARS
Shipment of 3 SKID(S) of PEACHES
Shipment of 5 BASKET_6QUART(S) of CHERRIES
Shipment of 12 BUSHEL(S) of APPLES

Pre-order traversal:
Shipment of 5 BASKET_6QUART(S) of CHERRIES
Shipment of 7 BASKET_11QUART(S) of PEARS
Shipment of 3 SKID(S) of PEACHES
Shipment of 12 BUSHEL(S) of APPLES

Post-order traversal:
Shipment of 3 SKID(S) of PEACHES
Shipment of 7 BASKET_11QUART(S) of PEARS
Shipment of 12 BUSHEL(S) of APPLES
Shipment of 5 BASKET_6QUART(S) of CHERRIES

Binary trees should not be Guarded.

18.1.14 AVL Trees

AVL trees are binary trees that are balanced. On every insert or delete, the tree is restructured to keep its symmetry. As a result, the trees must be sorted by the Booch component and the program using the AVL tree must provide a "<" function to sort the tree by.

The AVL package provides fewer subprograms than the binary tree package:

- **Clear** - destory the AVL tree
- **Insert** - add an item into the AVL tree
- **Delete** - remove an item form the AVL tree
- **Extent** - return the number of items in the AVL tree
- **Is_Null** - true if there are no items in the AVL tree
- **Is_Member** - true if an item is in the AVL tree
- **Visit** - traverse the tree in-order executing a "read only" procedure
- **Modify** - traverse the tree in-order executing a procedure that can alter the items.

There are no subprograms for walking the tree.

Here is a sample declaration:

```
with BC.Containers.Trees.AVL;
with Global_Heap;

package fruit_avl is

   -- more fun with fruit

   type aQuantity is ( Unknown, Basket_6Quart, Basket_11Quart, Bushel, Skid, Boxcar );
   type aFruit    is ( Apples, Grapes, Peaches, Cherries, Pears, Plums, Other );

   type aShipment is record
```

```
            number   : Positive;   -- number of containers
            quantity : aQuantity;  -- the containers
            contents : aFruit;     -- type of fruit
    end record;


    function sortCriteria( left, right : aShipment ) return boolean;
    -- for sorting the AVL tree

    package shipmentContainers is new BC.Containers( item=> aShipment );
    -- basic fruit container

    package shipmentTrees is new shipmentContainers.Trees;
    -- basic tree support

    package shipmentAVLTrees is new shipmentTrees.AVL(
                sortCriteria,
                Storage_Manager => Global_Heap.Pool,
                Storage => Global_Heap.Storage );
    -- our AVL tree support

end fruit_avl;
```

```
package body fruit_avl is

    function sortCriteria( left, right : aShipment ) return boolean is
    begin
        return left.number < right.number;
    end sortCriteria;

end fruit_avl;
```

AVL trees have slower inserts and deletes than binary trees but are faster than a normal binary tree for searching.

18.1.15 Multiway Trees

A multiway tree is a tree with any number of unsorted children (as opposed to a binary tree which always has no more than two chidren).

The subprograms are similar to a binary tree. The append procedures add child items to an item. A new function called "Arity" returns the number children an item has.

Multiway trees should not be Guarded.

18.1.16 Graphs

Essentially, graphs are a generalization of maps where any number of items can be related to each other (as opposed to only two).

A directed graph is a set of items (vertices) that are connected by relationships (edges or "arcs"). Like a single linked list, a program can only move forward along an arc.

Items can also be linked to themselves.

The graphs-directed package provides the following subprograms:

- **Create_Arc** - add a relationship between two items

- **Number_Of_Incoming_Arcs** - return the number of incoming arcs to an item
- **Number_Of_Outgoing_Arcs** - return the number of outgoing arcs to an item
- **Set_From_Vertex** - move an arch's source to a new item
- **Set_To_Vertex** - move an arc's destination to a new item
- **From_Vertex** - return the source item for an arc
- **To_Vertex** - return the destination item for an arc

There are four iterators: a graph iterator, and three iterators for visiting items (incoming, outgoing and both).

An undirected graph is a directed graph with pointers to both the previous and next item along an arc. Like a double linked list, a program can move forwards or backwards along an arc.

The graphs-undirected package provides the following subprograms:

- **Create_Arc** - add a relationship between two items
- **Arity** - return the number of arcs. Self-arcs are counted only once
- **Set_First_Vertex** - move an arch's first item to a new item
- **Set_Second_Vertex** - move an arc's second item to a new item
- **First_Vertex** - return the first item for an arc
- **Second_Vertex** - return the second item for an arc

There are two iterators: a graph iterator and an item iterator.

Graphs should not be Guarded.

18.1.17 Smart Pointers

Smart pointers are an access type that counts the number of references to the item being pointed to. Your program allocates the item. The item is deallocated when no more pointers point to it. Smart pointers are a simplified form of garbage collection.

The smart package provides the following subprograms:

- **Create** - create a new smart pointer from an access variable
- **Value** - return the item pointed to by the smart pointer

```
with BC.smart;

package depts is

   type departments is ( accounting, information_technology, shipping, human_resources );
   type deptAccess  is access all departments;
   package deptPtrs is new BC.smart( departments, deptAccess );

end depts;
```

```
with ada.text_io, depts;
use ada.text_io, depts;

procedure sp_demo is
   accountingPtr  : deptPtrs.Pointer;
   accounting2Ptr : deptPtrs.Pointer;
   department     : deptAccess;
begin
   Put_Line( "This is a demo of the Booch components: smart pointers" );
   New_Line;
```

```
    department := new departments'( accounting );

    Put_Line( "Assigning dynamically allocate value to a smart pointer" );
    accountingPtr := deptPtrs.Create( department );
    Put_Line( "The accounting pointer points at " & deptPtrs.Value( accountingPtr ).all'img );
    New_Line;

    Put_Line( "Assigning a smart pointer to a smart pointer" );
    accounting2Ptr := accountingPtr;
    Put_Line( "The accounting pointer 2 points at " & deptPtrs.Value( accounting2Ptr ).all'img );
    New_Line;

    Put_Line( "The memory is released when the program ends or no more pointers" );
    Put_Line( "access the memory." );
end sp_demo;
```

```
This is a demo of the Booch components: smart pointers

Assigning dynamically allocate value to a smart pointer
The accounting pointer points at ACCOUNTING

Assigning a smart pointer to a smart pointer
The accounting pointer 2 points at ACCOUNTING

The memory is released when the program ends or no more pointers
access the memory.
```

18.1.18 Booch Multithreading

Booch components can be guarded (manually "locking" the structure for exclusive access) or synchronized (implicit blocking) for multithreading purposes.

Guarding is implemented by creating extending a container type to a Guarded_Container using the GC.Containers.Guarded package. Guarded containers contain two new subprograms, "Seize" and "Release", to lock and unlock a container. (This is implemented using a semaphore.) Any Booch data structure can be made guarded using guarded containers, but in some cases guarding will not work as expected and should not be used (for example, with lists).

The basic semaphore locks individual objects (although it many not work as expected on certain structures such as lists, according to AdaPower.Net). The basic semaphore can be extended and customized by a programmer.

Rewriting the Bags example with guards:

```
with BC.Containers.Bags.Unbounded;
with BC.Containers.Guarded;
with BC.Support.Synchronization;
with Global_Heap;

package guarded_customers is

    type aCustomerID is new integer range 1_000..9_999;

    function IDHash( id : aCustomerID ) return Positive;
    -- our hash function

    package customerContainers is new BC.Containers (Item => aCustomerID);
    -- this is the basic container
```

```
   package customerBags is new customerContainers.Bags;
   -- create a new bag support for our using container type

   package customerBag is new customerBags.Unbounded(
                  Hash => IDHash,
                  Buckets => 99,
                  Storage_Manager => Global_Heap.Pool,
                  Storage => Global_Heap.Storage);
   -- create an unbounded bag holding customer numbers

   package customerGuardedBag is new customerContainers.Guarded (
        Base_Container => customerBag.Unbounded_Bag,
        Semaphore => BC.Support.Synchronization.Semaphore );
   -- create a new controlled tagged record container for customers

end guarded_customers;
```

A new guarded bag can now be declared:

```
   customers : customerGuardedBag.Guarded_Container;
```

and the bag can be locked using

```
   customerGuardedBag.Seize( customers );
```

Synchronized access by threads is implemented in special versions of the data structure packages (for example, maps.synchronized). With synchronized packages, the implementation details are hidden from the user.

18.2 XMLAda - Unicode, XML, SAX and DOM

Ada Core Technologies provides a set of XML and Unicode packages called **XMLAda**. It is open source and can be downloaded from the ACT Europe's web site at https://libre2.adacore.com/xmlada/. Limited documentation is available online at this web site as well.

Unicode is bundled together with XML because XML uses Unicode characters. Unicode characters are not the same as Ada's standard 16-bit wide character type. Wide characters are based on a different standard.

This is an overview of using these packages, including short examples to see how they are used. It does not discuss the details of Unicode or XML.

When you configure XMLAda, you're given a choice of where to install the library (/usr/local might be a good choice). To compile, bind and link your programs, you'll need to include the locations of the library and Ada package spec files. For example, for /usr/local you will need to use these options:

1. gnatmake -c -I. -I/usr/local/include/xmlada/ -L/usr/local/lib/ my_program
2. gnatbind -aO./ -I. -I/usr/local/include/xmlada/ -I- -x my_program.ali
3. gnatlink my_program.ali -L/usr/local/lib/ -lxmlada_unicode -lxmlada_input_sources -lxmlada_sax -lxmlada_dom

If you are not using a particular subsystem, you can omit the -l library option. However, DOM is built using SAX--if you use the DOM packages, you'll need to include the SAX library.

18.2.1 Unicode Basics

Unicode comes in three versions: fixed length 32-bit UTF-32, variable length 16-bit UTF-16, and variable length UTF-8. All three versions can represent the same characters, from ASCII to ancient languages. **UTF-32** is the basic Unicode type and each character is always 32 bits. **UTF-16** is a compressed version of UTF-32: each character is 16-bits but some character as represented by two 16-bit characters in a row. **UTF-8** has even more compression: the first 128 characters are equivalent to ASCII (or Latin-1), the standard Ada character sets, but the upper 128 codes can be used for characters using up to 6 consecutive bytes for one character. The Ada packages support all three types of Unicode.

An example of UTF-8 in many languages is available on the UTF-8 Sampler Page. This web page is composed of UTF-8 characters and uses "<META http-equiv="Content-Type" content="text/html; charset=utf-8"> in the HTML header. The xterm program is also supposed to support UTF-8 characters (for example, if you dump a UTF-8 document to standard output in an xterm window).

Do you need to know the character codes? They are available in sets of PDF charts at unicode.org.

The Unicode packages sometimes work with string types, sometimes with numeric types. A basic UTF-32 character (type unicode_char defined inunicode.ads) and is a 32-bit number.

```
type Unicode_Char is mod 2**32;
```

Basic UTF-32 functions such as is_letter, is_digit or to_unicode (ASCII/Latin-1 to UTF-32 character) are found in unicode.ads.

18.2.2 UTF and ASCII Characters

The first 127 characters in Unicode are identical to the 127 character ASCII set. If you are strictly working with these characters, it's easy to convert them to Unicode characters.

```
with unicode;
use unicode;
procedure ASCII_to_Unicode is
   utf_32  : unicode_char;   -- UTF-32 character
   ascii_ch : character;      -- Latin-1 (or ASCII) character
begin
   utf_32 := character'pos( 'A' );
   ascii_ch := character'val( utf_32 );
end ASCII_to_Unicode;
```

Characters above ASCII 127 will result in garbage or a run-time CONSTRAINT_ERROR exception.

The Unicode packages do not define a UTF-8 character type but you can create your own type.

```
procedure ASCII_to_UTF_8 is
   subtype unicode_utf_8_char is unicode_char range 0..255;
   utf_8 : unicode_utf_8_char;
   ascii_ch : character;
begin
   utf_8 := character'pos( 'A' );
   ascii_ch := character'val( utf_8 );
end ASCII_to_UTF_8;
```

Memory can be saved by limiting the UTF-8 character to 7 bits.

```
type unicode_utf_8_char is new unicode_char range 0..127;
for unicode_utf_8_char'size use 7;
```

18.2.3 Unicode and Latin-1 Conversions

Latin-1 (the extended ASCII standard used by Ada) and the first 8 bits of UTF-32 are not identical. To convert between them, you'll need to use a Unicode conversion function. The Unicode packages contain many conversion functions under the unicode.ccs package hierarchy. In particular,unicode.css.iso_8859_1.ads performs Latin-1 conversions.

The following short program demonstrates how to convert between UTF-32 strings and Ada unbounded strings (Latin-1 characters). (The next section will show how to convert any string to any character set.)

```
with ada.text_io,
        ada.strings.unbounded,
        unicode.ccs.iso_8859_1;
use ada.text_io,
        ada.strings.unbounded,
        unicode,                -- basic Unicode
        unicode.ccs,             -- Unicode conversions
        unicode.ccs.iso_8859_1;    -- Latin-1 conversions

procedure unitest is

    type a_unicode_string is array(1..80) of unicode_char;

    function to_unicode_string( msg : unbounded_string ) return a_unicode_string is
       -- to 32 byte unicode characters (UTF-32)
       result : a_unicode_string := (others => 0);
    begin
       for c in 1..length( msg ) loop
             result(c) := to_unicode( element( msg, c ) );
       end loop;
       return result;
    end to_unicode_string;

    function to_unbounded_string( msg : a_unicode_string ) return unbounded_string is
       -- to Latin-1
       result : unbounded_string;
       c : natural;
    begin
       c := 1;
       while c <= a_unicode_string'last and then msg( c ) /= 0  loop
             result := result & character'val( to_iso_8859_1( msg( c ) ) );
             c := c + 1;
       end loop;
       return result;
    end to_unbounded_string;

    msg : unbounded_string := to_unbounded_string( "this is a test" );
    umsg : a_unicode_string;
    msg2 : unbounded_string;

begin

    put_line( "Original: " & to_string( msg ) );
    umsg := to_unicode_string( msg );
    msg2 := to_unbounded_string( umsg );
    put_line( "After conversion: " & to_string( msg2 ) );

end unitest;
```

Using characters above Latin-1 255 will result in garbage or a run-time CONSTRAINT_ERROR exception.

18.2.4 Ada, UTF-8 and UTF-16 Strings

UTF-32 characters are the most flexible form of Unicode but they use a large amount of space. UTF-8 uses less space than UTF-32 and is compatible with ASCII but the characters can require up to 6 consecutive bytes. Unfortunately, although the Unicode packages support UTF-8, it is not easy to use. It is not supported directly and all conversions must be done through UTF-32. This creates some confusing terminology as there are many "to_utf32" and "from_utf32" functions that work on different kinds of 32-bit characters (not necessarily "UTF" characters at all!).

Converting an Ada string into UTF-8 format is a 3 step process:

1. Convert the Ada string from 8-bit to 32-bit characters (using the unicode.ces.basic_8bit) package.
2. Convert the 32-bit characters to UTF-32 32-bit characters (using the unicode.ces.utf32 package and the unicode.css.iso_8859_1 package).
3. Convert the UTF-32 string into a UTF-8 string (using the unicode.ces.utf8) package.

The unicode.ces packages include string types that work with the functions. String types with the name "UTF32" have 32-bit characters. "UTF8" strings have 8-bit characters. "LE" strings are "little-endian". These strings are all renamings of a standard Ada string and are provided to make programs easier to read. That is, a unicode_char is a number but a utf32_le_string is a string type, not an array of numbers as one might expect. Another effect is that you cannot usethe unicode.ces packages and refer to the functions: they must be named in full so the compiler will know what functions you are referring to.

Putting it together, here is the method to convert an Ada string to a UTF-8 string:

```
s : string := "This is a test";
--  8-bit Latin-1 string (normal Ada string)

s_32 : utf32_le_string := unicode.ces.basic_8bit.to_utf32( s );
-- 32-bit Latin-1 string (normal Ada string with 32-bit characters)

u_32 : utf32_le_string := unicode.ces.utf32.to_unicode_le( s_32,
            cs => unicode.ccs.iso_8859_1.iso_8859_1_character_set );
-- UTF-32 string (convert Latin-1 to Unicode characters)

u_8  : utf8_string := unicode.ces.utf8.from_utf32( u_32 );
-- change UTF-32 to UTF-8
```

The UTF-8 string may have more bytes than the original string. Use the unicode.ces.utf8.length function (instead of the 'length attribute) to determine the number of characters in the string.

To convert from UTF-8 to Ada strings:

1. Convert the UTF-8 string into a UTF-32 string (using the unicode.ces.utf8) package.
2. Convert the 32-bit characters to Latin-1 32-bit characters (using the unicode.ces.utf32 package and the unicode.css.iso_8859_1 package).
3. Convert the 32-bit Ada string to 8-bit characters (using the unicode.ces.basic_8bit) package.

Here is a complete example that translates a string to UTF-8 and back to an Ada string:

```
with ada.text_io,
        unicode.ces.utf8,
        unicode.ces.utf32,
        unicode.ces.basic_8bit,
```

```
        unicode.ccs.iso_8859_1;
use  ada.text_io,
        unicode,
        unicode.ccs,
        unicode.ces,
        unicode.ces.utf8,
        unicode.ces.utf32;

procedure there_and_back_again is

    s    : string := "This is a test";
    --  8-bit Latin-1 string (normal Ada string)

    s_32 : utf32_le_string := unicode.ces.basic_8bit.to_utf32( s );
    -- 32-bit Latin-1 character (normal Ada string with 32-bit characters)

    u_32 : utf32_le_string := unicode.ces.utf32.to_unicode_le( s_32,
                cs => unicode.ccs.iso_8859_1.iso_8859_1_character_set );
    -- UTF-32 string (convert Latin-1 to Unicode characters)

    u_8  : utf8_string := unicode.ces.utf8.from_utf32( u_32 );
    -- change UTF-32 to UTF-8

    undo_u_32 : utf32_le_string := unicode.ces.utf8.to_utf32( u_8 );
    -- change UTF-8 to UTF-32

    undo_s_32 : utf32_le_string := unicode.ces.utf32.to_unicode_le( undo_u_32,
                cs => unicode.ccs.iso_8859_1.iso_8859_1_character_set );
    -- change UTF-32 to Latin-1 with 32-bit characters
    -- KB: is this right?  Would make more sense as to_cs but that function
    -- throws an exception...

    undo_s : string := unicode.ces.basic_8bit.from_utf32( undo_s_32 );
    -- back to original string "This is a test"

begin
    put_line( "Original string = '" & s & "'" );
    put_line( "       To UTF-8 = '" & u_8 & "'" );
    put_line( "     Back again = '" & undo_s & "'" );
    if s = undo_s then
        put_line( "Translation successful" );
    end if;
end there_and_back_again;
```

The results:

```
Original string = 'This is a test'
             To UTF-8 = 'This is a test'
          Back again = 'This is a test'
Translation successful
```

Using the same process, you can use UTF-16 by using the appropriate packages.

KB: Do the strings need to be free'd?

18.2.5 Inputting XML

To provide a standard way of reading XML information, XMLAda includes a set of input packages. This packages are designed to read information from a file or string. You can also create custom input functions to read XML information from other sources.

The Input packages are designed to look like standard Ada file I/O operations. To use an XML file, use the Input_Sources.File package. You open the file, get a Unicode character, check for the end of file and close the file. The file could be a string or some other form of input. The Set_Encoding procedure informs the Input packages about the type of characters (Latin_1, UTF-8, etc.), although the encoding of a file can usually be determined automatically.

```
data : Input_Sources.File.File_Input;
...
Input_Sources.File.Open( "data.xml", data );
-- Parse the information
Input_Sources.File.Close( data );
```

If you are using SAX or DOM, you normally only need to open or close the XML source. The parsers will handle reading the XML data.

This program will dump the contents of an XML file called "data.xml" to standard output:

```
with ada.text_io,
        unicode,
        input_sources.file;
use  ada.text_io,
        unicode,
        input_sources.file;

procedure xml_dump is
    data : File_Input;
    uc   : unicode_char;
begin
    Open( "data.xml", data );
    while not Input_Sources.File.Eof( data ) loop
        Next_Char( data, uc );
        if uc >= 32 and uc < 127 then   -- Printable ASCII?
            put( character'val( uc ) );
        elsif uc = 10 then              -- Linefeed
            new_line;
        else
            put( "[#" & uc'img & ']' ); -- other character, print numeric code
        end if;
    end loop;
    new_line;
    Close( data );
end xml_dump;
```

Strings are more difficult. Although there is a Input_Sources.Strings package, there is no predefined handling for Ada or Unicode strings. You have to write a set of callbacks to read the string yourself.

```
with ada.text_io,
        unicode.ces,
        unicode.ccs.iso_8859_1,
        input_sources.strings;
use ada.text_io,
        unicode,
        unicode.ces,
        input_sources.strings;
```

```
procedure xml_dump_string is

   data : String_Input;
   uc   : unicode_char;

   s    : aliased byte_sequence := "<thing>This is a test</thing>";
   -- s is the source string

   -- These callbacks are described in unicode.ces.  For Latin_1, they need
   -- to work with one byte characters only.

   procedure s_read( Str   : Byte_Sequence; Index : in out Positive; Char  : out Unicode_Char) is
   begin
      char := character'pos( str( index ) );
      index := index + 1;
   end s_read;

   function s_width( char : Unicode_Char ) return natural is
   begin
      return 1;
   end s_width;

   procedure s_encode( Char   : Unicode_Char; Output : in out Byte_Sequence; Index  : in out Natural) is
   begin
      output( index+1 ) := character'val( char );
   end s_encode;

   function s_length( str : byte_sequence ) return natural is
   begin
     return str'length;
   end s_length;

   scheme : encoding_scheme := ( s_read'unrestricted_access, s_width'unrestricted_access,
s_encode'unrestricted_access, s_length'unrestricted_access);
   -- a collection of callbacks to read string s
   -- I should not use unrestricted_access but don't want to fool around with
   -- Ada's pointer scoping restrictions for this short example...
begin

   Open( s'unrestricted_access, scheme, data );
   while not Eof( data ) loop
      Next_Char( data, uc );
      if uc >= 32 and uc < 127 then   -- Printable ASCII?
         put( character'val( uc ) );
      elsif uc = 10 then              -- Linefeed
         new_line;
      else
         put( "[#" & uc'img & ']' );
      end if;
   end loop;
   new_line;
   Close( data );

end xml_dump_string;
```

The results:

<thing>This is a test</thing>

18.2.6 Parsing XML Using SAX (Simple API for XML)

Interpreting XML is difficult because it has a recursive structure: using nested tags allows great flexibility but it means that any individual tag only has meaning when considered in the context of the surrounding tags. Or to put it another way, you can't grep XML.

XMLAda provides two methods for interpreting (or parsing) an XML document. The first is SAX (Simple API for XML) and is particularily useful for short documents or documents without a lot of nested context. SAX is sometimes referred to as as callback or push-based parsing. SAX reads through an XML document and calls procedures you define to handle the different parts of the document SAX encounters. Because SAX is a standard, using SAX in Ada is very similar to using SAX in other computer languages.

Create a set of handlers for different XML components (elements (or tags), attributes, free text, etc.). These handlers should be put in a tagged record extended from sax.readers.reader. With objects, you can create a hierarchy of readers for different kinds of XML files.

Since SAX uses callbacks, it doesn't have to load the entire XML document at one time. XML documents with simple structures can be quickly analysed.

The simplest SAX program reads through an XML file and does nothing by default:

```
with ada.text_io,
        unicode.ces,
        input_sources.file,
        sax.readers;
use ada.text_io,
        unicode,
        input_sources.file,
        sax.readers;

procedure sax_nothing is
    data : File_Input;
    r    : sax.readers.reader;
begin
    Open( "data.xml", data );
    Parse( r, data );
    Close( data );
end sax_nothing;
```

To do something useful, you have to extend the class and create new handler functions. Create a new tagged record based on sax.readers.reader and replace that default content handlers for the content you want to examine. The most common ones are:

- **Start_Document**: called before parsing the document
- **End_Document**: called after the document is finished parsing
- **Start_Element**: called when SAX reads an XML open tag
- **End_Element**: called for a XML closing tag
- **Characters**: called for the free characters between the tags, not including ignorable whitespace (another callback). The SAX standard does not specify whether or not the handler receives all or part of a set of free characters--XMLAda calls Characters with a complete set of free characters.

The parameters and their names are based on the SAX standard. The most common parameters are:

- **Local_Name**: the name of the item minus an namespace
- **Qname**: the name as seen in the document
- **Attributes**: the attributes attached to a tag (as a tagged record)

There are many possible handlers. Check sax.readers.ads for a complete list.

If the XML file is incorrect, an XML_FATAL_ERROR exception will be raised.

18.2.7 SAX Example: RSS Headlines

RSS is an XML standard for reporting news articles. An RSS file contains news articles look like this:

```
<item>
    <title>The Headline</title>
    <link>Link to the article</title>
    ...
</item>
```

Suppose you want to write a program to display news headlines. To do this, you'd need to capture the free text (the Characters callback) and display the text when the </title> tag is reached (the End_Element callback). But you only want to display the <title> tags inside of <item> tags because there are other titles (such as the title of the web site where the items came from).

Create a new package called my_readers to contain our RSS tagged record called my_rss_reader. The record will have handlers for starting and ending tags as well as character content. The titles are displayed with the dump procedure. This procedure assumes that the XML file is ASCII or Latin-1, but the procedure can be rewritten to handle Unicode as discussed above.

```
-- my_readers.ads
with Sax.Exceptions;
with Sax.Locators;
with Sax.Readers;
with Sax.Attributes;
with Sax.Models;
with Unicode.CES;

with ada.strings.unbounded;
use  ada.strings.unbounded;

package my_readers is
   type my_rss_reader is new sax.readers.reader with private;

     procedure Start_Element
        (Handler      : in out my_rss_reader;
         Namespace_URI : Unicode.CES.Byte_Sequence := "";
         Local_Name    : Unicode.CES.Byte_Sequence := "";
         Qname         : Unicode.CES.Byte_Sequence := "";
         Atts          : Sax.Attributes.Attributes'Class);
     -- The start of a tag.  e.g. <title>

     procedure End_Element
        (Handler : in out my_rss_reader;
         Namespace_URI : Unicode.CES.Byte_Sequence := "";
         Local_Name    : Unicode.CES.Byte_Sequence := "";
         Qname         : Unicode.CES.Byte_Sequence := "");
     -- The end of a tag.  e.g. </title>

     procedure Characters
        (Handler : in out my_rss_reader; Ch : Unicode.CES.Byte_Sequence);
     -- The free text contained between tags.  Ch is a string not a single character.

     -- The rest we don't need
```

```
private

   type my_rss_reader is new sax.readers.reader with record
           title   : unbounded_string;    -- last free text
           in_item : boolean := false;    -- true if in an <item>
   end record;

end my_readers;
```

```
-- my_readers.adb
with ada.text_io;
use  ada.text_io;

package body my_readers is

     procedure dump( ch : unicode.ces.byte_sequence ) is
        -- Dump some ASCII-compatible characters.  This assumes ASCII or
        -- Latin-1 characters in the XML file.
        c : character;
        ch_code : natural;
     begin
        for i in 1..ch'length loop
            c := ch(i);
            ch_code := character'pos( c );
            if ch_code >= 32 and ch_code < 127 then       -- Printable ASCII?
                put( c );
            elsif ch_code = 10 then                       -- Linefeed
                new_line;
            else                                  -- Others
                put( "[#" & ch_code'img & ']' );
            end if;
        end loop;
   end dump;

   procedure Start_Element
      (Handler       : in out my_rss_reader;
       Namespace_URI : Unicode.CES.Byte_Sequence := "";
       Local_Name    : Unicode.CES.Byte_Sequence := "";
       Qname         : Unicode.CES.Byte_Sequence := "";
       Atts          : Sax.Attributes.Attributes'Class) is
   begin
        if local_name = "item" then                   -- <item>
            Handler.in_item := true;                  -- starting an item
        end if;
   end Start_Element;

   procedure End_Element
      (Handler : in out my_rss_reader;
       Namespace_URI : Unicode.CES.Byte_Sequence := "";
       Local_Name    : Unicode.CES.Byte_Sequence := "";
       Qname         : Unicode.CES.Byte_Sequence := "") is
   begin
        if local_name = "title" then                  -- </title>
            if Handler.in_item then                   -- item title?
                dump( to_string( handler.title ) );       -- show it
                new_line;
            end if;
```

```
        elsif local_name = "item" then                  -- </item>
                Handler.in_item := false;                   -- leaving an item
        end if;
    end End_Element;

    procedure Characters
       (Handler : in out my_rss_reader; Ch : Unicode.CES.Byte_Sequence) is
    begin
       handler.title := to_unbounded_string( ch );
    end Characters;

end my_readers;
```

```
-- rss.adb
with ada.text_io,
         unicode.ces,
         input_sources.file,
         my_readers;
use  ada.text_io,
         unicode,
         input_sources.file,
         my_readers;

procedure rss is
      data : File_Input;
      r    : my_rss_reader;
begin
      Open( "coder.rss", data );
      Parse( r, data );
      Close( data );
end rss;
```

The results for my coder.rss file for my Lone Coder blog is as follow:

```
Lone Coder: Google: Lawful Good or Chaotic Neutral?
Lone Coder: The Tyrrany of the Label
Lone Coder: Losing Control of your Linux Startup
Lone Coder: OpenSuSE 10: Developer Dream or Crippleware?
Lone Coder: In Mourning of Statftime
Lone Coder: The Need for Speed: Speedbumps on the Web
```

18.2.8 SAX Attribute Handling

An attribute is additional information added to an XML tag. For example, in HTML with <p align="right">, align="right" is an attribute. align is the name of the attribute. right is the value of the attribute.

The Start_Element handler has an Atts tagged record that describes any attributes included with the tag. The subprograms available for attributes are described in sax-attributes.ads. The most commonly used subprograms are:

- **Get_Length** - return the number of attributes
- **Get_Local_Name** - return the name of an attribute
- **Get_Value** - return the value of an attribute

Attributes are numbered from zero. To display the attributes on an XML tag, use the following in your Start_Element handler:

```
use SAX.Attributes;
...
for i in 0..get_length( Atts )-1 loop
        put_line( "Name = " & get_local_name( Atts, i ) );
        put_line( "Value = " & get_value( Atts, i ) );
end loop;
```

18.2.9 Parsing XML Using DOM (Document Object Model)

SAX calls your callbacks as it parsers an XML file. DOM, Document Object Model, works by loading the entire XML file and building a tree based on the tags. Using the DOM packages, you can "walk" the tree, node by node. This uses more memory but doesn't constrain how you examine the document the way SAX does (top to bottom). DOM, like SAX, is a standard and using DOM in Ada is similar to using DOM in other languages.

The default DOM reader, Tree_Reader, is used to create the document tree. Unlike SAX, the default reader will provide most of the functionality you need and you won't need to extend it.

```
with input_sources.file,
          dom.readers,
          dom.core;
use  dom.readers,
          dom.core,
          input_sources.file;

procedure dom_nothing is
      data : File_Input;
      r    : Tree_Reader;
      d    : Document;
begin
      Open( "data.xml", data );
      d := get_tree( r );
      Parse( r, data );
      free( r );
      Close( data );
end dom_nothing;
```

A longer example: display the first XML tag with attributes.

```
with input_sources.file,
          dom.readers,
          dom.core.documents,
          dom.core.nodes;
use dom.readers,
          dom.core,
          dom.core.documents,
          dom.core.nodes,
          input_sources.file;

with ada.text_io;
use  ada.text_io;

procedure dom_first_node is

      procedure dump( n : node ) is
      begin
         if n /= null then
               case n.node_type is
               when element_node =>                    -- An XML tag
```

```
                    put( "<" );
                    put( node_name( n ) );                  -- show name
                    declare                                 -- attributes
                       AS : Named_Node_Map := Attributes (N);
                    begin
                       for I in 0..Length( AS )-1 loop      -- numbered from zero
                          put( " " );
                          put( node_name( Item(AS, I) ) );
                          put( "=" );
                          put( node_value( Item(AS, I) ) );
                       end loop;
                    end;
                    put_line( ">" );
              when attribute_node =>                        -- An attribute
                    put_line( node_name( n ) & "=" & node_value( n ) );
              when others =>                                -- Other
                    put_line( "Unknown tree node type" );
              end case;
           end if;
        end dump;

        data : File_Input;
        r    : Tree_Reader;
        d    : Document;
        document_root : element;

begin
        Open( "data.xml", data );
        Parse( r, data );
        d := get_tree( r );
        document_root := get_element( d );
        dump( document_root );
        free( r );
        Close( data );
end dom_first_node;
```

For my coder.rss file, this program displays:

```
<rss version=2.0>
```

The dom-core-nodes.ads package includes navigation subprograms to move around the document and examine nodes (XML items). first_child returns the first child nested in an XML element. next_sibling moves to the next adjacent item. For a complete example, David Botton has an example program that walks an XML document using DOM at his AdaPower web site.

18.3 General Purpose Libraries

18.3.1 AdaCL

AdaCL is a library for writing small day-to-day programs normally handled by scripting languages. It includes CGI, garbage collection and improved string functions. The home page for the project is AdaCL.

An alternative to my BUSH project

.

18.3.2 SAL (Stephe's Ada Library)

SAL is another "standard template library"-type project providing general data structures, vectors and math functions with a heavy emphasis on generics. The library is located at http://www.toadmail.com/~ada_wizard/ada/sal.html.

Summary of packages:

SAL.Aux.Definite_Private_Items;
SAL.Aux.Definite_Private_Items;
SAL.Aux.Enum_Iterators;
SAL.Aux.Indefinite_Limited_Items;
SAL.Aux.Indefinite_Private_Items.Comparable;
SAL.Aux.Indefinite_Private_Items;
SAL.Aux.Sort_Indefinite_Items_Definite_Keys;
SAL.Config_Files.Boolean;
SAL.Config_Files.Duration;
SAL.Config_Files.Integer;
SAL.Config_Files.Time;
SAL.Config_Files;
SAL.Endianness;
SAL.Gen.Alg.Count;
SAL.Gen.Alg.Find_Binary;
SAL.Gen.Alg.Find_Linear;
SAL.Gen.Alg.Find_Linear.Sorted;
SAL.Gen.Alg.Process_All_Constant;
SAL.Gen.Alg;
SAL.Gen.Gray_Code;
SAL.Gen.Lists.Double.Iterators;
SAL.Gen.Lists.Double;
SAL.Gen.Lists;
SAL.Gen.Stacks.Bounded_Limited;
SAL.Gen.Stacks.Bounded_Nonlimited;
SAL.Gen.Stacks;
SAL.Gen.Word_Order_Convert.Scalar_32;
SAL.Gen.Word_Order_Convert.Scalar_64;
SAL.Gen.Word_Order_Convert;
SAL.Gen_Array_Image;
SAL.Gen_Array_Text_IO;
SAL.Gen_FIFO;
SAL.Gen_Math.Gen_Den_Hart;
SAL.Gen_Math.Gen_DOF_2;
SAL.Gen_Math.Gen_DOF_3.Gen_Image;
SAL.Gen_Math.Gen_DOF_3;
SAL.Gen_Math.Gen_DOF_6.Gen_Image;
SAL.Gen_Math.Gen_DOF_6.Gen_Integrator_Utils;
SAL.Gen_Math.Gen_DOF_6;
SAL.Gen_Math.Gen_Inverse_Array;
SAL.Gen_Math.Gen_Manipulator;
SAL.Gen_Math.Gen_Runge_Kutta_4th;
SAL.Gen_Math.Gen_Scalar;
SAL.Gen_Math.Gen_Square_Array.Gen_Inverse;
SAL.Gen_Math.Gen_Square_Array;
SAL.Gen_Math.Gen_Vector.Gen_Image;
SAL.Gen_Math.Gen_Vector;
SAL.Generic_Binary_Image;
SAL.Generic_Decimal_Image;
SAL.Generic_Float_Image;
SAL.Generic_Hex_Image;
SAL.Math_Double.Den_Hart;
SAL.Math_Double.DOF_2;
SAL.Math_Double.DOF_3.Cacv_Inverse;
SAL.Math_Double.DOF_3.Image;

SAL.Math_Double.DOF_3;
SAL.Math_Double.DOF_6.Image;
SAL.Math_Double.DOF_6.DC_Array_DCV_Inverse;
SAL.Math_Double.DOF_6.Integrator_Utils;
SAL.Math_Double.DOF_6;
SAL.Math_Double.Elementary;
SAL.Math_Double.Scalar;
SAL.Math_Double.Text_IO;
SAL.Math_Float.Den_Hart;
SAL.Math_Float.DOF_2;
SAL.Math_Float.DOF_3.Cacv_Inverse;
SAL.Math_Float.DOF_3.Image;
SAL.Math_Float.DOF_3;
SAL.Math_Float.DOF_6.DC_Array_DCV_Inverse;
SAL.Math_Float.DOF_6.Integrator_Utils;
SAL.Math_Float.DOF_6;
SAL.Math_Float.Elementary;
SAL.Math_Float.Polynomials;
SAL.Math_Float.Scalar;
SAL.Math_Float.Text_IO;
SAL.Math_Float_Kraft_HC_Nominal;
SAL.Math_Float_Manipulator_6;
SAL.Math_Float_Manipulator_7;
SAL.Math_Float_RRC_K1607_Nominal;
SAL.Memory_Streams.Address;
SAL.Memory_Streams.Bounded;
SAL.Memory_Streams;
SAL.Poly.Alg.Count;
SAL.Poly.Alg.Find_Linear;
SAL.Poly.Alg.Process_All_Constant;
SAL.Poly.Binary_Trees.Sorted.Iterators;
SAL.Poly.Binary_Trees.Sorted;
SAL.Poly.Binary_Trees;
SAL.Poly.Function_Tables.Monotonic.First_Order;
SAL.Poly.Function_Tables.Monotonic;
SAL.Poly.Function_Tables;
SAL.Poly.Lists.Double;
SAL.Poly.Lists.Single.Iterators;
SAL.Poly.Lists.Single;
SAL.Poly.Lists;
SAL.Poly.Stacks.Unbounded_Array;
SAL.Poly.Stacks;
SAL.Poly.Unbounded_Arrays;
SAL.Poly;
SAL.Simple.Function_Tables.Monotonic.First_Order;
SAL.Simple.Function_Tables.Monotonic;
SAL.Simple.Function_Tables;
SAL.Simple.Searches.Binary;
SAL.Simple.Searches;
SAL.Simple;
SAL.Time_Conversions;

18.3.2 Jeffery Carter's PragmARCs

The PragmAda Reusable Components (PragmARCs) are another data structure library. Support contracts are available for commercial projects. The library is located at http://pragmada.home.mchsi.com/pragmarc.htm.

Summary of packages:

PragmARC.Ansi_Tty_Control
PragmARC.Assertion_Handler

PragmARC.Assignment
PragmARC.Bag_Unbounded
PragmARC.Bag_Unbounded_Unprotected
PragmARC.Binary_Searcher
PragmARC.Binary_Semaphore_Handler
PragmARC.Character_Regular_Expression_Matcher
PragmARC.Complex
PragmARC.Date_Handler
PragmARC.Deck_Handler
PragmARC.Forwarder
PragmARC.Genetic_Algorithm
PragmARC.Get_Line
PragmARC.Hash_Fast_Variable_Length
PragmARC.Images
PragmARC.Images.Image
PragmARC.Least_Squares_Fitting
PragmARC.Linear_Equation_Solver
PragmARC.List_Bounded
PragmARC.List_Bounded_Unprotected
PragmARC.List_Unbounded
PragmARC.List_Unbounded_Unprotected
PragmARC.Math
PragmARC.Math.Integer_Functions
PragmARC.Math.Functions
PragmARC.Matrix_Math
PragmARC.Menu_Handler
PragmARC.Min_Max
PragmARC.Mixed_Case
PragmARC.Monitor_Handler
PragmARC.Postfix_Calculator
PragmARC.Protected_Option
PragmARC.Queue_Bounded
PragmARC.Queue_Bounded_Blocking
PragmARC.Queue_Bounded_Unprotected
PragmARC.Queue_Unbounded
PragmARC.Queue_Unbounded_Blocking
PragmARC.Queue_Unbounded_Unprotected
PragmARC.Quick_Searcher
PragmARC.Reflection
PragmARC.Regular_Expression_Matcher
PragmARC.REM_NN_Wrapper
PragmARC.Safe_Pointers
PragmARC.Safe_Suspension_Objects
PragmARC.Safe_Semaphore_Handler
PragmARC.Set_Discrete
PragmARC.Skip_List_Unbounded
PragmARC.Skip_List_Unbounded.Put
PragmARC.Sort_Heap
PragmARC.Sort_Insertion
PragmARC.Sort_Quick_In_Place
PragmARC.Sort_Radix
PragmARC.Stack_Unbounded
PragmARC.Stack_Unbounded_Unprotected
PragmARC.Three_Way
PragmARC.Transporter_Handler
PragmARC.Universal_Random
PragmARC.Us_Card
PragmARC.Us_Deck
PragmARC.Word_Input
PragmARC.Wrapping

19 Java Byte Code and Mixing Languages

19.1 Ada Meets Java

ACT's JGNAT compiler will compile Gnat source files into Java applications or applets. This section introduces JGNAT.

19.1.1 The Java Virtual Machine

Java is an interpreted language. It is compiled in an artificial machine language for a computer that doesn't exist. This computer is called the Java Virtual Machine (JVM) and the instructions are known as bytecode. The process is similar to the one used by the UCSD Pascal, a popular language from the 1980s that also used a virtual machine language (called P-Code).

When Java is compiled, the source code is converted to a .class file. This file contains the instructions for the JVM.

The JVM language is not directly related to Java. It is possible for other languages to create JVM executables. The JGNAT compiler converts Ada source files into .class files that can be executed by Java.

The official Java Virtual Machine Specification is available at Sun's Java website.

19.1.2 JGnat

Most of the Gnat tools have a corresponding JGnat version, including gnatmake. To compile an Ada program into a Java byte-code program, use jgnatmake:

```
jgnatmake hello
```

Table: jgnatmake switches

JGnatmake Switch	*Description*
-a	Consider all files, even readonly ali files
-c	Compile only, do not bind and link
-f	Force recompilations of non predefined units
-i	In place. Replace existing ali file, or put it with source
-jnum	Use nnn processes to compile
-k	Keep going after compilation errors
-m	Minimal recompilation
-M	List object file dependences for Makefile
-n	Check objects up to date, output next file to compile if not
-o name	Choose an alternate executable name
-q	Be quiet/terse
-s	Recompile if compiler switches have changed
-v	Display reasons for all (re)compilations
-z	No main subprogram (zero main)
--GCC=command	Use this jgnat command

--GNATBIND=command	Use this gnatbind command
--GNATLINK=command	Use this gnatlink command
-aLdir	Skip missing library sources if ali in dir
-Adir	like -aLdir -aIdir
-aOdir	Specify library/object files search path
-aIdir	Specify source files search path
-Idir	Like -aIdir -aOdir
-I-	Don't look for sources & library files in the default directory
-Ldir	Look for program libraries also in dir
-nostdinc	Don't look for sources in the system default directory
-nostdlib	Don't look for library files in the system default directory
-cargs opts	opts are passed to the compiler
-bargs opts	opts are passed to the binder
-largs opts	opts are passed to the linker
-g	Generate debugging information
-Idir	Specify source files search path
-I-	Do not look for sources in current directory
-O[0123]	Control the optimization level
-gnata	Assertions enabled. Pragma Assert/Debug to be activated
-gnatA	Avoid processing gnat.adc, if present file will be ignored
-gnatb	Generate brief messages to stderr even if verbose mode set
-gnatc	Check syntax and semantics only (no code generation)
-gnatd?	Compiler debug option ? (a-z,A-Z,0-9), see debug.adb
-gnatD	Debug expanded generated code rather than source code
-gnate	Error messages generated immediately, not saved up till end
-gnatE	Dynamic elaboration checking mode enabled
-gnatf	Full errors. Verbose details, all undefined references
-gnatF	Force all import/export external names to all uppercase
-gnatg	GNAT implementation mode (used for compiling GNAT units)
-gnatG	Output generated expanded code in source form
-gnath	Output this usage (help) information
-gnati?	Identifier char set (?=1/2/3/4/8/p/f/n/w)
-gnatk	Limit file names to nnn characters (k = krunch)
-gnatl	Output full source listing with embedded error messages
-gnatL	Use longjmp/setjmp for exception handling
-gnatmnnn	Limit number of detected errors to nnn (1-999)
-gnatn	Inlining of subprograms (apply pragma Inline across units)

-gnato	Enable overflow checking (off by default)
-gnatO nm	Set name of output ali file (internal switch)
-gnatp	Suppress all checks
-gnatP	Generate periodic calls to System.Polling.Poll
-gnatq	Don't quit, try semantics, even if parse errors
-gnatR	List representation information
-gnats	Syntax check only
-gnatt	Tree output file to be generated
-gnatTnnn	All compiler tables start at nnn times usual starting size
-gnatu	List units for this compilation
-gnatU	Enable unique tag for error messages
-gnatv	Verbose mode. Full error output with source lines to stdout
-gnatw?	Warning mode. (?=s/e/l/u for suppress/error/elab/undefined)
-gnatW	Wide character encoding method (h/u/s/e/8/b)
-gnatx	Suppress output of cross-reference information
-gnatX	Language extensions permitted
-gnaty	Enable all style checks
-gnatyxxx	Enable selected style checks xxx = list of parameters: • 1-9 check indentation • b check no blanks at end of lines • c check comment format • e check end labels present • f check no form feeds/vertical tabs in source • h check no horizontal tabs in source • i check if-then layout • k check casing rules for keywords, identifiers • m check line length <= 79 characters • Mnnn check line length <= nnn characters • r check RM column layout • s check separate subprogram specs present • t check token separation rules
-gnatz	Distribution stub generation (r/s for receiver/sender stubs)
-gnatZ	Use zero cost exception handling
-gnat83	Enforce Ada 83 restrictions

jgnatmake will create two files: hello.class and ada_hello.class. To run the program under the Java interpreter, type

```
java hello
```

Table: java switches

Java Interpreter Switch	*Description*
`-help`	Print usage info
`-version`	Print version number
`-ss size`	Maximum native stack size
`-mx size`	Maximum heap size
`-ms size`	Initial heap size
`-as size`	Heap increment
`-classpath path`	Set classpath
`-verify`	Verify all bytecode
`-verifyremote`	Verify bytecode loaded from network
`-noverify`	Do not verify any bytecode
`-Dproperty=value`	Set a property
`-verbosegc`	Print message during garbage collection
`-noclassgc`	Disable class garbage collection
`-v, -verbose`	Be verbose
`-verbosejit`	Print message during JIT code generation
`-verbosemem`	Print detailed memory allocation statistics
`-debug`	Trace method calls
`-noasyncgc`	Do not garbage collect asynchronously
`-cs, -checksource`	Check source against class files
`-oss size`	Maximum java stack size
`-jar`	Executable is a JAR

Limitations: Ada streams don't work with Jgnat.

19.2 ASIS

The Ada Semantic Interface Specification (ASIS) is a standard for building development tools for Ada 95. ASIS is implemented as a series of Ada packages and they are included with the Gnat compiler. Debuggers, source code browsers and code checkers can all be written using ASIS.

ASIS works like a database. Different ASIS child packages return different information. A program using ASIS issues queries to the ASIS packages and ASIS returns the query results.

Information and tutorials on ASIS is available from the ASIS Workgroup at http://info.acm.org/sigada/WG/asiswg/asiswg.html.

19.3 Assembly Language

This section discusses embedding assembly language into an Ada 95 program using Gnat. There is a tutorial on Ada assembly language programming is available at http://www.adapower.com/articles.

Optimizing your programs is more than just rewriting your Ada source code. Gnat performs many basic optimizations for you. For example, Gnat will take these statements

```
x := 5;
y := x+1;
```

and rewrite them as

```
x := 5;
y := 6;
```

to eliminate the addition operation.

In order to improve the performance of you source code, you'll have to consider issues that Gnat cannot know beforehand or issues that Gnat does not consider when it compiles:

- **unusual data types**: if a certain variable will always be even or odd, there is no way to express that in an Ada type statement. You may be able to take advantage of that situation by using assembly language. Also, some kinds of instructions, like SSE2 array operations, can be difficult for a language to apply to a program. Your only choice may be to use assembly language.
- **new processor instructions**: gcc (at least at the time of writing this) has no Pentium optimizations. You can use assembly language to take advantage of the latest features of a processor that Gnat may not know about. For example, you can prefetch data into the processor's internal cache.
- **processor designs**: the latest Pentium family processors attempt to run multiple instructions at once and attempt to predict what you will attempt to do next before actually reading the instructions. Gnat is limited in the ways it can reorder (or rewrite) your statements to get the most benefit. To a certain extent, this can also be done at the Ada source level by reordering your statements and/or breaking long statements into short, discrete operations.
- **eliminating high-level overhead**: even with optimizations, Gnat may implement certain high-level features (such as type checks or exception handling) where they are not necessary because it isn't certain that they can be eliminated. Using assembly language, you can explictly instruct the processor to do the operation without implementing these features.
- **unusual optimizations**: some kinds of assembly language tricks, like multiple entry points into a function, simply cannot be expressed in a high-level language like Ada. Some specialized Pentium instructions, like bsr, do not translate easily from a high-level language--that is, there's no Ada function that represents a bsr.

The latest Pentium processors make great effort to reduce the average length of time it takes to execute an instruction, even when it makes programs difficult to optimize. In a sense, the processors take such drastic measures to improve bad source code that it's difficult to write good, efficient source code. Even in Ada, reversing two statements can measurably improve (or degrade) performance on a Pentium family processor.

However, there are other reasons to use assembly language besides optimization:

- **readability**: some algorithms are simply easier to read and understand in assembly language.
- **education**: you want to experiment with the processor and learn how it works.
- **device programming**: you may need to guarantee the order of certain low-level operations.
- **entertainment**: let's face it--it can be a lot of fun.

19.3.1 Pentium Family Processors

The latest Pentium processors are referred to as "IA-32" (32-bit Intel architecture) in the Intel literature. Although the instructions they execute are based on the 80386 processor, you can think of them as hardware emulators that pretend to follow 80386 instructions: internally, the instructions are broken down into different instructions called "microops".

Despite years of extensions, optimizations, and cache increases, the Pentium family is still just a 32-bit 80386 at its core. There are 8 general purpose registers--although they aren't really general purpose since some are more "general" than others. Only six are normally useful:

- **EAX** - accumulator
- **EBX** - data segment pointer
- **ECX** - string and loop counter
- **EDX** - I/O pointer
- **ESI** - string operation source
- **EDI** - string operation destination

The remaining two represent the stack.

- **EBP** - frame base pointer
- **ESP** - stack pointer

Instructions can address less than a register's entire contents. For example EAX can be refered to as AX (lower 16-bits) AH (the second (high) byte), and AL (the first (low) byte). The second four can address only the lower 16-bits by dropping the leading "E" (BP, SI, DI, SP).

There are also 6 segment registers (CS, DS, SS, ES, FS, GS), but you don't normally need to work with these.

An additional register, EFLAGS, contains the status bits for the processor. This represents the results of the last operation or modes that the processor is running in:

bit 0 - carry
bit 1 - always 1
bit 2 - parity
bit 3 - always 0
bit 4 - aux carry (BCD)
bit 5 - always 0
bit 6 - zero flag
bit 7 - sign
bit 8 - trap flag
bit 9 - interrupt enable
bit 10 - direction
bit 11 - overflow
bit 12 - I/O privilege
bit 13 - I/O privilege
bit 14 - nested task
bit 15 - always 0
bit 16 - resume flag
bit 17 - virtual-8086 mode
bit 18 - alignment check
bit 19 - virtual interrupt
bit 20 - virtual interrupt pending
bit 21 - ID flag (has CPUID instruction)

Bits 22-31 are zero.

19.3.2 Instruction Set

The actual instruction set for the Pentium family is too large to list here. The manual is available for download from Intel and is nearly 1000 pages long. They are available at Intel's Web Site. (The instruction set documentation on the Pentium II site covers all IA-32 processors, including Pentium III, 4, and so on.)

If you are using the AMD x86_64 architecture, these links may be helpful:

- http://www.x86-64.org/documentation/assembly
- http://www.x86-64.org/documentation/abi.pdf

Floating point instructions use a separate set of registers and their own instructions usually prefaced by a "F".

The MMX, SSE and SSE2 are collectively referred to as SIMD instructions in the Intel documentation. "SIMD" is an acronym simply meaning an array operation. These instructions load, save and perform operations on part of an array. For example, using MMX instructions, you can load 32 16-bit integers, add them to a second set of 32 16-bit integers, and save the results with only 3 instructions. Since they work on fixed-size blocks, your arrays must be sized accordingly.

Some basic instructions for discussion purposes are:

```
call   - call a subroutine
clc    - clear the carry flag
add    - add without carry
adc    - add with carry
and    - binary and
inc    - increment
jxx    - jump on condition where xx is
                - overflow - O / NO
                - carry - C, B, NAE, NC / NB /NAE
                - equal/zero - E / Z / NE / NZ
                - below and equal / above - BE / NA / A / NBE
                - negative - S / NS
                - parity - P / PE / NP / PO
                - < or >=  - L / NGE / GE / NL
                - <= or > - LE / NG / G / GLE
                - aux carry - U / NU
mov    - load and save registers or transfer memory data
nop    - no operation (do nothing), useful for experimenting
or     - binary or
pop    - pop/pull data from the stack
push   - push data onto the stack
pushfl - push EFLAGS onto the stack
ret    - return from subroutine
sal    - arithmetic shift left (multiply by 2)
sar    - arithmetic shift right (divide by 2)
sbb    - subtract with borrow
stc    - set carry (set a borrow)
sub    - subtract without borrow
xor    - binary exclusive or
```

Most of these basic instructions can take a length suffix indicating the amount of data: b (byte), w (word), l (long). MOVW moves a 16-bit value. INCL increments a 32-bit value. This suffix is different in Linux than in the Intel documentation because Linux uses the AT&T syntax. For example, PUSHFD (push flags double) is PUSHFL (push flags long) in Linux.

Many of the IA-32 instructions are "CISC" instructions containing an instruction that does the equivalent of two or more simplier instructions at one time. For example, "CMOV" performs a test and then moves the data if the test succeeds--implicitly a jump plus a move. Since the lastest processors are heavily pipelined and paralleled, this type of combined instruction has less of an impact on performance than you might think. The processor sees both CMOV and a jump+MOV as the same sequence of microops internally.

19.3.3 Operands

Assembly language instructions take zero or more operands. Here are some examples of operands.

- $0 - an immediate value (ie. a literal 0)
- $0XF - a hexadecimal immediate value
- $x - the location of variable x
- x - the content of variable x
- %%eax - a register
- (%%eax) - indirect addressing
- -4(%%eax) - indirect addressing with a byte offset
- x(%%eax) - indirect addressing with a variable's content as offset

To tie in Ada variables, refer to the variable as %0...%n. The compiler will substitute in the proper operand to access the value of that variable.

In the AT&T assembly language syntax, the order of operands is reversed to that of Intel's literature. For example, to load hex F into EAX, the mov operands would be "$0XF, %eax" not "%eax, $0XF".

19.3.4 System.Machine_Code.Asm

To use assemble language, include the System.Machine_Code package in your Ada source file. This package contains a procedure called "asm" for inserting assembly language instructions into a program. This is very similar to the C language function of the same name.

C: The Ada *Asm* procedure is identical to the C *asm* function, except that Ada uses the syntax for a normal proc

```
asm( "shrl $8, %0" : "=r" (answer) : "r" (operand) : "cc" );
```

becomes

```
Asm( "shrl $8, %0", Unsigned_32'Asm_Input( "r", operand ), Unsigned_32'Asm_Output( "=r",
```

The first and only required parameter (named template =>) is the text, as a string, to be given to the assembler. Expect this to be quite literally saved into a temporary file for the GNU assembler to process. As a result, including any formatting, such as line feeds and tabs, so the assembler will read your instructions properly.

The shortest and safest example of asm is:

```
asm( "nop" ); -- read instruction but do nothing
```

In order for Ada to use your assembly code to do something useful, it needs to know how to interface the Ada variables. The "inputs" and "outputs" parameters do this. These parameters use items created by the special 'Asm_Input and 'Asm_Output attributes.

```
type'asm_input( constraint_string, variable )
type'asm_output( constraint_string, variable )
```

The constraint string tell Ada how to load/save the variables:

- "m" - load/save variable from/to memory. Don't use a register.
- "I" - identifier is a constant
- "a" - variable in EAX
- "b" - variable in EBX
- "c" - variable in ECX
- "d" - variable in EDX

- "S" - variable in ESI
- "D" - variable in EDI
- "t" - top floating point register
- "u" - second-from-top floating point register
- "A" - 8-bit long long value stored in EAX and EDX

These constraints implicitly let Ada know that these registers will be used by you and if it was using them, it will save them prior to executing your assembly code. You don't need to save them yourself.

In addition, there are general constraints that don't specify a particular register:

- "x" - 128-bit SSE register
- "y" - 64-bit MMX register
- "f" - floating point register
- "r" - use register one the basic EAX, EBC, ECX or EDX registers
- "q" - use register one the basic EAX, EBC, ECX, EDX, ESI or EDI registers
- "R" - use any available general register purpose register
- "g" - global--let Gnat can choose memory or a register

These aren't as useful as you might think. When using a general constraint, Gnat doesn't keep track of which registers it has assigned. "r", the constraint for any available register, will likely be the EAX accumulator. Using "r" for two inputs is the same as using "a" twice and will cause one value to overwrite the other.

All output constraints must use a "=" to indicate that it's for output.

For example,

```
result : interfaces.unsigned_32;
...
movl %%eax, %0  ; save result
```

would could be done as

```
outputs => interfaces.unsigned_32'asm_outputs( "=a", result )
-- save EAX accumulator into variable named "result"
```

Multiple input/output parameters are specified as "=> (first, second, ...)".

When counting, the inputs are numbered first. That is, if you have one item in inputs and one item in outputs, the input is %0 and the output is %1.

19.3.5 Other Asm Flags

Another asm parameter, **clobber**, is a string with the names of the registers that need to be saved besides the ones implicilty referred to in the inputs and outputs. Clobber strings can be a register name, "cc" for processor flag or "memory" for a memory location.

The Asm procedure is treated as a normal Ada procedure. During optimization, Gnat may change the order in which the instructions in your program are executed to improve performance. For example, if your Asm procedure is inside a loop, Gnat may move the procedure outside of the loop if it thinks it is save to do so. This is safe to do for an Ada procedure, but an Asm procedure may suffer side-effects and not function correctly. Use the fourth Asm parameter,**volatile**, to indicate to Gnat that it is not safe to move your Asm procedure during optimzation.

For the same reason, you should not use two or more Asm procedures in one block of source code because Gnat may attempt to reorder them. Instead, place all your instructions into one Asm procedure to ensure the instructions will execute in the proper order.

19.3.6 A Complete Example

```
with Ada.Text_IO, Interfaces, System.Machine_Code;
use  Ada.Text_IO, Interfaces, System.Machine_Code;

procedure asm4 is
-- A demonstration of Pentium assembly language programming.
--
-- We'll use the Interface package's unsigned_32 integers for the 32-bit
-- values stored in registers.  Of course, we could have made our own types
-- to do the same thing...

function do_math( value1, value2 : unsigned_32 ) return unsigned_32 is
   -- Do some arbitrary math in assembly language.  We'll use
   -- ( value + 1 ) * 2 - value2 in this example.
   result : unsigned_32;
begin

   asm(

           "incl %%eax" & ASCII.LF & ASCII.HT &      -- increment by 1
           "sall %%eax" & ASCII.LF & ASCII.HT &      -- shift left ( * 2 )
           "subl %%ebx, %%eax",                      -- subtract value2 (ebx)

           -- EAX register := value1;
           -- EBX register := value2;

           inputs => (
                   unsigned_32'asm_input( "a", value1 ),  -- value1 in EAX
                   unsigned_32'asm_input( "b", value2 )   -- value2 in EBX
           ),

           -- result := EAX register;

           outputs => unsigned_32'asm_output( "=a", result ),

           -- The carry flag will be altered in EFLAGS by subl

           clobber => "cc"

   );
   return result;
end do_math;
pragma inline( do_math );

value1 : unsigned_32;
value2 : unsigned_32;
result : unsigned_32;

begin
   value1 := 5;
   value2 := 8;
   result := do_math( value1, value2 );
   put_line( "do_math will do ( value1 + 1 ) * 2 - value2" );
```

```
    put( "do_math(" & value1'img & "," & value2'img & ") = " );
    put_line( result'img );
end asm4;
```

```
do_math will do ( value1 + 1 ) * 2 - value2
do_math( 5, 8) =  4
```

19.3.7 Assembly to Ada

It is possible to call Ada from assembly language. I haven't tried this. Some posts on comp.lang.ada suggest that using C "shims" is useful: that is, creating C functions to setup Ada as opposed to trying to do this from assembly language, and then call the C functions instead of the Ada function directly.

19.4 Calling Ada from C

It is possible to combine Ada 95 with a main program written in C. Using Ada 95 classes, functions and procedures from another language is more difficult than the reverse process. While the GNAT compiler has a lot of support for other languages, the other languages do not supply the same level of support. Be prepared to do some manual chores in order to compile and link your program.

It is best to use the same GCC compiler for all the source files. Both the ACT and ALT versions of GCC have the C language enabled.

As discussed under types in this document, most C types have direct correspondence to Ada types. A C "int" is the same as an Ada "integer". For greater portability, the Interfaces.C package and its children contain the definitions of many standard C types. Arrays and records are directly equivalent to C arrays and structures. Special cases are noted below.

Before calling any Ada 95 subprograms, the C program should call the function **adainit** which performs the initializations and elaborations for the Ada 95 source code. Before the C program exits, it should call **adafinal** to perform any cleanup. These functions are created by gnatbind so you cannot create a C test program without creating at least one an Ada source file as well.

Suppose you want to call a single Ada procedure with no parameters. Your C main program would look something like this.

```
// main.c
//

#include

extern void adainit( void );
extern void adafinal( void );

int main( int argc, char **argv) {

    puts("C main() started.");
    adainit();

    ada_subroutine();

    adafinal();
```

```
    puts("adafinal() returned.");

    return 0;
}
```

Create an Ada package containing the "ada_subroutine" procedure. The procedure should be exported to C using *pragma export*. Because C is a case sensitive language, pragma export will convert the procedure name to lower case characters. (There's another pragma that can change how the case conversion is performed.) Alternatively, you can explicitly supply a new C name in pragma export.

```
package Test_Subr is

    procedure Ada_Subroutine;
    pragma export(C, Ada_Subroutine);

end Test_Subr;
```

```
with Ada.Text_IO;
use Ada.Text_IO;

package body Test_Subr is

    procedure Ada_Subroutine is
    begin
        Put("Ada_Subroutine has been invoked from C.");
    end Ada_Subroutine;

end Test_Subr;
```

To build the project:

1. Compile the C and Ada files separately.
2. Because the project contains Ada source, it will have to be bound using gnatbind -n. The -n switch indicates there is no Ada main program.
3. Compile file generated by gnatbind. (This contains adainit() and adafinal().)
4. Link with gnatlink.

The following example uses the ALT GNAT 3.13p:

```
$ gnatgcc -c main.c
$ gnatgcc -c test_subr.adb
$ gnatbind -n test_subr
$ gnatgcc -c b~test_subr
$ gnatlink -o main main.o test_subr.ali
$ ./main
C main() started.
Ada_Subroutine has been invoked from C.
adafinal() returned.
```

Functions can likewise be exported.

```
function Times_2( i : integer ) return integer;
pragma export(C, Times_2);
```

Although the exported subprograms don't need to be prototyped (declaring the function headers), all subprograms should be prototyped in the same way that external C functions are prototyped. Prototyping ensures that the functions will be called with the proper parameters.

```
extern int times_2( int );
```

Declaring a function in Ada doesn't ensure that the parameters are strongly typed. The parameters are set up by C prior to calling the Ada function.

```
printf( "3 times 2 is %d\n", times_2( 3.0 ) );
```

will compile and return 6, just as if times_2 was a C function.

If Times_2 is overloaded, exporting it will cause an "already defined" error during linking. You will have to provide pragma export with alternate C names that won't conflict with each other.

```
function Times_2( i : integer ) return integer;
pragma export(C, Times_2, "int_times_2");

function Times_2( f : float ) return float;
pragma export( C, Times_2, "float_times_2" );
```

If the overloaded Ada function differs only in return value, only one version is exported. You will have to use trial-and-error to determine which one. If you use "rename" to create alternate namings of functions for C++, Ada will not allow you to use pragma export with the renamed functions. If the original function was exported, Ada considers the renamed function to be already exported under the old name.

Default parameters are not allowed for non-Ada languages. C++ must explicitly include the optional parameter.

Inline functions can be exported to C++: the inline process doesn't affect their ability to be called by an outside language.

Variables can also be exported.

```
type a_test_record is record
    i : integer := 3;
    f : float := 1.5;
end record;
test_record : a_test_record;
pragma export(C, test_record );
```

Define the Ada record as a C structure.

```
struct a_test_record {
    int i;
    float j;
};
extern struct a_test_record test_record;
```

You can now access the fields of the Ada record from your C program.

```
printf( "test_record.i is %d\n", test_record.i );
printf( "test_record.j is %f\n", test_record.j );
```

```
test_record.i is 3
test_record.j is 1.500000
```

Likewise **arrays** can be exported. Remember that in C, array indices always start at zero.

```
type a_test_array is array(1..5) of character;
test_array : a_test_array := ('a','b','c','d','e');
```

```
pragma export(C, test_array );
```

```
extern char test_array[5];
...
printf( "test_array[0] is %c\n", test_array[0] );
printf( "test_array[4] is %c\n", test_array[4] );
```

```
test_array[0] is a
test_array[4] is e
```

Certain variable types are more difficult to deal with. For example, Ada **enumerated types** must be declared with "pragma convention(C" to be compatible with C's enumerated type. This is the only case where Gnat stores a type differently for the benefit of another language.

```
type ada_enum_type is (red, green, blue);
pragma convention( C, ada_enum_type );

ada_enum : ada_enum_type := green;
pragma export( C, ada_enum );
```

```
enum ada_enum_type {red, green, blue};
extern enum ada_enum_type ada_enum;
...
if ( ada_enum == green )
    printf( "ada_enum is green\n" );
else
    printf( "ada_enum isn't green\n" );
```

```
ada_enum is green
```

The contents of arrays or records packed with pragma pack are not easily accessible from C. Ada represents these as raw bits.

Ada unconstrained arrays (including unconstrained strings) have no equivalent in C. The easiest work around for strings is to use C strings instead, as defined in the Interfaces.C package. For C to call a subprogram with unbounded Ada strings as parameters, the C program must also pass the string bounds as parameters. [If anyone knows how to do this, email me.--KB]

Ada access types are usually equivalent to C pointers.

Packages

If you want to export a **private items**, use pragma export in the private section of the package, not at the forward declaration.

If you want to export something from a **package**, the pragma export must appear in the same package. The names of the items to be exported must be local names and cannot be referred to as "package.item".

For **generic packages**, if you export items in the generic package package will have the same external name. You can't create unique names by using pragma export in the place where the generic package is instantiated because pragma export can only appear in the same declaration section as the items being exported. For example, suppose you have a generic linked list package

and you export the "list" record to C++. If you instantiate a boolean list, an integer list and a string list, only one of those will be available to "C++" as a "list" structure...probably whichever one Ada instantiated last will overwrite the previous definitions of "list".

To work around this problem, you will have have to create "wrapper" types and subprograms. For example, create a record to hold a variable from the generic package and create short subprograms with new names that call instantiated subprograms. For example, if "strlist" is instantiated package for lists of strings:

```
-- wrapping a strlist.list in a record and exporting the record
type C_List is record
        list : strlist.list;
end record;
pragma export( C, C_List );

-- wrapping the strlist.sort procedure in another procedure and exporting
procedure C_List_Sort( list : C_List );
pragma export( C, C_List_Sort );
...
procedure C_List_Sort( the_list : C_List ) is
begin
      strlist.sort( the_list.list ); -- run on behalf of C
end C_List_Sort;
```

19.5 Calling C++ from Ada

The Ada 95 standard doesn't support interfacing to C++, but Gnat provides extensions so that GNU C++ source code can be used with Ada.

If possible, the same GCC compiler should be used for all source files. If you are using the ACT version of GNAT 3.x or earlier, you should recompile GNAT to enable C++. The ALT version of GNAT 3.x (usually) has C++ enabled. If C++ is not enabled, you will receive a message from GCC about "cc1plus" (the C++ compiler) not being found. However, for the complete example below, I used two different version of GCC and had no problems.

For the most part, calling C++ from Ada is done the same was as calling C from Ada. Instead of using "C" in pragma import, use "CPP" (that is, C++) as the language convention. However, Gnat provides no support for C++ "name mangling": all C++ declarations should use extern "C" to stop the name mangling. (If you are an adventurer, use the dumpobj -t command to determine the symbol names used by C++ and include them explicitly in pragma import.)

If Ada and C++ use different GCC compilers, the linker may not be able to tell which version of libgcc to use. You can check which library is being used with the gnatlink -v -v (very verbose) switches.

Importing C++ objects into Ada is possible but difficult. C++ and Ada implement objects using different Object Oriented Programming models. C++ objects are not identical to tagged records. Gnat has special pragmas for importing C++ objects:

- **CPP_Class** - indicates that the record or tagged record type corresponds to a C++ class. This pragma also declares the type as "limited" (it cannot be assigned) and disables certain Ada features ('tag will not work--a C++ object has no tag)
- **CPP_Constructor** - declares and imports a C++ constructor

- **CPP_Virtual** - identifies which C++ methods are declared as virtual and what position in the dispatch table contains the method. Also provides a way to declare non-overridden virtual functions in Ada.
- **CPP_VTable** - identifies which Ada record field represents the C++ vtable (virtual method dispatch table). The final field in an imported C++ class should be a vtable. The vtable type is declared in the Interfaces.CPP package.

The Gnat C++ interface proposal also has a **CPP_Destructor** pragma, but this has not been implemented. [Perhaps it is not necessary? --KB]

There are two naming problems. First, name mangling is necessary with C++ classes. You will have to use objdump to determine the C++ method names. Second, if two C++ methods from two different classes have the same name (this is often the case when overridding), you'll have to declare the C classes in separate Ada packages. Otherwise, pragma import will report an error when attempting to import the same name twice.

Suppose you want to import a C++ car class named cpp_car.

```
type cpp_car is tagged record
      year      : integer;                  -- year of car
      weight    : integer;                   -- weight of car
      length    : integer;                  -- length of car
      car_vtable : Interfaces.CPP.Vtable_Ptr;   -- always the last field
end record;
pragma CPP_Class( cpp_car );                -- this is a C++ class
pragma CPP_Vtable( cpp_car, car_vtable, entry_count => 3 ); -- car_vtable has 3 virtual methods
```

If a class has no vtable, it cannot be imported in Ada. CPP_Class will report an error.

Only the C++ classes you intend to use need to be imported. If cpp_car has a parent class called cpp_vehicle, it does not have to be declared in Ada if it will not be used. However, any fields in cpp_vehicle will have to be added to the beginning of the cpp_car tagged record or they will be missing.

C++ classes imported into Ada can't be assigned. This has to do with the differences in assignment semantics between the two languages. C++ classes used in Ada should always have a constructor because this is the only way to assign values to the object.

```
function Default_Constructor return cpp_car'class;
pragma CPP_Constructor( Default_Constructor );
-- the default constructor for cpp_car
```

The name of the constructor is not important. It simply gives the C++ constructor an Ada compatible name.

Other constructors can be imported using different parameters.

```
function Copy_Constructor( c : cpp_car'class ) return cpp_car'class;
pragma CPP_Constructor( Copy_Constructor );
-- construct a copy of a cpp_car object
```

Methods that are not virtual can be imported without a special pragma.

```
function get_year( c : cpp_car'class ) return integer;
pragma import( CPP, get_year );
```

C++ has no object parameter--it is implied. In Ada, the object must be declared and it can be declared in any position in the parameter list. When importing C++ objects, always put the object name in the position of the first parameter. This is the parameter used by C++ (even though it is not seen by the programmer).

One of the differences between C++ and Ada objects is that Ada has no equivalent of "virtual methods". In Ada, whether or not a method is virtual is determined by the way the class is declared. Also, Ada doesn't allow a class-wide type to be overridden by any children--a class-wide type is always class-wide with no hidden surprises further down the class tree. In C++, virtual functions must be explicitly declared as "virtual". Ada doesn't require all the methods in a C++ class to be imported.

pragma CPP_Virtual identifies which methods are virtual and the position in the vtable. In the simplest case, the first C++ virtual function is at position 1, the second is at position 2, and so on.

```
function get_total_weight( c : cpp_car ) return integer;
pragma import( CPP, get_total_weight );
pragma CPP_Virtual( get_total_weight, car_vtable, 1 );
-- first virtual function in class
-- child tagged records may override this function

function get_total_length( c : cpp_car ) return integer;
pragma import( CPP, get_total_length );
pragma CPP_Virtual( get_total_length, car_vtable, 2 );
-- second virtual function in class
-- child tagged records may override this function
```

When a virtual function is not overridden, it must be declared in Ada. Use CPP_Virtual to indicate which parent function to use.

```
type cpp_luxury_car is new cpp_car ...

function get_total_weight( c : cpp_luxury_car ) return integer;
pragma import( CPP, get_total_weight );
pragma CPP_Virtual( get_total_length, car_vtable, 3 );
-- overridden virtual function

function get_total_length( c : cpp_luxury_car ) return integer;
pragma import( CPP, get_total_length );
pragma CPP_Virtual( get_total_length, car_vtable, 2 );
-- not overridden virtual function
-- in C++, this function does not appear.  It is implied.
-- for a cpp_luxury_car, use cpp_car get_total_length in vtable at position 2
```

If necessary, Gnat allows the C++ class to be extended with Ada-specific tagged records. (A multi-language class may make a project unnecessarily complex and difficult to debug.)

Private and protected fields in a C++ object and be simulated using a combination of Ada's **private** keyword and the information hiding capabilities of Ada packages.

To build the project:

1. Compile the C++ files with the c++ command
2. Make the Ada project with gnatmake
3. Bind the Ada project with gnatbind -x
4. Link the project using gnatlink. Include the C++ library (-lstdc++) and direct Gnat to use the c++ linker (--link=c++). Use -L, if necessary, to specify the stdc++ directory

Here is a complete example using two simple C++ objects.

```
// c_class.h

   /* Unimaginative class declaration */

   class c_root_class {
      public:
         int i;
```

```
        c_root_class( void );
        int get_value ( void ) const;
        void set_value( int new_i );
        virtual int get_total_value( void ) const;
};
class c_extended_class: c_root_class {
    public:
        int j;
    c_extended_class( void );
    int get_j_value ( void ) const;
    void set_j_value( int new_j );
    virtual int get_total_value( void ) const;
};
```

```
// c_class.cc
//
#include
#include "c_class.h"

/* c_root_class: Method Bodies */

    c_root_class::c_root_class( void ) {
        i = 0;
    }

    int c_root_class::get_value( void ) const {
        return i;
    }

    void c_root_class::set_value( int new_i ) {
        i = new_i;
    }

    int c_root_class::get_total_value( void ) const {
        return i;
    }

/* c_extended_class: Method Bodies */

    c_extended_class::c_extended_class( void ) {
        j = 0;
    }
    int c_extended_class::get_j_value( void ) const {
        return j;
    }
    void c_extended_class::set_j_value( int new_j ) {
        j = new_j;
    }
    int c_extended_class::get_total_value( void ) const {
        return i+j;
    }
```

Now determine the C++ mangled names with objdump.

```
$ c++ -Wall -c c_class.cc
```

```
$ objdump -t c_class.o
c_class.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 c_class.cc
00000000 l    d  .text  00000000
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000 l       .text  00000000 gcc2_compiled.
00000000 l    d  .gnu.linkonce.d.__vt_16c_extended_class       00000000
00000000 l    d  .gnu.linkonce.d.__vt_12c_root_class    00000000
00000000 l    d  .rodata        00000000
00000000 l    d  .gnu.linkonce.t.__tf12c_root_class     00000000
00000000 l    d  .gnu.linkonce.t.__tf16c_extended_class 00000000
00000000 l    d  .note  00000000
00000000 l    d  .comment       00000000
00000000 g     F .text  00000015 __12c_root_class
00000000  w    O .gnu.linkonce.d.__vt_12c_root_class    00000010 __vt_12c_root_class
00000018 g     F .text  0000000c get_value__C12c_root_class
00000024 g     F .text  0000000d set_value__12c_root_classi
00000034 g     F .text  0000000c get_total_value__C12c_root_class
00000040 g     F .text  00000029 __16c_extended_class
00000000  w    O .gnu.linkonce.d.__vt_16c_extended_class       00000010 __vt_16c_extended_class
0000006c g     F .text  0000000d get_j_value__C16c_extended_class
0000007c g     F .text  0000000e set_j_value__16c_extended_classi
0000008c g     F .text  00000011 get_total_value__C16c_extended_class
00000000  w    F .gnu.linkonce.t.__tf16c_extended_class 00000034 __tf16c_extended_class
00000000  w    F .gnu.linkonce.t.__tf12c_root_class     0000002b __tf12c_root_class
00000008       O *COM*  00000004 __ti12c_root_class
00000010       O *COM*  00000004 __ti16c_extended_class
00000000         *UND*  00000000 __rtti_user
00000000         *UND*  00000000 __rtti_class
```

Write the corresponding Ada packages containing the C++ class interface. Since virtual methods are used, we'll need to define each C++ class in a separate package to avoid problems with pragma import.

```
with Interfaces.CPP;

package c_root_class_package is

   type c_root_class is tagged record
         i : integer;
         vtable : Interfaces.CPP.VTable_Ptr; -- C++ vtable
   end record;
   pragma CPP_Class( c_root_class );
   pragma CPP_Vtable( c_root_class, vtable, entry_count => 2 );

   function default_constructor return c_root_class'class;
   pragma import( CPP, default_constructor, "__12c_root_class" );
   pragma CPP_Constructor( default_constructor );

   function get_value( cr : c_root_class'class ) return integer;
   pragma import( CPP, get_value, "get_value__C12c_root_class" );

   procedure set_value( cr : c_root_class'class; new_i : integer );
   pragma import( CPP, set_value, "set_value__12c_root_classi" );

   function get_total_value( cr : c_root_class ) return integer;
```

```
    pragma import( CPP, get_total_value, "get_total_value__C12c_root_class" );
    pragma CPP_Virtual( get_total_value, vtable, 1 );

end c_root_class_package;
```

```
with Interfaces.CPP;
with c_root_class_package; use c_root_class_package;

package c_extended_class_package is

    type c_extended_class is new c_root_class with record
       j : integer;
    end record;
    pragma CPP_Class( c_extended_class );

    function get_j_value ( ce : c_extended_class'class ) return integer;
    pragma import( CPP, get_j_value, "get_j_value__C16c_extended_class" );

    procedure set_j_value( ce : c_extended_class'class; new_j : integer );
    pragma import( CPP, set_j_value, "set_j_value__16c_extended_classi" );

    function extended_constructor return c_extended_class'class;
    pragma import( CPP, extended_constructor, "__16c_extended_class" );
    pragma CPP_Constructor( extended_constructor );

    function get_total_value( cr : c_extended_class ) return integer;
    pragma import( CPP, get_total_value, "get_total_value__C16c_extended_class" );
    pragma CPP_Virtual( get_total_value, vtable, 2 );

end c_extended_class_package;
```

The main program will declare two objects and test the methods.

```
with Interfaces.CPP;
with Ada.Text_IO; use Ada.Text_IO;
with c_root_class_package; use c_root_class_package;
with c_extended_class_package; use c_extended_class_package;

procedure main is

    object : c_root_class;
    extended_object : c_extended_class;

begin
    put_line( "Ada Using a C++ Class" );
    new_line;
    put_line( "With object (a c_root_class class object):" );
    put_line( "  object constructor should assign value 0, value is" & get_value( object )'img );
    set_value( object, 12 );
    put_line( "  set_value( object, 12 ), value is" & get_value( object )'img );
    put_line( "  object total value (virtual method) is" & get_total_value( object )'img );
    new_line;
    put_line( "With extended_object (a c_extended_class class object):" );
    put_line( "  extended object constructor should assign j value 0, value j is" & get_j_value( extended_object )'img );
    set_value( extended_object, 12 );
    put_line( "  set_value( extended_object, 12 ), value is" & get_value( extended_object )'img );
    set_j_value( extended_object, 15 );
```

```
    put_line( "  set_j_value( extended_object, 15 ), j value is" & get_j_value( extended_object )'img );
    put_line( "  extended object total value (virtual method) is" & get_total_value( extended_object )'img );
    new_line;

    put_line( "Ada finishes" );
end main;
```

Build and run the project.

```
$ gnatmake -c main.adb
$ gnatbind -x main.ali
$ gnatlink main c_class.o -lstdc++ -L/usr/lib/gcc-lib/i386-redhat-linux/2.96 --link=c++
$ ./main
Ada Using a C++ Class

With object (a c_root_class class object):
    object constructor should assign value 0, value is 0
    set_value( object, 12 ), value is 12
    object total value (virtual method) is 12

With extended_object (a c_extended_class class object):
    extended object constructor should assign j value 0, value j is 0
    set_value( extended_object, 12 ), value is 12
    set_j_value( extended_object, 15 ), j value is 15
    extended object total value (virtual method) is 27

Ada finishes
```

19.6 Calling Ada from C++

A C++ program can interface to Ada in the same way as C program.

If possible, the same GCC compiler should be used for all source files. If you are using the ACT version of GNAT 3.x or earlier, you should recompile GNAT to enable C++. The ALT version of GNAT 3.x (usually) has C++ enabled. If C++ is not enabled, you will receive a message from GCC about "cc1plus" (the C++ compiler) not being found. However, the example below was compiled with two different versions of GCC and there were no errors.

For the most part, calling Ada from C++ is done the same was as calling Ada from C. Instead of using "C" in pragma export, use "CPP" (that is, C++) as the language convention. However, Gnat provides no support for CPP "name mangling": all Ada extern declarations in a C++ file should use extern "C".

If Ada and C++ use different GCC compilers, the linker may not be able to tell which version of libgcc to use. You can check which library is being used with the gnatlink -v -v (very verbose) switches.

The following is the same sample C program used above in 19.4, converted to C++.

```
// main.cc
//
#include

extern "C" {
    void adainit(void);
    void adafinal(void);
```

```
    void ada_subroutine( void );
}

int main(int argc, char **argv) {

    puts("C++ main() started.");
    adainit();

    ada_subroutine();

    adafinal();
    puts("adafinal() returned.");

    return 0;
}
```

The Ada package is the same except that convention CPP is used.

```
package Test_Subr is

      procedure Ada_Subroutine;
      pragma export(CPP, Ada_Subroutine );

end Test_Subr;
```

```
with Ada.Text_IO;
use Ada.Text_IO;

package body Test_Subr is

      procedure Ada_Subroutine is
      begin
          Put("Ada_Subroutine has been invoked from C++.");
      end Ada_Subroutine;

end Test_Subr;
```

The steps to build the project are similar to C:

1. Compile the C++ and Ada files separately.
2. Because the project contains Ada source, it will have to be bound using gnatbind -n. The -n switch indicates there is no Ada main program.
3. Compile file generated by gnatbind. adafinal()).
4. Link with gnatlink and use --link=c++ switch. This switch ensures that the C++ elaboration (such as calling constructors on global objects) is done correctly.

```
$ c++ -c test.cc
$ gnatgcc -c test_subr
$ gnatbind -n test_subr
$ gnatgcc -c b~test_subr
$ gnatlink -o main test.o test_subr.ali --link=c++
$ ./main
C++ main() started.
Ada_Subroutine has been invoked from C++.
adafinal() returned.
```

Tagged records cannot be exported directly to C++. Gnat does not understand C++ name mangling and it cannot give the tagged record subprograms names that C++ would recognize. Also, C++ does

not understand Ada's Object Oriented Programming model--Ada tagged records are not identical to C++ object classes.

In order to use Ada tagged records from C++, you will have (dynamically) declare the objects in Ada and pass a "handle" (an ID number or a pointer) back to C++ to use in reference. The Ada source must have special subprograms to match the handle to a particular object and call the appropriate Ada subprogram for that object on behalf of C++.

19.7 Calling Ada from Java

It should be possible to call Ada using Java's C++ importing features.

19.8 GCC GNAT for Microcontrollers

There is a SourceForget project called AVR-Ada to create a version of Ada based on GCC Ada to compile programs for embedded devices with AVR microcontrollers.

20 Developing Your Project

20.1 The Project Proposal

Before you begin any project that will be released to the public, it's a good idea to draw up a proposal. The proposal should be about one page document describing the purpose of the project, who it's being made for, and how long it will take and what kind of investments (time, money or otherwise) you expect. This is especially important if there is anybody working with you. Don't assume your teammates see the project in exactly the same was as you do: write a proposal to avoid misunderstandings.

For example, calling a project "a database" doesn't say much. Calling it a "fast, distributed database for businesses" tells your teammates where the database will be used, gives them an idea about the features required, and tells that the design emphasis is on execution speed.

Once your proposal is finished, bounce the ideas of a few people you respect and trust, especially if they are potential users of your program. If none of them think the project is practical, you may want to change the target audience or features of your project, or chose another project altogether.

You can later use your proposal as the basic text for an announcement of the release of your program.

20.2 The Design Phase

When it comes time to begin designing the basic layout of a project, remember that Ada has features designed just for this task.

Break up your project into a series of packages, and include basic type definitions and subprograms (using pragma import(stubbed). Remember that the design doesn't have to be perfect, but you need a starting place for you and your teammates to discuss the work. Use lots of comments to avoid continually explaining the purpose of each package and it's contents.

When you have a basic layout, compile each of the specs to make sure the design is sound.

20.3 The Development Phase

Check list:

Did you use pragma pure, preelaborate or no_elaboration_code whenever possible?

Did you use pragma Initialize_Scalars whenever possible?

20.4 The Alpha/Beta Release

Check list for first alpha or beta release:

Check your integers: did you use integer when short_integer or long_integer would have been better?

Do you have pragma Optimize set in all of your packages?

Did you use pragma Pack all arrays and records that need packing? Do some need packing turned off?

Did you assign your access types to a debug pool in order to check for run-time errors?

20.5 Releasing Your Software

Check list:

Did you remove all pragma Normalize_Scalars?

Did you remove all access type references to debug pools?

20.5.1 A Third Party Library

If you want to release a package as a third party library:

Change your .ads files to read-only with **chmod -w**.

Collect your executables into an archive with the **ar** command (see the section on libraries above).

Include instructions for installing the archive and make sure you mention that those who use your library must use the -f option for gnatmake. This option treats all read-only files as third party libraries that cannot be recompiled because the package bodies were not included.

20.6 Distribution Formats

20.6.1 RPM: Red Hat Package Manager

RPM (Red Hat Package Manager) is the most popular installation tool. It installs, uninstalls, and checksums packages. S.u.S.E.'s YaST (Yet Another Setup Tool) works using RPM. RPM files end in ".rpm". Full details on the RPM format are available from Red Hat's RPM site at http://www.rpm.org.

The **-q** command checks for a package. **-a** shows all installed packages.

[root@redbase /root]# rpm -q uucp

uucp-1.06.1-14

[root@redbase /root]# rpm -q kernel

kernel-2.0.32-2

[root@redbase /root]# rpm -q -a

setup-1.9.1-1

filesystem-1.3.1-2

basesystem-4.9-1

AnotherLevel-0.5-2

ldconfig-1.9.5-2

…

XFree86-Mach64-3.3.1-14

Creating a new RPM archive is a cumbersome, multistep process.

20.6.2 TGZ Packages

TGZ (tar-ed gzip packages) are created by collecting all the files into one file using **tar** (tape archiver) and compressing the file **gzip** (GNU zip) or tar's **z**(compress) option. The files are usually named with ".tgz" ending but sometimes have the ".tar.gz" longform ending.

To create a new .tar archive, use the "cfv" options.

```
tar cfv archivename file
```

To add additional files to the archive, use "rfv".

```
tar rfv archivename file
```

When the tar file is finished, compress it with gzip

```
gzip —9 archivename.tar
```

And rename it to .tgz

```
mv archivename.tar.gz archivename.tar.gz
```

20.6.3 TAR.BZ2 Packages

TAR.BZ2 (tar-ed bzip packages) are another option. Like TGZ, these are tar files that are compressed, but instead use the new **bzip2** command that compresses better than gzip.

Other Formats

TZ is an older format, these are tar files that are compressed with the old compression command, **compress**.

ZIP packages are collected and compressed in the popular PC zip format using **zip**.

ZOO packages use an older compression program, **zoo**.

CPIO (Copy In-Out) is another archiving program similar to tar. It collects files but doesn't compress them.

DEB is a package for the Debian distribution.

There are a host of other tools and formats, including ones to create archives for other platforms.

20.7 Man Pages

Linux man pages are special text files formatted for the groff program (GNU run off) is based on the older UNIX programs troff (for printers) and nroff (for terminals). Troff was originally created in 1973 by Joseph F. Ossanna.

man pages are text files containing groff markup codes embedded in the text. These codes, much like HTML tags in a web page, control the fonts, layout and graphics used in the pages. You can also define your own groff codes (using groff macros).

Here's an example of a man page with groff markup codes:

```
.\"This is a comment
.TH MAN 7 "25 July 1993" "Linux" "Linux Programmer's Manual"
.SH NAME
man \- macros to format man pages
.SH SYNOPSIS
.B groff \-Tascii \-man
.I file
```

Here, the ".B" groff code indicates that the text that follows should be bold (similar to <b>), and the ".SH" groff code incidcates the text that follows is a subheading (similar to <h*n*>).

The groff predefined macros pretaining to manual pages are documented in the section 7 manual page on man ("man 7 man").

All the man pages are stored in subdirectories in /usr/man. The subdirectories are numbered, each number representing a different section number of the Linux manual. The manuals sections include:

1. Linux introduction
2. System Calls
3. C Library Calls
4. Summaries and Data Structures

For example, the C library call manual pages are located in /usr/man/man3.

The easiest way to create a simple man page for your program is to find a similar man page and make a copy. Use this copy as a basis for your new man page. You can perform a simple test on your new man page by

```
groff mypage | less
```

To convert your page to another format, use

groff mypage > mypage.ps

to create a PostScript version of your man page (or use the -Tdvi switch to create a TeX .dvi file). Use one of the free conversion programs available on the Internet to translate the PostScript file to another format.

20.8 Linux Software Map Entry

The Linux Software Map (http://www.ExecPC.com/lsm/) is a web site devoted to tracking Linux software. Software registered with the map uses a LSM (Linux Software Map) file to describe programs. One important Linux site, Metalab (http://metalab.unc.edu), requires a LSM file for every program in its archive.

A Linux Software Map entry is a text file ending with a ".lsm" suffix. It's formatted like an email message header. There are named fields that begin with a keyword and a colon, followed by the data for that field. Continue lines by pacing over beneath the previous line.

Here is an example LSM entry:

Begin3

Title: YAK - Bulletin Board System for Linux

Version: 1.08b

Entered-date: 09JUN97

Description: BBS software with sources for DOS, OS/2 and Linux. Includes also

tosser and tick program without sources.

Keywords: yak bbs tosser conference bulletin board

Author: skyreader@fw.nullnet.fi (Timo Sirainen)

Primary-site: Skyliner BBS +358-15-176242

Alternative-site: sunsite.unc.edu

Platform: DOS, OS/2, Linux, ...

Copying-policy: GPL

End

Here is the LSM entry for System Manager in a Box 0.9.1 (beta):

Begin3

Title: System Manager in a Box

Version: 0.9.1 (beta)

Entered-date: Wednesday, May 26, 1999

Description: Linux configuration and administration utility using AI techniques.

PegaSoft home page is http://www.pegasoft.ca

Keywords: system administration box ai pegasoft

Author: pegasoft@pegasoft.ca (PegaSoft Canada)

Maintained-by: pegasoft@pegasoft.ca (PegaSoft Canada)

Primary-site: metalab.unc.edu /pub/Linux/system/admin

700kB smiab-0.9.1.tgz

Alternate-site:

Original-site:

Platforms:

Copying-policy: freeware

End

[KB-platform or platforms?]

Details about the format are available from the LSM web site.

To register a program with the Linux Software Map, email your LSM entry to 'lsm@execpc.com' with the subject 'add'.

20.9 Software Licensing Options

The following is a very simplistic overview of the basic licensing options for Linux:

Commercial — sold for money, with warranty. Windows 95 is commercial, as are most programs that run on it.

Free/Freeware — free for all use, usually has no warranty.

GPL (GNU Public License) — free for use and no warranty. If it's a programming tool, you can only incorporate it into your programs to create more GPL software. In other words, GPL is free public software that can only be used to make more free public software. Imagine a free engine for cars. If any car is built to take that free engine, it must be sold for free as well.

LGPL (Library GPL) — same as GPL. Commercial programs may only use it if it's shared, not statically linked. A car can be sold with no engine in it, and the engine can be added separately by the dealer, but you can't sell the car with the free engine factory-installed.

Shareware — commercial software that's sold on the honor system: people who like the software and who use it are expected to send in a cheque to the author. There's a lot of shareware for Windows.

Xfree86 uses a different licence that's compatible with GPL/LGPL.

Virtually all the standard C libraries are LGPL, including libc, but you should check to documentation or C header files to make sure.

Details on these and other licensing options, and how they interact, are described in the book *Linux Application Development* from Addision-Wesley-Longman.

Appendix A: The Linux Shell

The default Linux shell is bash. Here's a summary of common bash shell commands.

ls— lists the files in the current directory

```
[root@armitage temp]# ls
typescript
```

touch — create a new, blank file. If the file exists, changes the time it was last modified but otherwise leaves the file unchanged.

```
[root@armitage temp]# touch temp.txt
[root@armitage temp]# ls
temp.txt    typescript
```

rm — permanently remove a file

```
[root@armitage temp]# rm temp.txt
rm: remove `temp.txt'? y
```

Note: Red Hat defines aliases for rm, mv and cp that prompt before they overwrite or erase a file. Most other distributions use the default behaviour, which is to take action without warning. You can disable Red Hat's aliases with the unalias command.

mv — change the name of a file, or move it to a new location

```
[root@armitage temp]# touch temp.txt
[root@armitage temp]# mv temp.txt temp2.txt
[root@armitage temp]# ls
temp2.txt   typescript
```

cp — copy a file

```
[root@armitage temp]# cp temp2.txt temp3.txt
[root@armitage temp]# ls
temp2.txt   temp3.txt   typescript
```

grep — search a file for a word or phrase

```
[root@armitage temp]# grep "procedure" /home/ken/ada/basicio2.adb
procedure basicio2 is
```

find — search for a file

```
[root@armitage temp]# find /home/ken -type f -name basicio3.adb
/home/ken/ada/basicio3.adb
```

lpr — print a file

```
[root@armitage temp]# lpr basicio3.adb
```

lprm — stop printing a file, if the file hasn't started printing yet

```
[root@armitage temp]# lprm
dfA017Aa01370 dequeued
cfA017Aa01370 dequeued
```

lpq — list your files waiting to be printed

```
[root@armitage temp]# lpq
no entries
```

cat — display a file

```
[root@armitage temp]# cat hello.adb
with Ada.Text_IO;
use Ada.Text_IO;

procedure hello is
begin
    Put_Line( "Hello world!" );
end hello;
```

less — display a file one screen at a time,
allowing you to move around

```
[root@armitage temp]# less basicio.adb
```

tr — translate characters. To translate a DOS text file to a Linux text file, use

```
tr —d '\r' < dos.txt > linux.txt
```

Appendix B: Linux Error Codes

Linux numeric error codes as defined by the Linux kernel.

C Name	*Value*	*Description*
EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Arg list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No child processes
EAGAIN	11	Try again
ENOMEM	12	Out of memory
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Device or resource busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a tty device
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of func

ERANGE	34	Math result not representable
EDEADLK	35	Resource deadlock would occur
ENAMETOOLONG	36	File name too long
ENOLCK	37	No record locks available
ENOSYS	38	Function not implemented
ENOTEMPTY	39	Directory not empty
ELOOP	40	Too many symbolic links encountered
EWOULDBLOCK	same as EAGAIN	Operation would block
ENOMSG	42	No message of desired type
EIDRM	43	Identifier removed
ECHRNG	44	Channel number out of range
EL2NSYNC	45	Level 2 not synchronized
EL3HLT	46	Level 3 halted
EL3RST	47	Level 3 reset
ELNRNG	48	Link number out of range
EUNATCH	49	Protocol driver not attached
ENOCSI	50	No CSI structure available
EL2HLT	51	Level 2 halted
EBADE	52	Invalid exchange
EBADR	53	Invalid request descriptor
EXFULL	54	Exchange full
ENOANO	55	No anode
EBADRQC	56	Invalid request code
EBADSLT	57	Invalid slot
EDEADLOCK	same as EDEADLK	-
EBFONT	59	Bad font file format
ENOSTR	60	Device not a stream
ENODATA	61	No data available
ETIME	62	Timer expired
ENOSR	63	Out of streams resources
ENONET	64	Machine is not on the network
ENOPKG	65	Package not installed
EREMOTE	66	Object is remote
ENOLINK	67	Link has been severed
EADV	68	Advertise error
ESRMNT	69	Srmount error
ECOMM	70	Communication error on send

EPROTO	71	Protocol error
EMULTIHOP	72	Multihop attempted
EDOTDOT	73	RFS specific error
EBADMSG	74	Not a data message
EOVERFLOW	75	Value too large for defined data type
ENOTUNIQ	76	Name not unique on network
EBADFD	77	File descriptor in bad state
EREMCHG	78	Remote address changed
ELIBACC	79	Can not access a needed shared library
ELIBBAD	80	Accessing a corrupted shared library
ELIBSCN	81	.lib section in a.out corrupted
ELIBMAX	82	Linking in too many shared libraries
ELIBEXEC	83	Cannot exec a shared library directly
EILSEQ	84	Illegal byte sequence
ERESTART	85	Interrupted system call should be restarted
ESTRPIPE	86	Streams pipe error
EUSERS	87	Too many users
ENOTSOCK	88	Socket operation on non-socket
EDESTADDRREQ	89	Destination address required
EMSGSIZE	90	Message too long
EPROTOTYPE	91	Protocol wrong type for socket
ENOPROTOOPT	92	Protocol not available
EPROTONOSUPPORT	93	Protocol not supported
ESOCKTNOSUPPORT	94	Socket type not supported
EOPNOTSUPP	95	Operation not supported on transport endpoint
EPFNOSUPPORT	96	Protocol family not supported
EAFNOSUPPORT	97	Address family not supported by protocol
EADDRINUSE	98	Address already in use
EADDRNOTAVAIL	99	Cannot assign requested address
ENETDOWN	100	Network is down
ENETUNREACH	101	Network is unreachable
ENETRESET	102	Network dropped connection because of reset
ECONNABORTED	103	Software caused connection abort
ECONNRESET	104	Connection reset by peer
ENOBUFS	105	No buffer space available
EISCONN	106	Transport endpoint is already connected
ENOTCONN	107	Transport endpoint is not connected

ESHUTDOWN	108	Cannot send after transport endpoint shutdown
ETOOMANYREFS	109	Too many references: cannot splice
ETIMEDOUT	110	Connection timed out
ECONNREFUSED	111	Connection refused
EHOSTDOWN	112	Host is down
EHOSTUNREACH	113	No route to host
EALREADY	114	Operation already in progress
EINPROGRESS	115	Operation now in progress
ESTALE	116	Stale NFS file handle
EUCLEAN	117	Structure needs cleaning
ENOTNAM	118	Not a XENIX named type file
ENAVAIL	119	No XENIX semaphores available
EISNAM	120	Is a named type file
EREMOTEIO	121	Remote I/O error
EDQUOT	122	Quota exceeded
ENOMEDIUM	123	No medium found
EMEDIUMTYPE	124	Wrong medium type

Appendix C: Linux Kernel Calls

This is a list of Linux kernel calls from section 2 of the manual.

_exit - terminate the current process
_llseek - reposition read/write file offset
_newselect - NQS
sysctl - read/write system parameters
accept - accept a connection on a socket
access - check user's permissions for a file
acct - switch process accounting on or off
adjtimex - tune kernel clock
afs_syscall - unimplemented
alarm - set an alarm clock for delivery of a signal
bdflush - start, flush, or tune buffer-dirty-flush daemon
bind - bind a name to a socket
break - unimplemented
brk - change data segment size
cacheflush - (MIPS) flush contents of instruction and/or data cache
chdir - change working directory
chmod - change permissions of a file
chown - change ownership of a file
chroot - change root directory
__clone - create a child process for multithreading
close - close a file descriptor
connect - initiate a connection on a socket
creat - open and possibly create a file or device
create_module - create a loadable module entry
delete_module - delete a loadable module entry
dup - duplicate a file descriptor
dup2 - duplicate a file descriptor
execve - execute program
exit - cause normal program termination
fchdir - change working directory
fchmod - change permissions of a file
fchown - change ownership of a file
fcntl - manipulate file descriptor
fdatasync - synchronize a file's in-core data with that on disk
flock - apply or remove an advisory lock on an open file
fork - create a child process
fstat - get file status
fstatfs - get file system statistics
fsync - synchronize a file's complete in-core state with that on disk
ftruncate - truncate a file to a specified length
get_kernel_syms - retrieve exported kernel and module symbols
getdents - get directory entries
getdomainname - get domain name
getdtablesize - get descriptor table size
getgid - get group identity
geteuid - get user identity
getgid - get group identity
getgroups - get/set list of supplementary group
gethostid - get the unique identifier of the current host
gethostnamee - get host name
getitimer - get value of an interval timer
getpagesize - get system page size
getpeername - get name of connected peer
getpgid - get process group

getpgrp - get process group
getpid - get process identification
getppid - get process identification
getpriority - get/set program scheduling priority
getresgid - get real, effective and saved group ID
getresuid - get real, effective and saved user ID
getrlimit - get resource limits
getrusage - get resource limits
getsid - get session ID
getsockname - get socket name
getsockopt - get options on sockets
gettimeofday - get time
getuid - get user identity
gtty - unimplemented
idle - make process 0 idle
init_module - initialize a loadable module entry
ioctl - control device
ioperm - set port input/output permissions
iopl - change I/O privilege level
ipc - System V IPC system calls
kill - send signal to a process
killpg - send signal to a process group
lchown - change ownership of a file
link - make a new name for a file
listen - listen for connections on a socket
llseek - reposition read/write file offset
lock - unimplemented
lseek - reposition read/write file offset
lstat - get file status
mkdir - create a directory
mknod - create a directory or special or ordinary file
mlock - disable paging for some parts of memory
mlockall - disable paging for calling process
mmap - map files or devices into memory
modify_ldt - get or set ldt
mount - mount and unmount filesystems.
mprotect - control allowable accesses to a region of memory
mpx - unimplemented
mremap - re-map a virtual memory address
msgctl - message control operations
msgget - get a message queue identifier
msgrcv - receive a messsage
msgsnd - send a message
msync - synchronize a file with a memory map
munlock - reenable paging for some parts of memory
munlockall - reenable paging for calling process
munmap - unmap files or devices into memory
nanosleep - pause execution for a specified time
nfsservctl - syscall interface to kernel nfs daemon
nice - change process priority
oldfstat - obsolete
oldlstat - obsolete
oldolduname - obsolete
oldstat - obsolete
olduname - obsolete
open - open a file or device
outb, outw, outl - port output macros
pause - wait for signal

personality - set the process execution domain
pipe - create pipe
poll - wait for some event on a file descriptor
prctl - operations on a process
prof - unimplemented
ptrace - process trace
query_module - query the kernel for various bits pertaining to modules
quotactl - manipulate disk quotas
read - read from a file descriptor
readdir - read directory entry
readlink - read value of a symbolic link
readv - read a vector
reboot - reboot or enable/disable Ctrl-Alt-Del
recv - receive a message from a socket
recvfrom - receive a message from a socket
recvmsg - receive a message from a socket
rename - change the name or location of a file
rmdir - delete a directory
sbrk - change data segment size
sched_get_priority_max - get static priority range
sched_get_priority_min - get static priority range
sched_getparam - get scheduling parameters
sched_setscheduler - get schedule algorithm/parameters
sched_rr_get_interval - get the SCHED_RR interval for the named process
sched_setparam - set scheduling parameters
sched_setscheduler - set schedule algorithm/parameters
sched_yield - yield the processor
select - synchronous I/O multiplexing
semctl - semaphore control operations
semget - get a semaphore set identifier
semop - semaphore operations
send - send a message from a socket
sendfile - transfer data between file descriptors
sendmsg - send a message from a socket
sendto - send a message from a socket
setdomainname - set domain name
setegid - set effective group ID
seteuid - set effective user ID
setfsgid - set group identity used for file system checks
setfsuid - set user identity used for file system checks
setgid - set group identity
setgroups - set list of supplementary group
sethostid - set the unique identifier of the current host
sethostname - set host name
setitimer - get or set value of an interval timer
setpgid - set process group ID
setpgrp - set process group
setpriority - set program scheduling priority
setregid - set real group ID
setresgid - set real, effective and saved user
setresuid - set real, effective and saved user
setreuid - set real and / or effective user ID
setrlimit - set resource limits
setsid - creates a session and sets the process group ID
setsockopt - set options on sockets
settimeofday - get / set time
setuid - set user identity
setup - setup devices and file systems, mount root file (not available)

sgetmask - ANSI C signal handling
shmat - shared memory operations
shmctl - shared memory control
shmdt - shared memory operations
shmget - allocates a shared memory segment
shutdown - shut down part of a full-duplex connection
sigaction - change signal action
sigblock - change blocked signals
siggetmask - get blocked signals
sigmask - C macro to create signal masks
signal - install signal handler
sigpause - atomically release blocked signals and wait for interrupt
sigpending - examine pending signals
sigprocmask - change blocked signals
sigreturn - return from signal handler and cleanup stack
sigsetmask - set net group of blocked signals
sigsuspend - replace signal mask and suspend process
sigvec - obsolete
socket - create an endpoint for communication
socketcall - socket system calls entry point
socketpair - create a pair of connected sockets
ssetmask - NQS
stat - get file status
statfs - get file system statistics
stime - set time
stty - unimplemented
swapoff - stop swapping to file/device
swapon - start swapping to file/device
symlink - make a new name for a file
sync - commit buffer cache to disk
sysctl - read/write system parameters
sysfs - get file system type information
sysinfo - returns information on overall system statistics
syslog - read and/or clear kernel message ring buffer; set console_loglevel
time - get time in seconds
times - get process times
truncate - truncate a file to a specified length
umask - set file creation mask
umount - unmount filesystems
uname - get name and information about current kernel
unlink - delete a name and possibly the file it refers to
uselib - select shared library
ustat - get file system statistics
utime - change access and/or modification times of an inode
utimes - change access and/or modification times of an inode
vfork - alias for fork
vhangup - virtually hangup the current tty
vm86 - (Intel) enter virtual 8086 mode
vm86old - obsolete
wait - wait for process termination
wait3 - wait for process termination, BSD style
wait4 - wait for process termination, BSD style
waitpid - wait for process termination
write - write to a file descriptor
writev - read or write a vector

Appendix D: Signals

Be aware that the mapping of names to signals may be -to-one. There may be aliases. Also, for allsignal names that are not supported on the current systemthe value of the corresponding constant will be zero.

SIGHUP -- hangup
SIGINT -- interrupt (rubout)
SIGQUIT -- quit (ASCD FS)
SIGILL -- illegal instruction (not reset)
SIGTRAP -- trace trap (not reset)
SIGIOT -- IOT instruction
SIGABRT used by abort, SIGIOT in the future
SIGFPE -- floating point exception
SIGKILL -- kill (cannot be caught or ignored)
SIGBUS -- bus error
SIGSEGV -- segmentation violation
SIGPIPE -- write on a pipe with no one to read it
SIGALRM -- alarm clock
SIGTERM -- software termination signal from kill
SIGUSR1 -- user defined signal 1
SIGUSR2 -- user defined signal 2
SIGCLD -- child status change
SIGCHLD -- 4.3BSD's/POSIX name for SIGCLD
SIGWINCH -- window size change
SIGURG -- urgent condition on IO channel
SIGPOLL -- pollable event occurred
SIGIO -- input/output possible, SIGPOLL alias (Solaris)
SIGSTOP -- stop (cannot be caught or ignored)
SIGTSTP -- user stop requested from tty
SIGCONT -- stopped process has been continued
SIGTTIN -- background tty read attempted
SIGTTOU -- background tty write attempted
SIGVTALRM -- virtual timer expired
SIGPROF -- profiling timer expired
SIGXCPU -- CPU time limit exceeded
SIGXFSZ -- filesize limit exceeded
SIGUNUSED -- unused signal
SIGSTKFLT -- stack fault on coprocessor
SIGLOST -- Linux alias for SIGIO
SIGPWR -- Power failure

Appendix E: Ioctl parameters

man ioctl_list gives a list of operations and parameters for ioctl. This is a copy of that man page. Be aware that Linux on different hardware will have different numeric codes for ioctl operations.

// Introduction

This is Ioctl List 1.3.27, a list of ioctl calls in Linux/i386
kernel 1.3.27. It contains 421 ioctls from
/usr/include/{asm,linux}/*.h.

For each ioctl, I list its numerical value, its name, and its
argument type.

An argument type of 'const struct foo *' means the argument is
input to the kernel. 'struct foo *' means the kernel outputs the
argument.

If the kernel uses the argument for both input and output, this
is marked with // I-O.

Some ioctls take more arguments or return more values than a
single structure. These are marked // MORE and documented further
in a separate section.

This list is incomplete. It does not include:

-- Ioctls defined internal to the kernel ('scsi_ioctl.h').
-- Ioctls defined in modules distributed separately from the kernel.

And, of course, it may have errors and omissions.

// Main table.
// <include/asm-i386/socket.h>

0x00008901 FIOSETOWN const int *
0x00008902 SIOCSPGRP const int *
0x00008903 FIOGETOWN int *
0x00008904 SIOCGPGRP int *
0x00008905 SIOCATMARK int *
0x00008906 SIOCGSTAMP timeval *

// <include/asm-i386/termios.h>

0x00005401 TCGETS struct termios *
0x00005402 TCSETS const struct termios *
0x00005403 TCSETSW const struct termios *
0x00005404 TCSETSF const struct termios *
0x00005405 TCGETA struct termio *
0x00005406 TCSETA const struct termio *
0x00005407 TCSETAW const struct termio *
0x00005408 TCSETAF const struct termio *
0x00005409 TCSBRK int
0x0000540A TCXONC int
0x0000540B TCFLSH int
0x0000540C TIOCEXCL void
0x0000540D TIOCNXCL void
0x0000540E TIOCSCTTY int
0x0000540F TIOCGPGRP pid_t *
0x00005410 TIOCSPGRP const pid_t *
0x00005411 TIOCOUTQ int *

0x00005412 TIOCSTI const char *
0x00005413 TIOCGWINSZ const struct winsize *
0x00005414 TIOCSWINSZ struct winsize *
0x00005415 TIOCMGET int *
0x00005416 TIOCMBIS const int *
0x00005417 TIOCMBIC const int *
0x00005418 TIOCMSET const int *
0x00005419 TIOCGSOFTCAR int *
0x0000541A TIOCSSOFTCAR const int *
0x0000541B FIONREAD int *
0x0000541B TIOCINQ int *
0x0000541C TIOCLINUX const char *

// MORE

0x0000541D TIOCCONS void
0x0000541E TIOCGSERIAL struct serial_struct *
0x0000541F TIOCSSERIAL const struct serial_struct *
0x00005420 TIOCPKT const int *
0x00005421 FIONBIO const int *
0x00005422 TIOCNOTTY void
0x00005423 TIOCSETD const int *
0x00005424 TIOCGETD int *
0x00005425 TCSBRKP int
0x00005426 TIOCTTYGSTRUCT struct tty_struct *
0x00005450 FIONCLEX void
0x00005451 FIOCLEX void
0x00005452 FIOASYNC const int *
0x00005453 TIOCSERCONFIG void
0x00005454 TIOCSERGWILD int *
0x00005455 TIOCSERSWILD const int *
0x00005456 TIOCGLCKTRMIOS struct termios *
0x00005457 TIOCSLCKTRMIOS const struct temios *
0x00005458 TIOCSERGSTRUCT struct async_struct *
0x00005459 TIOCSERGETLSR int *
0x0000545A TIOCSERGETMULTI struct serial_multiport_struct *
0x0000545B TIOCSERSETMULTI const struct serial_multiport_struct *

// <include/linux/ax25.h>

0x000089E0 SIOCAX25GETUID const struct sockaddr_ax25 *
0x000089E1 SIOCAX25ADDUID const struct sockaddr_ax25 *
0x000089E2 SIOCAX25DELUID const struct sockaddr_ax25 *
0x000089E3 SIOCAX25NOUID const int *
0x000089E4 SIOCAX25DIGCTL const int *
0x000089E5 SIOCAX25GETPARMS struct ax25_parms_struct * // I-O
0x000089E6 SIOCAX25SETPARMS const struct ax25_parms-struct *

// <include/linux/cdk.h>

0x00007314
STL_BINTR
void

0x00007315
STL_BSTART
void

0x00007316 STL_BSTOP void
0x00007317 STL_BRESET void

// <include/linux/cdrom.h>

0x00005301 CDROMPAUSE void
0x00005302 CDROMRESUME void
0x00005303 CDROMPLAYMSF const struct cdrom_msf *
0x00005304 CDROMPLAYTRKIND const struct cdrom_ti *
0x00005305 CDROMREADTOCHDR struct cdrom_tochdr *
0x00005306 CDROMREADTOCENTRY struct cdrom_tocentry * // I-O
0x00005307 CDROMSTOP void
0x00005308 CDROMSTART void
0x00005309 CDROMEJECT void
0x0000530A CDROMVOLCTRL const struct cdrom_volctrl *
0x0000530B CDROMSUBCHNL struct cdrom_subchnl * // I-O
0x0000530C CDROMREADMODE2 const struct cdrom_msf * // MORE
0x0000530D CDROMREADMODE1 const struct cdrom_msf * // MORE
0x0000530E CDROMREADAUDIO const struct cdrom_read_audio * // MORE
0x0000530F CDROMEJECT_SW int
0x00005310 CDROMMULTISESSION struct cdrom_multisession * // I-O
0x00005311 CDROM_GET_UPC struct { char [8]; } *
0x00005312 CDROMRESET void
0x00005313 CDROMVOLREAD struct cdrom_volctrl *
0x00005314 CDROMREADRAW const struct cdrom_msf * // MORE
0x00005315 CDROMREADCOOKED const struct cdrom_msf * // MORE
0x00005316 CDROMSEEK const struct cdrom_msf *

// <include/linux/cm206.h>

0x00002000 CM206CTL_GET_STAT int
0x00002001 CM206CTL_GET_LAST_STAT int

// <include/linux/cyclades.h>

0x00435901 CYGETMON struct cyclades_monitor *
0x00435902 CYGETTHRESH int *
0x00435903 CYSETTHRESH int
0x00435904 CYGETDEFTHRESH int *
0x00435905 CYSETDEFTHRESH int
0x00435906 CYGETTIMEOUT int *
0x00435907 CYSETTIMEOUT int
0x00435908 CYGETDEFTIMEOUT int *
0x00435909 CYSETDEFTIMEOUT int

// <include/linux/ext2_fs.h>

0x80046601 EXT2_IOC_GETFLAGS int *
0x40046602 EXT2_IOC_SETFLAGS const int *
0x80047601 EXT2_IOC_GETVERSION int *
0x40047602 EXT2_IOC_SETVERSION const int *

// <include/linux/fd.h>

0x00000000 FDCLRPRM void
0x00000001 FDSETPRM const struct floppy_struct *
0x00000002 FDDEFPRM const struct floppy_struct *
0x00000003 FDGETPRM struct floppy_struct *
0x00000004 FDMSGON void
0x00000005 FDMSGOFF void
0x00000006 FDFMTBEG void
0x00000007 FDFMTTRK const struct format_descr *
0x00000008 FDFMTEND void
0x0000000A FDSETEMSGTRESH int
0x0000000B FDFLUSH void
0x0000000C FDSETMAXERRS const struct floppy_max_errors *
0x0000000E FDGETMAXERRS struct floppy_max_errors *
0x00000010 FDGETDRVTYP struct { char [16]; } *

0x00000014 FDSETDRVPRM const struct floppy_drive_params *
0x00000015 FDGETDRVPRM struct floppy_drive_params *
0x00000016 FDGETDRVSTAT struct floppy_drive_struct *
0x00000017 FDPOLLDRVSTAT struct floppy_drive_struct *
0x00000018 FDRESET int
0x00000019 FDGETFDCSTAT struct floppy_fdc_state *
0x0000001B FDWERRORCLR void
0x0000001C FDWERRORGET struct floppy_write_errors *
0x0000001E FDRAWCMD struct floppy_raw_cmd * // MORE I-O
0x00000028 FDTWADDLE void

// <include/linux/fs.h>

0x0000125D BLKROSET const int *
0x0000125E BLKROGET int *
0x0000125F BLKRRPART void
0x00001260 BLKGETSIZE int *
0x00001261 BLKFLSBUF void
0x00001262 BLKRASET int
0x00001263 BLKRAGET int *
0x00000001 FIBMAP int * // I-O
0x00000002 FIGETBSZ int *

// <include/linux/hdreg.h>

0x00000301 HDIO_GETGEO struct hd_geometry *
0x00000302 HDIO_GET_UNMASKINTR int *
0x00000304 HDIO_GET_MULTCOUNT int *
0x00000307 HDIO_GET_IDENTITY struct hd_driveid *
0x00000308 HDIO_GET_KEEPSETTINGS int *
0x00000309 HDIO_GET_CHIPSET int *
0x0000030A HDIO_GET_NOWERR int *
0x0000030B HDIO_GET_DMA int *
0x0000031F HDIO_DRIVE_CMD int * // I-O
0x00000321 HDIO_SET_MULTCOUNT int
0x00000322 HDIO_SET_UNMASKINTR int
0x00000323 HDIO_SET_KEEPSETTINGS int
0x00000324 HDIO_SET_CHIPSET int
0x00000325 HDIO_SET_NOWERR int
0x00000326 HDIO_SET_DMA int

// <include/linux/if_eql.h>

0x000089F0 EQL_ENSLAVE struct ifreq * // MORE I-O
0x000089F1 EQL_EMANCIPATE struct ifreq * // MORE I-O
0x000089F2 EQL_GETSLAVECFG struct ifreq * // MORE I-O
0x000089F3 EQL_SETSLAVECFG struct ifreq * // MORE I-O
0x000089F4 EQL_GETMASTRCFG struct ifreq * // MORE I-O
0x000089F5 EQL_SETMASTRCFG struct ifreq * // MORE I-O

// <include/linux/if_plip.h>

0x000089F0 SIOCDEVPLIP struct ifreq * // I-O

// <include/linux/if_ppp.h>

0x00005490 PPPIOCGFLAGS int *
0x00005491 PPPIOCSFLAGS const int *
0x00005492 PPPIOCGASYNCMAP int *
0x00005493 PPPIOCSASYNCMAP const int *
0x00005494 PPPIOCGUNIT int *
0x00005495 PPPIOCSINPSIG const int *
0x00005497 PPPIOCSDEBUG const int *

0x00005498 PPPIOCGDEBUG int *
0x00005499 PPPIOCGSTAT struct ppp_stats *
0x0000549A PPPIOCGTIME struct ppp_ddinfo *
0x0000549B PPPIOCGXASYNCMAP struct { int [8]; } *
0x0000549C PPPIOCSXASYNCMAP const struct { int [8]; } *
0x0000549D PPPIOCSMRU const int *
0x0000549E PPPIOCRASYNCMAP const int *
0x0000549F PPPIOCSMAXCID const int *

// <include/linux/ipx.h>

0x000089E0 SIOCAIPXITFCRT const char *
0x000089E1 SIOCAIPXPRISLT const char *
0x000089E2 SIOCIPXCFGDATA struct ipx_config_data *

// <include/linux/kd.h>

0x00004B60 GIO_FONT struct { char [8192]; } *
0x00004B61 PIO_FONT const struct { char [8192]; } *
0x00004B6B GIO_FONTX struct console_font_desc * // MORE I-O
0x00004B6C PIO_FONTX const struct console_font_desc * //MORE
0x00004B70 GIO_CMAP struct { char [48]; } *
0x00004B71 PIO_CMAP const struct { char [48]; }
0x00004B2F KIOCSOUND int
0x00004B30 KDMKTONE int
0x00004B31 KDGETLED char *
0x00004B32 KDSETLED int
0x00004B33 KDGKBTYPE char *
0x00004B34 KDADDIO int // MORE
0x00004B35 KDDELIO int // MORE
0x00004B36 KDENABIO void // MORE
0x00004B37 KDDISABIO void // MORE
0x00004B3A KDSETMODE int
0x00004B3B KDGETMODE int *
0x00004B3C KDMAPDISP void // MORE
0x00004B3D KDUNMAPDISP void // MORE
0x00004B40 GIO_SCRNMAP struct { char [E_TABSZ]; } *
0x00004B41 PIO_SCRNMAP const struct { char [E_TABSZ]; } *
0x00004B69 GIO_UNISCRNMAP struct { short [E_TABSZ]; } *
0x00004B6A PIO_UNISCRNMAP const struct { short [E_TABSZ]; } *
0x00004B66 GIO_UNIMAP struct unimapdesc * // MORE I-O
0x00004B67 PIO_UNIMAP const struct unimapdesc * // MORE
0x00004B68 PIO_UNIMAPCLR const struct unimapinit *
0x00004B44 KDGKBMODE int *
0x00004B45 KDSKBMODE int
0x00004B62 KDGKBMETA int *
0x00004B63 KDSKBMETA int
0x00004B64 KDGKBLED int *
0x00004B65 KDSKBLED int
0x00004B46 KDGKBENT struct kbentry * // I-O
0x00004B47 KDSKBENT const struct kbentry *
0x00004B48 KDGKBSENT struct kbsentry * // I-O
0x00004B49 KDSKBSENT const struct kbsentry *
0x00004B4A KDGKBDIACR struct kbdiacrs *
0x00004B4B KDSKBDIACR const struct kbdiacrs *
0x00004B4C KDGETKEYCODE struct kbkeycode * // I-O
0x00004B4D KDSETKEYCODE const struct kbkeycode *
0x00004B4E KDSIGACCEPT int

// <include/linux/lp.h>

0x00000601 LPCHAR int
0x00000602 LPTIME int

0x00000604 LPABORT int
0x00000605 LPSETIRQ int
0x00000606 LPGETIRQ int *
0x00000608 LPWAIT int
0x00000609 LPCAREFUL int
0x0000060A LPABORTOPEN int
0x0000060B LPGETSTATUS int *
0x0000060C LPRESET void
0x0000060D LPGETSTATS struct lp_stats *

// <include/linux/mroute.h>

0x000089E0 SIOCGETVIFCNT struct sioc_vif_req * // I-O
0x000089E1 SIOCGETSGCNT struct sioc_sg_req * // I-O

// <include/linux/mtio.h>

0x40086D01 MTIOCTOP const struct mtop *
0x801C6D02 MTIOCGET struct mtget *
0x80046D03 MTIOCPOS struct mtpos *
0x80206D04 MTIOCGETCONFIG struct mtconfiginfo *
0x40206D05 MTIOCSETCONFIG const struct mtconfiginfo *

// <include/linux/netrom.h>

0x000089E0 SIOCNRGETPARMS struct nr_parms_struct * // I-O
0x000089E1 SIOCNRSETPARMS const struct nr_parms_struct *
0x000089E2 SIOCNRDECOBS void
0x000089E3 SIOCNRRTCTL const int *

// <include/linux/sbpcd.h>

0x00009000 DDIOCSDBG const int *
0x00005382 CDROMAUDIOBUFSIZ int

// <include/linux/scc.h>

0x00005470 TIOCSCCINI void
0x00005471 TIOCCHANINI const struct scc_modem *
0x00005472 TIOCGKISS struct ioctl_command * // I-O
0x00005473 TIOCSKISS const struct ioctl_command *
0x00005474 TIOCSCCSTAT struct scc_stat *

// <include/linux/scsi.h>

0x00005382 SCSI_IOCTL_GET_IDLU struct { int [2]; } *
0x00005383 SCSI_IOCTL_TAGGED_ENABLE void
0x00005384 SCSI_IOCTL_TAGGED_DISABLE void
0x00005385 SCSI_IOCTL_PROBE_HOST const int // MORE

// <include/linux/smb_fs.h>

0x80027501 SMB_IOC_GETMOUNTUID uid_t *

// <include/linux/sockios.h>

0x0000890B SIOCADDRT const struct rtentry * // MORE
0x0000890C SIOCDELRT const struct rtentry * // MORE
0x00008910 SIOCGIFNAME char []
0x00008911 SIOCSIFLINK void
0x00008912 SIOCGIFCONF struct ifconf * // MORE I-O
0x00008913 SIOCGIFFLAGS struct ifreq * // I-O
0x00008914 SIOCSIFFLAGS const struct ifreq *

0x00008915 SIOCGIFADDR struct ifreq * // I-O
0x00008916 SIOCSIFADDR const struct ifreq *
0x00008917 SIOCGIFDSTADDR struct ifreq * // I-O
0x00008918 SIOCSIFDSTADDR const struct ifreq *
0x00008919 SIOCGIFBRDADDR struct ifreq * // I-O
0x0000891A SIOCSIFBRDADDR const struct ifreq *
0x0000891B SIOCGIFNETMASK struct ifreq * // I-O
0x0000891C SIOCSIFNETMASK const struct ifreq *
0x0000891D SIOCGIFMETRIC struct ifreq * // I-O
0x0000891E SIOCSIFMETRIC const struct ifreq *
0x0000891F SIOCGIFMEM struct ifreq * // I-O
0x00008920 SIOCSIFMEM const struct ifreq *
0x00008921 SIOCGIFMTU struct ifreq * // I-O
0x00008922 SIOCSIFMTU const struct ifreq *
0x00008923 OLD_SIOCGIFHWADDR struct ifreq * // I-O
0x00008924 SIOCSIFHWADDR const struct ifreq * // MORE
0x00008925 SIOCGIFENCAP int *
0x00008926 SIOCSIFENCAP const int *
0x00008927 SIOCGIFHWADDR struct ifreq * // I-O
0x00008929 SIOCGIFSLAVE void
0x00008930 SIOCSIFSLAVE void
0x00008931 SIOCADDMULTI const struct ifreq *
0x00008932 SIOCDELMULTI const struct ifreq *
0x00008940 SIOCADDRTOLD void
0x00008941 SIOCDELRTOLD void
0x00008950 SIOCDARP const struct arpreq *
0x00008951 SIOCGARP struct arpreq * // I-O
0x00008952 SIOCSARP const struct arpreq *
0x00008960 SIOCDRARP const struct arpreq *
0x00008961 SIOCGRARP struct arpreq * // I-O
0x00008962 SIOCSRARP const struct arpreq *
0x00008970 SIOCGIFMAP struct ifreq * // I-O
0x00008971 SIOCSIFMAP const struct ifreq *

// <include/linux/soundcard.h>

0x00005100 SNDCTL_SEQ_RESET void
0x00005101 SNDCTL_SEQ_SYNC void
0xC08C5102 SNDCTL_SYNTH_INFO struct synth_info * // I-O
0xC0045103 SNDCTL_SEQ_CTRLRATE int * // I-O
0x80045104 SNDCTL_SEQ_GETOUTCOUNT int *
0x80045105 SNDCTL_SEQ_GETINCOUNT int *
0x40045106 SNDCTL_SEQ_PERCMODE void
0x40285107 SNDCTL_FM_LOAD_INSTR const struct sbi_instrument *
0x40045108 SNDCTL_SEQ_TESTMIDI const int *
0x40045109 SNDCTL_SEQ_RESETSAMPLES const int *
0x8004510A SNDCTL_SEQ_NRSYNTHS int *
0x8004510B SNDCTL_SEQ_NRMIDIS int *
0xC074510C SNDCTL_MIDI_INFO midi_info * // I-O
0x4004510D SNDCTL_SEQ_THRESHOLD const int *
0xC004510E SNDCTL_SYNTH_MEMAVL int * // I-O
0x4004510F SNDCTL_FM_4OP_ENABLE const int *
0xCFB85110 SNDCTL_PMGR_ACCESS struct patmgr_info * // I-O
0x00005111 SNDCTL_SEQ_PANIC void
0x40085112 SNDCTL_SEQ_OUTOFBAND const struct seq_event_rec *
0xC0045401 SNDCTL_TMR_TIMEBASE int * // I-O
0x00005402 SNDCTL_TMR_START void
0x00005403 SNDCTL_TMR_STOP void
0x00005404 SNDCTL_TMR_CONTINUE void
0xC0045405 SNDCTL_TMR_TEMPO int * // I-O
0xC0045406 SNDCTL_TMR_SOURCE int * // I-O
0x40045407 SNDCTL_TMR_METRONOME const int *
0x40045408 SNDCTL_TMR_SELECT int * // I-O

0xCFB85001 SNDCTL_PMGR_IFACE struct patmgr_info * // I-O
0xC0046D00 SNDCTL_MIDI_PRETIME int * // I-O
0xC0046D01 SNDCTL_MIDI_MPUMODE const int *
0xC0216D02 SNDCTL_MIDI_MPUCMD struct mpu_command_rec * // I-O
0x00005000 SNDCTL_DSP_RESET void
0x00005001 SNDCTL_DSP_SYNC void
0xC0045002 SNDCTL_DSP_SPEED int * // I-O
0xC0045003 SNDCTL_DSP_STEREO int * // I-O
0xC0045004 SNDCTL_DSP_GETBLKSIZ int * // I-O
0xC0045006 SOUND_PCM_WRITE_CHANNELS int * // I-O
0xC0045007 SOUND_PCM_WRITE_FILTER int * // I-O
0x00005008 SNDCTL_DSP_POST void
0xC0045009 SNDCTL_DSP_SUBDIVIDE int * // I-O
0xC004500A SNDCTL_DSP_SETFRAGMENT int * // I-O
0x8004500B SNDCTL_DSP_GETFMTS int *
0xC0045005 SNDCTL_DSP_SETFMT int * // I-O
0x800C500C SNDCTL_DSP_GETOSPACE struct audio_buf_info *
0x800C500D SNDCTL_DSP_GETISPACE struct audio_buf_info *
0x0000500E SNDCTL_DSP_NONBLOCK void
0x80045002 SOUND_PCM_READ_RATE int *
0x80045006 SOUND_PCM_READ_CHANNELS int *
0x80045005 SOUND_PCM_READ_BITS int *
0x80045007 SOUND_PCM_READ_FILTER int *
0x00004300 SNDCTL_COPR_RESET void
0xCFB04301 SNDCTL_COPR_LOAD const struct copr_buffer *
0xC0144302 SNDCTL_COPR_RDATA struct copr_debug_buf * // I-O
0xC0144303 SNDCTL_COPR_RCODE struct copr_debug_buf * // I-O
0x40144304 SNDCTL_COPR_WDATA const struct copr_debug_buf *
0x40144305 SNDCTL_COPR_WCODE const struct copr_debug_buf *
0xC0144306 SNDCTL_COPR_RUN struct copr_debug_buf * // I-O
0xC0144307 SNDCTL_COPR_HALT struct copr_debug_buf * // I-O

0x4FA44308 SNDCTL_COPR_SENDMSG const struct copr_msg *
0x8FA44309 SNDCTL_COPR_RCVMSG struct copr_msg *

0x80044D00 SOUND_MIXER_READ_VOLUME int *
0x80044D01 SOUND_MIXER_READ_BASS int *
0x80044D02 SOUND_MIXER_READ_TREBLE int *
0x80044D03 SOUND_MIXER_READ_SYNTH int *
0x80044D04 SOUND_MIXER_READ_PCM int *
0x80044D05 SOUND_MIXER_READ_SPEAKER int *
0x80044D06 SOUND_MIXER_READ_LINE int *
0x80044D07 SOUND_MIXER_READ_MIC int *
0x80044D08 SOUND_MIXER_READ_CD int *
0x80044D09 SOUND_MIXER_READ_IMIX int *
0x80044D0A SOUND_MIXER_READ_ALTPCM int *
0x80044D0B SOUND_MIXER_READ_RECLEV int *
0x80044D0C SOUND_MIXER_READ_IGAIN int *
0x80044D0D SOUND_MIXER_READ_OGAIN int *
0x80044D0E SOUND_MIXER_READ_LINE1 int *
0x80044D0F SOUND_MIXER_READ_LINE2 int *
0x80044D10 SOUND_MIXER_READ_LINE3 int *
0x80044D1C SOUND_MIXER_READ_MUTE int *
0x80044D1D SOUND_MIXER_READ_ENHANCE int *
0x80044D1E SOUND_MIXER_READ_LOUD int *
0x80044DFF SOUND_MIXER_READ_RECSRC int *
0x80044DFE SOUND_MIXER_READ_DEVMASK int *
0x80044DFD SOUND_MIXER_READ_RECMASK int *
0x80044DFB SOUND_MIXER_READ_STEREODEVS int *
0x80044DFC SOUND_MIXER_READ_CAPS int *
0xC0044D00 SOUND_MIXER_WRITE_VOLUME int * // I-O
0xC0044D01 SOUND_MIXER_WRITE_BASS int * // I-O
0xC0044D02 SOUND_MIXER_WRITE_TREBLE int * // I-O

0xC0044D03 SOUND_MIXER_WRITE_SYNTH int * // I-O
0xC0044D04 SOUND_MIXER_WRITE_PCM int * // I-O
0xC0044D05 SOUND_MIXER_WRITE_SPEAKER int * // I-O
0xC0044D06 SOUND_MIXER_WRITE_LINE int * // I-O
0xC0044D07 SOUND_MIXER_WRITE_MIC int * // I-O
0xC0044D08 SOUND_MIXER_WRITE_CD int * // I-O
0xC0044D09 SOUND_MIXER_WRITE_IMIX int * // I-O
0xC0044D0A SOUND_MIXER_WRITE_ALTPCM int * // I-O
0xC0044D0B SOUND_MIXER_WRITE_RECLEV int * // I-O
0xC0044D0C SOUND_MIXER_WRITE_IGAIN int * // I-O
0xC0044D0D SOUND_MIXER_WRITE_OGAIN int * // I-O
0xC0044D0E SOUND_MIXER_WRITE_LINE1 int * // I-O
0xC0044D0F SOUND_MIXER_WRITE_LINE2 int * // I-O
0xC0044D10 SOUND_MIXER_WRITE_LINE3 int * // I-O
0xC0044D1C SOUND_MIXER_WRITE_MUTE int * // I-O
0xC0044D1D SOUND_MIXER_WRITE_ENHANCE int * // I-O
0xC0044D1E SOUND_MIXER_WRITE_LOUD int * // I-O
0xC0044DFF SOUND_MIXER_WRITE_RECSRC int * // I-O

// <include/linux/umsdos_fs.h>

0x000004D2 UMSDOS_READDIR_DOS struct umsdos_ioctl * // I-O
0x000004D3 UMSDOS_UNLINK_DOS const struct umsdos_ioctl *
0x000004D4 UMSDOS_RMDIR_DOS const struct umsdos_ioctl *
0x000004D5 UMSDOS_STAT_DOS struct umsdos_ioctl * // I-O
0x000004D6 UMSDOS_CREAT_EMD const struct umsdos_ioctl *

0x000004D7 UMSDOS_UNLINK_EMD const struct umsdos_ioctl *
0x000004D8 UMSDOS_READDIR_EMD struct umsdos_ioctl * // I-O
0x000004D9 UMSDOS_GETVERSION struct umsdos_ioctl *
0x000004DA UMSDOS_INIT_EMD void
0x000004DB UMSDOS_DOS_SETUP const struct umsdos_ioctl *
0x000004DC UMSDOS_RENAME_DOS const struct umsdos_ioctl *

// <include/linux/vt.h>

0x00005600 VT_OPENQRY int *
0x00005601 VT_GETMODE struct vt_mode *
0x00005602 VT_SETMODE const struct vt_mode *
0x00005603 VT_GETSTATE struct vt_stat *
0x00005604 VT_SENDSIG void
0x00005605 VT_RELDISP int
0x00005606 VT_ACTIVATE int
0x00005607 VT_WAITACTIVE int
0x00005608 VT_DISALLOCATE int
0x00005609 VT_RESIZE const struct vt_sizes *
0x0000560A VT_RESIZEX const struct vt_consize *

// More arguments.

Some ioctl's take a pointer to a structure which contains
additional pointers. These are documented here in alphabetical
order. CDROMREADAUDIO takes an input pointer 'const struct
cdrom_read_audio *'. The 'buf' field points to an output buffer of
length CDROMREADCOOKED, CDROMREADMODE1, CDROMREADMODE2, and
CDROMREADRAW take an input pointer 'const struct cdrom_msf *'.
They use the same pointer as an output pointer to 'char []'. The length
varies by request. For CDROMREADMODE1, most drivers use
'CD_FRAMESIZE', but the Optics Storage driver uses 'OPT_BLOCKSIZE'
instead (both have the numerical value 2048).

CDROMREADCOOKED char [CD_FRAMESIZE]
CDROMREADMODE1 char [CD_FRAMESIZE or OPT_BLOCKSIZE]

CDROMREADMODE2 char [CD_FRAMESIZE_RAW0]
CDROMREADRAW char [CD_FRAMESIZE_RAW]

EQL_ENSLAVE, EQL_EMANCIPATE, EQL_GETSLAVECFG, EQL_SETSLAVECFG,
EQL_GETMASTERCFG, and EQL_SETMASTERCFG take a 'struct ifreq
*'. The 'ifr_data' field is a pointer to another structure as
follows:

EQL_ENSLAVE const struct slaving_request *
EQL_EMANCIPATE const struct slaving_request *
EQL_GETSLAVECFG struct slave_config * // I-O
EQL_SETSLAVECFG const struct slave_config *
EQL_GETMASTERCFG struct master_config *
EQL_SETMASTERCFG const struct master_config *

FDRAWCMD takes a 'struct floppy raw_cmd *'. If 'flags & FD_RAW_WRITE'
is non-zero, then 'data' points to an input buffer of length 'length'.
If 'flags & FD_RAW_READ' is non-zero, then 'data' points to an output
buffer of length 'length'.
GIO_FONTX and PIO_FONTX take a 'struct console_font_desc *' or a
a buffer of 'char [charcount]'. This is an output buffer for GIO_FONTX
and an input buffer for PIO_FONTX.

GIO_UNIMAP and PIO_UNIMAP take a 'struct unimapdesc *' or a
of 'struct unipair [entry_ct]'. This is an output buffer for GIO_UNIMAP
and an input buffer for PIO_UNIMAP.

KDADDIO, KDDELIO, KDDISABIO, and KDENABIO enable or disable access to
I/O ports. They are essentially alternate interfaces to 'ioperm'.
KDMAPDISP and KDUNMAPDISP enable or disable memory mappings or I/O port
access. They are not implemented in the kernel.

SCSI_IOCTL_PROBE_HOST takes an input pointer 'const int *', which is a
length. It uses the same pointer as an output pointer to a 'char []'
buffer of this length.

SIOCADDRT and SIOCDELRT take an input pointer whose type depends on
the protocol:

protocols const struct rtentry *
AX.25 const struct ax25_route *
NET/ROM const struct nr_route_struct *

SIOCGIFCONF takes a 'struct ifconf *'. The 'ifc_buf' field points to a
buffer of length 'ifc_len' bytes, into which the kernel writes a list of
type 'struct ifreq []'.

SIOCSIFHWADDR takes an input pointer whose type depends on the protocol:

Most protocols; const struct ifreq *
AX.25 const char [AX25_ADDR_LEN]

TIOCLINUX takes a 'const char *'. It uses this to distinguish several
independent sub-cases. In the table below, 'N + foo' means 'foo' after
an N-byte pad. 'struct selection' is implicitly defined in

TIOCLINUX-2 1 + const struct selection *
TIOCLINUX-3 void
TIOCLINUX-4 void
TIOCLINUX-5 4 + const struct { long [8]; } *
TIOCLINUX-6 char *
TIOCLINUX-7 char *
TIOCLINUX-10 1 + const char *

// Duplicate ioctls

This list does not include ioctls in the range SIOCDEVPRIVATE and SIOCPROTOPRIVATE.

0x00000001 FDSETPRM FIBMAP
0x00000002 FDDEFPRM FIGETBSZ
0x00005382 CDROMAUDIOBUFSIZ SCSI_IOCTL_GET_IDLUN
0x00005402 SNDCTL_TMR_START TCSETS
0x00005403 SNDCTL_TMR_STOP TCSETSW
0x00005404 SNDCTL_TMR_CONTINUE
TCSETSF

Appendix F: Overview of GNAT Packages

This is an overview of some of the more useful packages included with GCC Ada. There are more than 300 standard packages, and new versions of Ada will have additional packages that may not be summarized here:

File	*Package*	*Description*
a-astaco	Ada.Asynchronous_Task_Control	Unimplemented
a-caldel	Ada.Calendar.Delays	Sleeping using Calendar types
a-acalend	Ada.Calendar	Standard Ada Caledar package
a-chahan	Ada.Characters.Handling	Standard Ada character handling package
a-chlat1	Ada.Characters.Latin_1	Standard Latin 1 Character set definition
a-coliea	Ada.Command_Line.Environment	Standard Ada environment package
a-colire	Ada.Command_Line.Remove	Unset environment variables
a-comlin	Ada.Command_Line	Standard Ada command arguments package
a-cwila1	Ada.Characters.Wide_Latin_1	Standard Ada Latin 1 Wide character set
a-decima	Ada.Decimal	Limits and def'ns for Decimal types
a-adiocs	Ada.Direct_IO.C_Streams	Generic package for reading/writing C direct files
a-adireio	Ada.Direct_IO	Standard Ada generic direct I/O package
a-dynpri	Ada.Dynamic_Priorities	Changing task priorities on-the-fly
a-except	Ada.Exceptions	Standard Ada exception handling package
a-exctr	Ada.Exceptions.Traceback	Support for exception tracebacks
a-filico	Ada.Finalization.List_Controller	Support for controlled tagged records
a-finali	Ada.Finalization	Standard Ada controlled tagged record package
a-flteio	Ada.Float_Text_IO	Instantiated Text_IO for floats
a-fwteio	Ada.Float_WideText_IO	Instantiated Wide_Text_IO for floats
a-inteio	Ada.Integer_Text_IO	Instantiated Text_IO for integers
a-interr	Ada.Interrupts	Standard Ada signal handling package
a-intnam	Ada.Interrupts.Names	Linux signal names
a-ioexce	Ada.IO_Exceptions	I/O exceptions used in std packages
a-iwteio	Ada.Integer_Wide_Text_IO	Instantiated Wide_Text_IO for integers
a-lfteio	Ada.Long_Float_Text_IO	Instantiated Text_IO for long floats
a-lfwtio	Ada.Long_Float_Wide_Text_IO	Instantiated Wide_Text_IO for long floats

a-liteio	Ada.Long_Integer_Text_IO	Instantiated Text_IO for long integers
a-liwtio	Ada.Long_Integer_Wide_Text_IO	Instantiated Wide_Text_IO for long integers
a-llftio	Ada.Long_Long_Float_Text_IO	Instantiated Text_IO for long long floats
a-llfwti	Ada.Long_Long_Float_Wide_Text_IO	Inst. Wide_Text_IO for long long floats
a-llitio	Ada.Long_Long_Integer_Text_IO	Inst. Text_IO for long long integers
a-lliwti	Ada.Long_Long_Integer_Wide_Text_IO	Inst. Wide_Text_IO for long long integers
a-ncelfu	Ada.Numerics.Complex_Elementary_Function	Inst. of std ops for complex nbrs
a-ngcefu	Ada.Numerics.Generic_Complex_Elementary_Functions	Generic package of std ops for complex nbrs
a-ngcoty	Ada.Numerics.Generic_Complex_Types	Generic complex numbers package
a-ngelfu	Ada.Numerics.Generic_Elementary_Functions	Generic std ops for complex numbers
a-nlcefu	Ada.Numerics.Long_Complex_Elementary_Functions	Inst. of std ops for long complex nbrs
a-nlcoty	Ada.Numerics.Long_Complex_Types	Instantiation of long float complex nbrs
a-nlelfu	Ada.Numerics.Long_Elementary_Functions	Instantiation of std ops for long floats
a-nllcef	Ada.Numerics.Long_Long_Complex_Elementary_Functions	Inst. of std ops for long long complex nbrs
a-nllcty	Ada.Numerics.Long_Long_Complex_Types	Instantiation of long long float complex nbrs
a-nllefu	Ada.Numerics.Long_Long_Elementary_Functions	Inst. of std ops for long long floats
a-nscefu	Ada.Numerics.Short_Complex_Elementary_Functions	Inst. of std ops for short float complex nbrs
a-nscoty	Ada.Numerics.Generic_Complex_Types	Instantiation of short float complex nbrs
a-nselfu	Ada.Numerics.Short_Elementary_Functions	Inst. of std ops for short floats
a-nucoty	Ada.Numerics.Complex_Types	Instantiation of float complex numbers
a-nudira	Ada.Numerics.Discrete_Random	Generic integer random number package
a-nuelfu	Ada.Numerics.Elementary_Function	Inst. of std ops for float complex nbrs
a-nuflra	Ada.Numerics.Float_Random	Floating point random number package
a-numaux	Ada.Numerics.Aux	Internal use
a-numeri	Ada.Numerics	Defn's of Pi and epsilon
a-reatim	Ada.Real_Time	Real-time timing declarations
a-retide	Ada.Real_Time.Delays	Sleeping using real-time types

a-sequio	Ada.Sequential_IO	Standard Ada generic sequential I/O package
a-sfteio	Ada.Short_Float_Text_IO	Instantiated Text_IO package for short floats
a-sfwtio	Ada.Short_Float_Wide_Text_IO	Instantiated Wide_Text_IO package for short floats
a-siocst	Ada.Sequential_IO.C_Streams	Generic package for reading/writing sequential C files
a-siteio	Ada.Short_Integer_Text_IO	Instantiated Text_IO package for short integers
a-siwtio	Ada.Short_Integer_Wide_Text_IO	Inst. Wide_Text_IO package for short integers
a-ssicst	Ada.Streams.Stream_IO.C_Streams	Package for reading/writing C streams
a-ssitio	Ada.Short_Short_Integer_Text_IO	Inst. Text_IO package for short short integers
a-ssiwti	Ada.Short_Short_Integer_ Wide_Text_IO	Inst. Wide_Text_IO package for short short integers
a-stmaco	Ada.Strings.Maps.Constants	Upper_Set, Lower_Set and other char mappings
a-storio	Ada.Storage_IO	-
a-strbou	Ada.Strings.Bounded	Standard Ada bounded strings package
a-stream	Ada.Streams	Standard Ada streams package
a-strfix	Ada.Strings.Fixed	Standard Ada fixed strings package
a-string	Ada.Strings	Standard Ada string defn's
a-strmap	Ada.Strings.Maps	Standard Ada string mapping package
a-strsea	Ada.Strings.Search	Internal Use
a-strunb	Ada.Strings.Unbounded	Standard Ada unbounded strings package
a-ststio	Ada.Streams.Stream_IO	Standard Ada streams I/O package
a-stunau	Ada.Streams.Unbounded.Aux	Additional unbounded string subprograms
a-stwibo	Ada.Strings.Wide_Bounded	Wide bounded strings package
a-stwifi	Ada.Strings.Wide_Fixed	Wide fixed strings package
a-stwima	Ada.Strings.Wide_Maps	Wide version of strings.maps
a-stwise	Ada.Strings.Wide_Search	Internal Use
a-stwiun	Ada.Strings.Wide_Unbounded	Wide unbounded strings package
a-suteio	Ada.Strings.Unbounded.Text_IO	Unbounded strings package
a-swmwco	Ada.Strings.Wide_Maps.Wide_Constant	Upper_Set, Lower_Set and other wide char mappings
a-swuwti	Ada.Strings.Wide_Unbounded.Wide_Text_ IO	Wide unbounded strings package

a-sytaco	Ada.Synchronous_Task_Control	Subprograms to synchronize tasks
a-tags	Ada.Tags	Standard Ada tag package
a-tasatt	Ada.Task_Attributes	Set/get task attributes
a-taside	Ada.Task_Identification	Task ID package
a-teioed	Ada.Text_IO.Editing	Package for formatted Text_IO
a-textio	Ada.Text_IO	Standard generic Text_IO package
a-ticoau	Ada.Text_IO.Complex_Aux	Basic long long complex I/O package
a-ticoio	Ada.Text_IO.Complex_IO	Generic Text_IO package for complex numbers
a-tideau	Ada.Text_IO.Decimal_Aux	Internal Use
a-tideio	Ada.Text_IO.Decimal_IO	Internal Use
a-tienau	Ada.Text_IO.Enumeration_Aux	Internal Use
a-tienio	Ada.Text_IO.Enumeration_IO	Internal Use
a-tifiio	Ada.Text_IO.Fixed_IO	Internal Use
a-tiflau	Ada.Text_IO.Float_Aux	Internal Use
a-tiflio	Ada.Text_IO.Float_IO	Internal Use
a-tigeau	Ada.Text_IO.Generic_Aux	Internal Use
a-tiinau	Ada.Text_IO.Integer_Aux	Internal Use
a-tiinio	Ada.Text_IO.Integer_IO	Internal Use
a-timoau	Ada.Text_IO.Modular_Aux	Internal Use
a-timoio	Ada.Text_IO.Modular_IO	Internal Use
a-tiocst	Ada.Text_IO.C_Streams	Text_IO for reading/writing C text files
a-titest	Ada.Text_IO.Text_Streams	Text_IO stream definition
a-unccon	Ada.Unchecked_Conversion	Standard Ada unchecked conversions subprogram
a-uncdea	Ada.Unchecked_Deallocation	Standard Ada unchecked deallocation subprogram
a-witeio	Ada.Wide_Text_IO	Text_IO package for wide characters
a-wtcoau	Ada.Wide_Text_IO.Complex_Aux	Basic Text_IO package for long long float complex numbers
a-wtcoio	Ada.Wide_Text_IO.Complex_IO	Generic Wide_Text_IO package for complex numbers
a-wtcstr	Ada.Wide_Text_IO.C_Streams	Wide_Text_IO package for reading/writing wide C text files
a-wtdeau	Ada.Wide_Text_IO.Decimal_Aux	Internal Use
a-wtdeio	Ada.Wide_Text_IO.Decimal_IO	Internal Use
a-wtedit	Ada.Wide_Text_IO.Editing	Package for formatted Wide_Text_IO
a-wtenau	Ada.Wide_Text_IO.Enumeration_Aux	Internal Use
a-wtenio	Ada.Wide_Text_IO.Enumeration_IO	Internal Use

a-wtfiio	Ada.Wide_Text_IO.Fixed_IO	Internal Use
a-wtflau	Ada.Wide_Text_IO.Float_Aux	Internal Use
a-wtflio	Ada.Wide_Text_IO.Float_IO	Internal Use
a-wtgeau	Ada.Wide_Text_IO.Generic_Aux	Used by wide character IO generic packages
a-wtinau	Ada.Wide_Text_IO.Integer_Aux	Internal Use
a-wtinio	Ada.Wide_Text_IO.Integer_IO	Internal Use
a-wtmoau	Ada.Wide_Text_IO.Modular_Aux	Internal Use
a-wtmoio	Ada.Wide_Text_IO.Modular_IO	Internal Use
a-wttest	Ada.Wide_Text_IO.Text_Streams	Definition of wide text I/O streams
g-busora	GNAT.Bubble_Sort_A	Bubblesort using access types
g-busorg	GNAT.Bubble_Sort_G	Generic bubblesort package
g-calend	GNAT.Calendar	Ada.Calendar plus day of week, second duration, etc.
g-casuti	GNAT.Case_Util	Character case conversion without Characters.Handling
g-catiio	GNAT.Calendar.Time_IO	Formatted I/O for time values, like Linux strftime()
g-comlin	GNAT.Command_Line	More powerful than Ada.Command_Line, like Linux getopts()
g-curexc	GNAT.Current_Exception	DEC Ada 83 / VADS Ada style exception handling
g-debpoo	GNAT.Debug_Pools	Storage pool with allocation and dereference error checking
g-debuti	GNAT.Debug_Utilities	Program debugging utilities: eg. system address output
g-dirope	GNAT.Directory_Operations	Linux directory changing, creating, walking
g-except	GNAT.Exceptions	Ada predefined exceptions for pure packages
g-flocon	GNAT.Float_Control	Set the floating point processor back to the Gnat defaults
g-hesora	GNAT.Heap_Sort_A	Heapsort package using access types
g-hesorg	GNAT.Heap_Sort_G	Generic heapsort package
g-htable	GNAT.HTable	Generic hash table package
g-io	GNAT.IO	Text I/O for preelaborated packages
g-io_aux	GNAT.IO_Aux	Get_Line functions and file existence test for Text_IO
g-locfil	GNAT.Lock_Files	Package for locking files/directories with retry capability

g-os_lib	GNAT.OS_Lib	Package for common Linux O/S operations
g-regexp	GNAT.Regexp	Simple package for Linux globbing pattern matching and Ada BNF
g-regpat	GNAT.Regpat	Package providing full UNIX regular expression pattern matching
g-speche	GNAT.Spell_Checker	Check for a typo, similar to my Typo_Of in TextTools
g-spipat	GNAT.Spitbol.Pattern	Package providing SPITBOL pattern matching
g-spitbo	GNAT.Spitbol	SPITBOL string processing data structures
g-sptabo	GNAT.Spitbol.Table_Boolean	Boolean type SPITBOL table
g-sptain	GNAT.Spitbol.Table_Intege	Integer type SPITBOL table
g-sptavs	GNAT.Spitbol.Table_VString	Unbounded string type SPITBOL table
g-table	GNAT.Table	Dynamic one-dimensional arrays package
g-tasloc	GNAT.Task_Lock	Package for protecting critical regions in tasks
g-thread	GNAT.Threads	Import C threads as Ada tasks
g-traceb	GNAT.Traceback	Non-symbolic traceback support
g-trasym	GNAT.Traceback.Symbolic	Symbolic tracebacks
i-c	Interfaces.C	Standard Ada C interfacing package
i-cexten	Interfaces.C.Extensions	Additional C types not covered by Interfaces.C
i-cobol	Interfaces.COBOL	Standard Ada COBOL interfacing package
i-cpoin	Interfaces.C.Pointers	C style pointer arithmetic
i-cpp	Interfaces.CPP	GNAT C++ class interfacing package
i-csthre	Interfaces.C.Sthreads	Dummy package
i-cstrea	Interfaces.C_Streams	Thin binding to C sequential files
i-cstrin	Interfaces.C.Strings	GNAT C string operations
i-fortra	Interfaces.Fortran	Standard Ada Fortran interfacing package
i-os2err	Interfaces.OS2Lib.Errors	OS/2 error codes
i-os2lib	Interfaces.OS2Lib	OS/2 support
i-os2syn	Interfaces.OS2Lib.Synchronization	OS/2 support
i-os2th	Interfaces.OS2Lib.Threads	OS/2 support
i-pacdec	Interfaces.Packed_Decimal	Packed decimal fixed types support for Machine_Radix 10 computers
i-vxwork	Interfaces.VxWorks	VxWords API support

i-addimg	System.Address_Image	Function returning a system.address image
s-arit64	System.Arith_64	64 bit arithmetic with support for intermediate results > 64 bits
s-atacco	System.Address_To_Access_Conversions	Converting between simple pointers and access types
s-bitops.ads	System.Bit_Ops	Low-level bitwise operations for 1, 2 or 4 bytes
s-chepoo	System.Checked_Pools	Storage pool with a function called for any dereference
i-exngen	Exn_Float_Type	Generic function for signed integer exponentiation
s-pooglo	System.Pool_Global	normal heap for GNAT global access types
s-pooloc	System.Pool_Local	normal heap for GNAT local access types
s-powtab	System.Powten_Table	table of powers of 10
s-stoele	System.Storage_Elements	Standard Ada package

Glossary

A glossary of common Ada/Linux terms and defintions.

AARM The Annotated Ada Reference Manual contains the entire text of the Ada 95 standard (ISO/IEC 8652:1995(E)), plus various annotations. It is intended primarily for compiler writers, validation test writers, and other language lawyers. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

ACT Ada Core Technologies, an offshoot of New York University, developer of GNAT and the original code for GCC Ada. Provides commerical support for GNAT / GCC Ada

Ada 0X the working title of Ada 2005 before the language was completed.

Ada 9X the working title of Ada 95 before the language was completed.

ASIS The Ada Semantic Interface Specification is a layered vendor-independent open architecture. ASIS queries and services provide a consistent interface to information within the Ada compilation environment.

Distribution In Linux, a software bundle containing the Linux Kernel, X Windows and user software to create a complete system, assembled and supported by a third-party. e.g. Debian Linux, Red Hat Linux, etc.

Dynamic Polymorphism Polymorphism implemented at run-time using a "tag" to determine the type of item; tagged records, objects.

GCC Ada Free Ada support in the GNU Compiler Collection (GCC), a standard language starting with GCC 3.0.

GMGPL GNAT-Modified GPL. An extension of the GNU Public Licence 2 that also covers Ada generics (that is, templates) under the license.

GNAT GNU New York University Ada Translator. This refers to either the original version of GCC Ada that ran on a custom version of GCC (GNAT/GPL), or the modern commerical version of GCC Ada supplied by ACT (GNATPro).

GNAT/GCC Alias for GCC Ada.

GNAT/GPL Term used to describe the original GNAT, as opposed to the commercial versions.

GNAT Pro Commercial version of GCC Ada supplied by ACT.

Inheritance Creating new items containing an original item's features without changing the original item.

Kernel The core of the operating system, controlling devices, scheduling resources and providing security. e.g. the Linux kernel

libc The standard C function library (e.g. printf).

libgnat see RTL.

LRM is the abbreviated name of the Language Reference Manual, sometimes called Ada Reference Manual. "LRM" was often used in the days of Ada 83; "RM" or "rm95"

Multiple Inheritance Creating new items from two or more original item's features without changing the original item.

Polymorphism A means of factoring out differences amongst a collection of items so that programs may be written in terms of the common features.

RM see LRM.

RM95 see LRM.

RTL GCC Ada Run-Time Library, libgnat. Contains all standard packages (e.g. Text_IO) as well as support functions required to run an Ada program (exception handling, etc.)

Syscall System call. A kernel function call, as opposed to a function call to a library.

Static Polymorphism Polymorphism implemented at compile-time; generics.